SCHOOL OF INFORMATICS

EDINBURGH UNIVERSITY

PHD THESIS

Increasing the Efficacy of Automated Instruction Set Extension

---

Author:
Richard Vincent BENNETT

Supervisor:
Prof. Nigel TOPHAM

Secondary Supervisor:
Dr. Björn FRANKE

July 14, 2011

Abstract

The use of Instruction Set Extension (ISE) in customising embedded processors for a specific application has been studied extensively in recent years. The addition of a set of complex arithmetic instructions to a baseline core has proven to be a cost-effective means of meeting design performance requirements. This thesis proposes and evaluates a reconfigurable ISE implementation called "Configurable Flow Accelerators" (CFAs), a number of refinements to an existing Automated ISE (AISE) algorithm called "ISEGEN", and the effects of source form on AISE.

The CFA is demonstrated repeatedly to be a cost-effective design for ISE implementation. A temporal partitioning algorithm called "staggering" is proposed and demonstrated on average to reduce the area of CFA implementation by 37% for only an 8% reduction in acceleration.

This thesis then turns to concerns within the ISEGEN AISE algorithm. A methodology for finding a good static heuristic weighting vector for ISEGEN is proposed and demonstrated. Up to 100% of merit is shown to be lost or gained through the choice of vector. ISEGEN early-termination is introduced and shown to improve the runtime of the algorithm by up to 7.26x, and 5.82x on average. An extension to the ISEGEN heuristic to account for pipelining is proposed and evaluated, increasing acceleration by up to an additional 1.5x. An energy-aware heuristic is added to ISEGEN, which reduces the energy used by a CFA implementation of a set of ISEs by an average of 1.6x, up to 3.6x. This result directly contradicts the frequently espoused notion that "bigger is better" in ISE.

The last stretch of work in this thesis is concerned with source-level transformation: the effect of changing the representation of the application on the quality of the combined hardware-software solution. A methodology for combined exploration of source transformation and ISE is presented, and demonstrated to improve the acceleration of the result by an average of 35% versus ISE alone. Floating point is demonstrated to perform worse than fixed point, for all design concerns and applications studied here, regardless of ISEs employed.

Acknowledgements

Special thanks is given to the following people:

"A master can tell you what he expects of you. A teacher, though, awakens your own expectations."

– Patricia Neal

Nigel Topham; for the opportunities, understanding, and sage advice.
Björn Franke; for front-line support, patience, and keen insight.
Without both of your help I would not have been able to navigate the strange world of academic computer science.

"I love being married. It's so great to find that one special person you want to annoy for the rest of your life."

– Rita Rudner

To Kate Weaver-Bennett; for making me whole.

"A scientist's aim in a discussion with his colleagues is not to persuade, but to clarify."

– Leo Szilard

Oscar Almer, Igor Böhm, Christophe Dubach, Salman Khan, Stephen Kyle, Hugh Leather, Alastair Murray, Mike O'Boyle, Freddie Qu, Chris Thompson, George Tournavatis, and Marcela Zuluaga; for providing countless hours of productive discussion, countless further hours of unproductive discussion, and travelling companionship across the world.

"You are the bows from which your children as living arrows are sent forth."

– Kahlil Gibran

Paul and Wendy Bennett; for getting me as far as you did, and many words of encouragement.
Patricia Curtis and Brian Weaver; for raising your daughter, and for providing a level of support that astounds me on a daily basis.

"A good friend is a connection to life - a tie to the past, a road to the future, the key to sanity in a totally insane world."

–Lois Wyse

Luke Bennett, Gemma Bennett, Thomas James, Jonathan Gray, Tom Feist, Charles Leahy, Thomas Haddow, James Hanlon, Alexandra Nagy, Douglas Llewellyn, Barbara Hind, John Turner, Rachel McKerrow, Eugene Hopkinson, Andreas Learch, David Weigl, James Addison, Courtney Fox, Matthew Mossman, Ryan Barton, Erin Roherty, Taylor Seymour, Colin McEwan, Adam Lion, Jim Rutherford, Glynn Weaver, Amanda Weaver, and countless others.

This thesis is dedicated to the memories of Colin Bennett and Alan Smith

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. The following work with which this thesis shares content was published:

- R.V. Bennett, A.C. Murray, B. Franke, and N. Topham “Combining source-to-source transformations and processor instruction set extension for the automated design-space exploration of embedded systems”. In: *Proceedings of Languages Compilers and Technology for Embedded Systems (LCTES), 2007.*
- O. Almer, R.V. Bennett, I. Böhm, A.C. Murray, X. Qu, M. Zuluaga, B. Franke and N.P. Topham “An End-to-End Design Flow for Automated Instruction Set Extension and Complex Instruction Selection based on GCC”. In: *Proceedings of the 1st International Workshop on GCC Research Opportunities (GROW), 2009.*

(Richard Vincent Bennett)

Document Conventions

Throughout the text of this document, the use of the following terms and abbreviations are made use of in an effort to increase its accuracy and clarity.

- AISE - Automated Instruction Set Extension.
- ASIC - Application Specific Integrated Circuit.
- ASIP - Application Specific Instruction-set Processor.
- CCA - Configurable Compute Accelerator.
- CFA - Configurable Flow Accelerator.
- CDFG - Control and Data Flow Graph.
- DFG - Data Flow Graph.
- DII - Data Initiation Interval (issue latency).
- DSE - Design Space Exploration.
- FPGA - Field Programmable Gate Array.
- ILP - Integer Linear Programming.
- ISA - Instruction Set Architecture.
- ISE - Instruction Set Extension.
- LUT - Look Up Table.
- NoC - Network On Chip
- OLP - Operator Level Parallelism (sometimes referred to as Instruction Level Parallelism in other works, but herein we refer to the phenomenon at a higher level in the IR).
- ML - Machine Learning.
- RFU - Reconfigurable Functional Unit.
- SoC - System On Chip

The meaning of some of these terms is expanded in the text of the document where appropriate.

CONTENTS

1 INTRODUCTION

"The last thing one knows in constructing a work is what to put first."

– Blaise Pascal

We are now living in the Information Age, where those who cooperate with the legion of digital information services can improve both productivity and quality of life. There is a trend towards increasingly more personalised management and delivery of information. The markets have concluded the only logical outcome of this trend is the development of more ubiquitous and faster embedded computing. Moving away from the "Desktop Model" created by such companies as Microsoft, IBM, and Intel has begun to dominate the microelectronics market. The struggle between high performance and low cost will continue for the foreseeable future, and will only be mitigated with the introduction of new, more cost-effective design innovation and automation.

The progression of commercially adopted computer architecture has always been largely guided by two factors:

- The productivity of the applications programmer.
- The maximum throughput or speed of execution obtainable.

The maximisation of both of these factors is the principle concern, as they govern two critical financial concerns to stakeholders in application development, namely:

- The cost of developing the application.
- The size and quality or domain of applications which may be developed; hence the markets which may be exposed and exploited by application development.

Computers exist solely for the purpose of the applications which run on them. Such applications were originally written directly in the machine code, and later the assembly languages of the machines which they ran on. This small addition of mnemonics as assembly language was a small luxury afforded by the feasibility of creating assemblers, the possibility itself created by advances in the hardware architecture. This was the first iteration of the evolutionary cycle, wherein we connected streams of ones and zeroes to Hindu-Arabic numerals and letters in the Roman alphabet.

Next came compilers, and the ensuing language debate. Whilst the issue of which language to use in software design is far from resolved, at least in our particular branch of this story there is a clear winner in terms of tried and tested efficacy; the imperative language C. Imperative programming in general has several advantages over other language paradigms from a practical point of view, for example:

- Imperative programming lends itself to natural, object-oriented, behavioural descriptions.
- Imperative programming is readily mapped to RISC execution, which are essentially imperative execution engines.

For these reasons and more, the majority of compiler research efforts at present focus on object-oriented and/or imperative languages such as C, C++, and Java. As the hardware which runs a compiler evolves, the ability to map from our high level languages to complex computer systems grows also.

The success of RISC is largely as a product of the increase in compiler design complexity and the decrease in memory cost, allowing the necessary fine grain control. Further along the explicit control axis, one finds architectures such as VLIW. With VLIW more effort must be put into code quality and scheduling in order to obtain greater performance (speed) for less cost (power, die area) [1]. Around the time that VLIW architectures were being commercialised, the idea of Hardware/Software co-design [2] came to light as a good engineering formulation, when designing embedded systems with performance constraints. Design becomes a process of mapping between the hard and soft elements of the system; allocating hardware resources where performance is critical, and software otherwise especially where flexibility is required.

During the 1990's there was a move towards electronic system design, utilising a degree of design automation in order to perform tasks which were becoming less and less tractable to human engineers due to their sheer size and complexity. The Verilog language became the industry standard for ASIC design specification and verification, largely as a product of its support for fast simulation of a design which allowed for engineers to test their designs before they were committed to silicon. More importantly, the language was able to be directly synthesised, meaning that a single description of an ASIC could be used throughout the design process. From this basis comes the idea of an iterative process of design, whereby the concerns of a design are worked in from the top level of abstraction (the system level view) down to the bottom (silicon layout, i.e. GDS-II). This is achieved through compiling a language such as Verilog down to a lower level of abstraction, then examining what further efforts are needed to make the design meet the more complex requirements of the lower levels of abstraction.

The previous GPP stage of evolution has failed in recent years; we are unable to arbitrarily increase clock speed, and so we must map more hardware instead. General purpose execution has very much taken the high(-level) road, in that it looks to byte-code languages such as Java, and C#, choosing C or C++ only when performance is absolutely critical. This constant raising of the abstraction level makes an efficient mapping to thread-level parallelism more readily accessible to programmers, but is expensive. Over-specification in the multi-core dimension is both costly and hard to program. Methods such as clock gating [3] have been proposed to help reduce the problem in terms of power and energy efficiency, but these do nothing to address

the ever increasing GPP real-estate.

It is as of yet unclear whether the multi-core trend will ultimately be the saviour of GPP. The big players like AMD and Intel are embracing the multi-core idea, putting more and more cores on a chip in an attempt to exploit the available silicon. One fact remains that prevent GPPs from gaining extensive foothold in the embedded domain: their performance is extremely expensive in terms of both power and die area.

The techniques and skill that have gone into GPP design have been both forward-looking and effective in achieving their goal: performance at whatever cost. The trend towards low power, limited function, mobile devices does require a rethink of this entire methodology. Suddenly engineers are presented with a very concrete limit to the amount of silicon and energy that may be used. The situation is now that for the specific functions of a mobile device the execution speed is expected to be at least equal to GPP. Audio and video decoding, encryption, rendering in both 2D and 3D, and automated control purposes are all good examples of domains which are driving the need for faster, cheaper embedded processing.

The focus of this thesis is therefore that of embedded applications hardware-software co-design, that having been one basic remit of ICSA for the last nine years. The inclusion of the compiler into the hardware-software co-design process has brought about a new way of thinking about architectural design; that of using an application to guide the design process automatically based on features and structures used in its execution. This design methodology is now introduced in more depth in the following section.

1.1 Instruction Set Extension

Many methodologies for hardware-software co-design will include Instruction Set Extension (ISE) as a core component of their design space [4]. ISE is a technique for processor customisation wherein the architecture (and often also the microarchitecture) are extended specifically with the computing requirements of the application software in mind. The architectures used with ISE are most often Reduced Instruction Set Computers (RISC), and hence the complexity of the instructions in the baseline core will be low, maintaining the generality of the instruction set in question. Low-complexity RISC-style instructions are also generally not the most efficient way of executing a given application, and so additional instructions (ISEs) will be added to this baseline in order to provide a better fit between the application in question, and the target architecture. ISE design introduces a dependency between the software and the hardware, requiring that the application software or at least its critical kernels be somewhat stable before hardware design gets underway. More stability is generally required with larger ISEs; smaller ISEs can be retargeted more readily due to lower complexity. This is generally at odds with the "old-fashioned" embedded application development methodology, wherein a prefabricated target architecture would be selected on the basis of much more high-level specifications, such

as the potential for executing DSP or control-flow dominated algorithms well. ISEs are generally far more specific than this, although the selection of a suitable prefabricated baseline is still important. ISEs require fine-grain structural design decisions in order to properly exploit their potential. A number of algorithms have been created in order to identify ISEs automatically based on the source code of an application, handling the fine-grain decisions by way of heuristic merit functions. These algorithms have enabled the rapid construction of ISEs for a particular application, putting the problem into computational costs rather than engineering. Both AMD and Intel have recently produced ISEs in the form of XOP [5] and AVX [6] respectively, demonstrating the relevance of application-specific instructions even with GPPs. These GPP ISEs are not as complex as those considered for ASIPs, and focus more on vectorising common complex operations than creating instructions with a single purpose. In this thesis ISE is examined relative to low-complexity baseline RISC architectures.

1.2 The Problem

The major concern with using ISE when constructing a combined hardware-software design is that of the speed and quality of the design work that can be achieved. The major challenge for increasing the efficacy of ISE is to:

- Reduce the time taken to produce a working system (architecture, microarchitecture, and tool-chain) through design automation.
- Increase the quality (lower area, lower energy, higher acceleration) of the specialised system.

At the time of writing this thesis, there are still engineers employed at companies such as ARM and ST Microelectronics who perform application-specific ISE design on a manual basis. The manual approach represents the slowest but highest-quality trade-off in engineering cost versus design quality. Automated synthesis of ISEs including identification, implementation, and exploitation is an extremely important avenue of research, to reduce the manual effort and hence engineering cost of utilising the technique. It is the central goal of AISE research to increase the efficacy of the methodology until it equals, and eventually surpasses that possible by manual design. The following sub-sections give more detail on the concerns in question when considering automatically synthesised ISEs.

1.2.1 Engineering Time

The manual effort required in order to include ISEs in a design is considerable, and requires several distinct "deliverables" to be produced in order to be part of a practical design solution:

- The semantics of ISEs including instruction encoding (dependant on the baseline architecture), function (usually a collection of "data-flow" as directed graphs), and assembler

mnemonic syntax.

- An implementation of ISEs as a structural extension to the baseline architecture, usually via a pre-defined extension interface. These must implement the semantics defined above.

- Additional tool-chain support for mapping the application software to the extended hardware, and potentially simulation of the extended hardware for testing & verification.

Time to market constraints often dominate the engineering trade-offs made in the design, with embedded applications requiring a fast turn-around from the conception of the application to the realisation of the device. Automated ISE (AISE) accelerates not only the application in question but the generation of the above deliverables also. This makes AISE especially attractive where the automated process can derive instructions of a suitably high quality with respect to design concerns compared to the manual approach. The major limiting factor to AISE is the tractability of the search problem. In its most naive form the identification problem alone has a search space size of 2^n, where n is the number of operators in the basic block being analysed. This intractability of the search space as a whole gives rise to the need for heuristics and search-space pruning; two methodologies used extensively in AISE algorithms.

1.2.2 Acceleration

Acceleration of the application when executing on the extended architecture has for a long time been the principal concern of ISE methodologies, despite it being only one of several design concerns for real-world solutions. The individual ISEs will cover a number of operations that would otherwise have been performed via the baseline RISC instruction set. This in effect packs sets of operators together in a manner which exploits a combination of serial and parallel acceleration. Serial acceleration is possible when two or more operators in sequence have the ceiling of the sum of their latencies less than the sum of the latencies of the RISC instructions required to execute them. Parallel acceleration is the more classical case similar to instruction-level parallelism as considered in VLIW, wherein multiple operators are independent from one-another and have their inputs available in an overlapping time-frame; referred to henceforth as "Operator Level Parallelism" (OLP).

Second-order speedup effects are possible, such as reduction of register pressure and hence spill code due to ISE-internal values being passed by wire, rather than via the register set. ISEs can also be combined with scratch-pad memories, further reducing the pressure on the register file. When taken together, this plethora of potential customisation can lead to a significant acceleration of the application; assuming a suitable portion of the application lies in frequently executed loops which are of a form suitable for ISE mapping.

1.2.3 Area

The process of reducing the area that ISEs utilise has not been neglected by researchers. Whether area is FPGA slices or millimetres of silicon area is an important contributor to the bottom line. Higher silicon area can contribute two-fold to higher monetary costs:

- In silicon the area has a direct price-tag attached per unit consumed, although often there is a specific limit which engineers are encouraged to get as close to as possible but not exceed.
- The larger an ASIC design is, the more likely an individual unit is to contain a fabrication flaw which will render that unit useless or lesser-functioned.

Several factors interact to determine the merit of using FPGA instead:

- FPGA implementation is generally a lot more expensive than ASIC when considered in high-volume, as most application-specific designs are.
- Regardless of volume, the size of the hardware design will determine the size of the FPGA required to contain it.
- Baseline processors in FPGA can lead to better baseline performance, but ISEs within a standard LUT FPGA will be of very poor performance.
- Some FPGA units contain a selection of arithmetic units as hard macros, and so fitting the design not only to the area but to the FPGA microarchitecture becomes essential.

ISE in FPGA is generally of lower efficacy than ISE in ASIC standard-cell technology. There is no reason to discount FPGA entirely as industry is making good progress in increasing the number of hard-macros available and the routing hierarchies. This thesis concentrates on silicon implementation, and hence any reference to area is standard cell area, in a commercial 130nm process.

Resource-sharing is absolutely essential to improve the cost-benefit performance of ISE. There are two major scenarios in ISE to exploit through resource sharing: Inter-ISE resource sharing wherein units from one ISE are shared with another, and intra-ISE resource sharing, wherein units are time-multiplexed between temporally separate sections of the ISE. These two methods are henceforth referred to as “spatial” and “temporal” resource sharing. Another area reduction technique, less beneficial but still substantial enough to note is that of replacing units not on the critical path of an ISE with higher latency, lower area equivalents. This is considered here to be more of a concern to developers of products such as DesignWare, than those using them to produce ISEs.

ISE is inherently an *extension* technique and therefore will never reduce the area of the baseline core under extension. However, the design space as seen by application engineers

should include a number of potential baseline cores. ISE could produce a solution which when compared to an equivalently performing (acceleration and energy) general purpose core does have lesser area. This is due to a better fit between the hardware and software, meaning the hardware does not have to use generalised acceleration techniques such as out-of-order execution which can add a large amount of area. Dynamic scheduling hardware is a major contributor to the inefficiencies of GPP.

Regardless of whether area is represented by FPGA cells or silicon area, there will generally be an increase in power consumption for an increase in area. Power consumption alone is generally not of great importance other than ensuring the source is not overloaded by any peak consumption. Power though, is a direct contributor to energy consumption.

1.2.4 Energy

When purchasing a mobile device, one of the primary concerns is often the question of how long the battery will last. As mobile devices shrink in size and their functional complexity increases, this inevitably places more and more drain on batteries. The SoC components of a mobile device such as a cell phone is not necessarily the entirety of its energy consumption; analogue components also play a large part in many instances. The SoC components do consume a significant enough proportion to warrant addressing the energy costs of executing an application. In addition, newer techniques such as software radio are placing more and more of the functional burden on the digital elements; this will further push SoC components into the fore in energy considerations.

The scenario is further complicated by developments in process technology tending towards the lower (sub-micron and beyond) nodes, changing the rules of thumb engineers can use when designing circuits. The progress in fabrication makes the problem of balancing energy consumption with other concerns an ever more complex one. The major shift is in the ratio of dynamic versus static power consumption, with dynamic power largely being handled with clock gating and static power requiring ever more audacious techniques to reduce it as it becomes more and more the dominant factor. This thesis largely focuses on the 130nm technology node, and leaves tackling newer process sizes to further work.

ISE can be used in reducing the energy consumption of an ASIP, but the process must be carefully managed so that the ISEs actually benefit rather than reduce the energy performance. If large ISEs are employed, the possibility that they will not lend themselves well to resource sharing is a real concern. Without efficient resource sharing, ISEs can bloat a design and result in a considerably worsened cost-benefit than just selecting a more capable baseline core. The energy cost is one of the concerns which can be worsened by poorly designed or implemented ISEs. In order to improve the efficacy in terms of energy consumption therefore, we need both a microarchitecture capable of efficient resource sharing and an algorithm capable of measuring and manipulating those microarchitectural elements with an impact on energy.

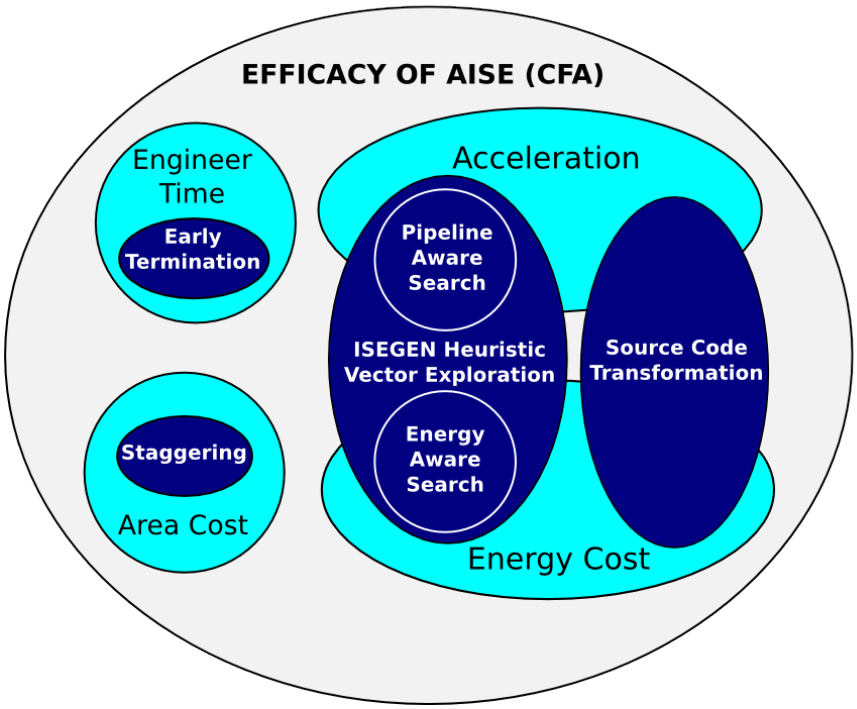


Fig. 1.1: A diagram showing how the different contributions cover the efficacy concerns of AISE. Light blue areas are concerns, and the dark blue areas are the contributions made in this thesis.

1.3 Contributions

The contributions made in this thesis are in several categories spanning the levels of design consideration in a hardware-software co-design.

In order to derive the low level performance of a realistic implementation, a microarchitectural solution to ISE dubbed the "Configurable Flow Accelerator" (CFA) is introduced and used as the basis for later exploration and analysis. The properties of this accelerator are explored in order to demonstrate that it is a viable design for implementing ISEs, and to determine the potential for performance improvement across the axes of design concern. *The CFA is demonstrated repeatedly to be a cost-effective design for ISE implementation.*

Introduced alongside the CFA is a process of temporal partitioning dubbed "CFA Staggering", due to its similarities with loop staggering (software pipelining). *Staggering is demonstrated on average to reduce the area of CFA implementation by 37% for only an 8% reduction in acceleration.*

Following the exploration of cost mitigation within the proposed CFA microarchitecture, this work moves to address the issues of identification, in particular determining whether the heuristics used within the ISEGEN algorithm are in fact efficacious in identifying ISEs which are implemented as CFAs. The first effort in this vein is concerned with the original algorithm, and is divorced from the idea of CFAs. There is a weighting vector located at the heart of

the ISEGEN algorithm which has previously been cited [7] as having a single static optimal value. *A methodology for finding a good static weighting vector for ISEGEN is proposed and demonstrated. Up to 100% of merit is shown to be lost or gained through the choice of vector.* Also demonstrated is that good ISEGEN weighting vectors are application-specific, and that in order to use a single vector across multiple benchmarks a small amount of performance must be lost from some of them. This motivates the use of feature-based dynamic vector selection, although this is not explored in this thesis. A maximally efficacious vector is located for the set of benchmarks and architecture targeted herein, to demonstrate that the vector is different from that given in [7] and to provide a suitably tuned heuristic for later work in this thesis.

The original ISEGEN algorithm is prone to performing a considerable amount of fruitless search, due to being unaware of when an invalid ISE will converge on a valid solution during the algorithm's execution. This thesis presents an early-termination modification which greatly reduces the ISEGEN algorithm's execution time, especially in the presence of larger graphs where the early termination heuristic is particularly efficacious. *ISEGEN early-termination is shown to improve the runtime of the algorithm by up to 7.26x, and 5.82x on average.* It is shown that whilst there are pathological conditions in which the early termination may reduce the quality of a result, these conditions do not occur in any of the benchmarks tested. This modification of the ISEGEN algorithm contributes to the concerns involving the time of the designer or engineer, as it reduces the amount of time taken for an engineer to evaluate a particular instance of an application's source code for suitable ISEs.

The CFA microarchitecture is pipelined with the potential to initiate operations with an issue latency of one cycle. Modelling the performance impact of overlapping independent ISEs in a CFA is investigated as a potential alternative to other I/O pipelining techniques. Instead of giving inputs and reading outputs for a single large ISE over several cycles, each ISE must only use a single cycle to perform input or output, and will generally be smaller for a given I/O constraint. If smaller ISEs can be used, area and energy consumed will be less and the efficacy of the ISE solution will be increased. New heuristics are produced to allow the ISEGEN algorithm to be aware of the additional acceleration obtained through overlapping templates, hence making it able to better exploit the temporal parallelism in the ISEs it produces. *Pipelining is shown to increase acceleration by up to an additional 1.5x.* The new pipeline heuristic is also shown to have better stability with regards to the weighting vector used versus the original combinational heuristic, in addition to producing better results. Two pipeline scheduling heuristics are evaluated within the greater context of the new pipeline merit heuristic, and a "shortest-first" policy is demonstrated to generally produce better schedules in the benchmarks evaluated.

In the earlier work to demonstrate the use of the CFAs in several applications, a near-linear correlation is demonstrated between the power consumed by a CFA and its standard cell area. It is also shown that CFAs can have a beneficial or detrimental effect on the energy perfor-

mance of a particular hardware-software co-design. In order to make the ISEGEN algorithm aware of the potential energy effects, a heuristic is produced using the relationship described in the earlier section 4.2. The new heuristic can be used to determine the "energy merit" of a particular ISE, utilising the integral of power versus time to model the energy cost of using a particular ISE when implemented as a CFA. The new heuristic is placed into the ISEGEN algorithm and evaluated with regards to several benchmarks, and found to produce better solutions across nearly all of them. *The energy-aware heuristic reduces the energy used by a CFA implementation of a set of ISEs by an average of 1.6x, up to 3.6x.* In fact, even though the CFAs are not constrained with regards to area, the new heuristic actually manages to produce a better acceleration for most of the applications than the original combinational heuristic does. The so-called energy heuristic actually makes improvements in acceleration, area, power, *and energy* in most benchmarks. This improvement does not come at the cost of engineering time either, as coincidentally the new heuristic is found to take less time to converge on its solution than the original combinational heuristic. The same process as before is used to demonstrate that a common and stable weighting vector exists for this heuristic, and like the pipeline heuristic this common vector is shown to be more stable than the combinational heuristic.

Compiler transformations are an important source of potential optimisation, and their effects when combined with ISEs are only just beginning to be understood. This thesis presents a transformation-space exploration, in which ISEs are produced after a range of source-code transformations are performed. *The methodology for combined exploration of source transformation and ISE is demonstrated to improve the acceleration of the result by an average of 35% versus ISE alone.* It is also demonstrated that there is a critical link between the efficacy of source transformation and the efficacy of the resulting ISE identification. Transformation and ISE overlap in where they obtain performance improvement from. Wherever transformations are ineffective, ISEs are shown to be generally more effective in accelerating. Source transformation comes with a lower cost than ISE, so this work shows that it is important to perform a zealous evaluation of potential software transformations in tandem with ISE identification.

Low-power embedded processors used for media applications often perform calculations for fractional numbers using fixed-point representations. This thesis performs an evaluation of applications implemented in both fixed- and floating-point arithmetic, with regards to the absolute performance of the solution that would be produced in a design utilising ISE. *It is demonstrated that the fixed point designs provide the best trade-off in execution time, area, power, and energy.* With the higher baseline power and energy performance of the floating point solution, benefits from ISEs in energy consumption are relatively greater for floating point than for fixed point. It is therefore shown that wherever floating point is utilised, ISEs can be used to significantly reduce both the execution time and energy cost of a design.

Contributions have been made across the spectrum of concerns for ISE; the efficacy of all design concerns have been addressed in one form or another, and improvements made in all

cases. Exploration into the ancillary concerns of ISE design such as source transformation has been produced, promoting a holistic approach to ISE wherever possible: In a hardware-software co-design utilising ISE, both sides of the design must be evaluated together in order to produce a better design.

1.4 Document Structure

This thesis is organised as follows:

- Chapter 2 introduces and describes the background material (empirical, practical and theoretical) required to understand the motivation and implementation of the work of this thesis.
- Chapter 3 describes efforts in the field of specialised computer architecture, specifically those relevant or related to the work performed in this thesis.
- Chapter 4 introduces the Configurable Flow Accelerator (CFA) ISE implementation, explores the strengths and weaknesses of the implementation as originally conceived, and introduces a temporal-partitioning algorithm for compressing the latency and resources of a CFA implementation.
- Chapter 5 looks at an existing high performance ISE identification algorithm (ISEGEN), with an eye towards increasing its efficacy for both the "classic" combinational ISE exploration and more specific situations such as energy optimisation and pipelined microarchitecture. In addition to improving the scenario-specific quality of the algorithm's output, the runtime of the algorithm itself is reduced by introducing search early termination.
- Chapter 6 is concerned with the form in which an application's software is delivered to the AISE methodology, in particular with regards to the software transformations and number formats utilised. In the latter case techniques from chapter 5 are applied as appropriate to determine if they are more or less efficacious than the original combinatorial methodology.
- Chapter 7 concludes this thesis with a critical analysis of the work performed therein and suggestions for future continuation of this effort.

1.5 Summary

In this chapter, a general introduction to the contents of this thesis has been covered. Special consideration has been given to design concerns this work is intended to address by improving the efficacy of ISE:

- Engineering Time.
- Acceleration.
- Area Cost.
- Energy Cost.

This chapter has introduced and motivated the process of processor specialisation via ISE, in particular the engineering concerns that will be encountered, so that these may be directly addressed in later chapters. Contributions made in addressing these concerns and the structure of this thesis hereon have been covered, and now we progress to providing the background information necessary to understand this thesis.

2 BACKGROUND

"I was born not knowing and have had only a little time to change that here and there."
– Richard Feynman

This chapter covers a review of the material necessary as background to understand the motivation and implementation of the efforts contained in this thesis.

2.1 Embedded Processors

General purpose processors (GPP) tend towards high-power, deep pipelining, large die-size, and high clock frequency; embedded processors on the other hand tend towards the exact opposite:

- Pipeline depths tend to be limited in order to reduce the cost of a flush; the degree of speculative hardware used in GPP is much greater than embedded cores, which tend towards simpler branch prediction schemes.
- Clock frequencies tend to be lower as a product of the shorter pipelines, as deeper pipelines also tend to burn a lot more static power through driving feedback in latches.
- On-die sizes of embedded processors tend to be smaller, as in general embedded processors are not provided as fabricated products; rather as IP blocks for integration in a system-on-chip design.

Embedded processors can largely be divided into three classes, although there is some overlap:

- Non-extensible, Non-reconfigurable cores. These are a slowly dying breed of cores which tend to favour higher performance serial processing and general purpose execution to some degree. These may include some application- or domain-specific acceleration functions. Examples include the Pentium-M at the high end of performance and cost, or the ARM 7TDMI at the lower end. These cores can generally only be externally accelerated, through the use of co-processors.
- Extensible Cores. These cores may undergo extensions to the architecture and microarchitecture pre-fabrication, in order to accelerate a particular application or domain. These are usually referred to as ASIP.
- Configurable Cores. These cores are defined with computing fabrics at the microarchitectural level which are reconfigurable post-fabrication. The degree of reconfigurability

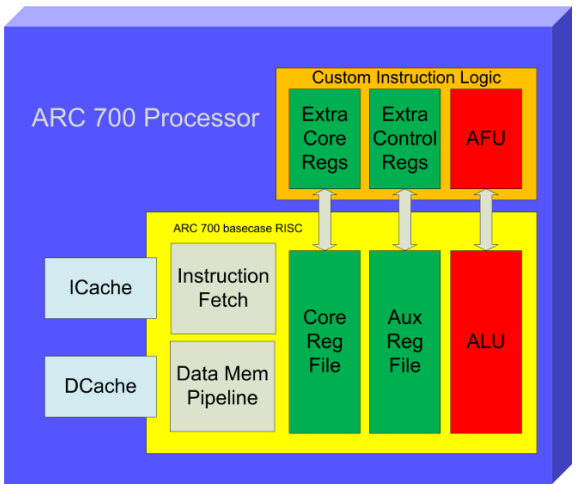


Fig. 2.1: A simplified system-level view of ARC 700 family architecture, demonstrating the pre-verified baseline core and its connection to ISE hardware through custom registers and arithmetic units.

varies from the less mutable microcode specification, to fully dynamically reconfigurable architectures.

The following sections cover the relevant details of the latter two classes, with respect to the problems addressed by the research detailed herein.

2.1.1 Extensible Processors

Extensible processors contain a number of variable components, essentially opening up design spaces inside the processor core for exploration by the designer of an ASIP-based system. Extensions to registers and supporting arithmetic logic are implemented outside of a prefab baseline core, the latter implementing all of the expected basic RISC functionality. In this manner, users may make the best use of the degrees of freedom provided, with the knowledge that their extensions will not make unpredictable timing changes to the core as a whole. Architectures are extended by implementing extensions in SystemC or Verilog with respect to the architecture's extension interface. Some extensible processors have supporting tool-chains which allows for an engineer-guided exploration of the application to determine a partitioning between software execution on the original architecture, and hardware execution through ISEs. Examples of Extensible processors include the ARC 700 (See figure 2.1), Tensilica's XTensa, and Altera's NiosII.

The intention with extensible processors is to create architectural extensions prior to fabrication, which are thereon fixed in their purpose to accelerate specific application components. Whilst this approach appears on the surface to provide the best potential for cheaply minimising the area-delay products and other such metrics, there is a widely recognised problem in the design process. Application sources must be mature or predictable enough to permit an

early specialisation of the architecture through extension. Once the hardware design is fixed, any changes to the application may invalidate previous assumptions about timing and data-flow which contributed to the efficacy of the overall HW/SW combination. The problem of the flexibility of an architecture becomes key, and therefore precludes very complex data-flow acceleration due to a possible mismatch between the identified extensions and an application at a later stage of its development. The Tensilica XTensa approach to this problem is to create a range of complexity in the ISEs generated. When the application drifts from its original specification the less complex ISEs are still able to be mapped for a performance advantage. At this point though, the benefit of the larger ISEs is lost entirely. The static ISEs cannot be reconfigured to match the structure of the new software.

2.1.2 Reconfigurable Processors

Reconfigurable Processors are an attempt to address the problems of application drift during the design process, as they allow for more flexibility in the acceleration microarchitecture at the cost of more control logic. Examples include the Xilinx combined FPGA/PowerPC, the Stretch S5000 and S6000 processors, and the ARM OptimoDE. This list is given in order of granularity; that is the level at which reconfiguration is possible. Without doubt, the most flexible in terms of achievable function space is the FPGA solution, wherein ISEs are implemented in an FPGA surrounding a hard-macro processor core. There are several inefficiencies to the FPGA approach, all stemming from the high degree of complexity and latency inherent in the interconnect of the gate array.

DSP applications will include a great deal of integer (fixed-point) and potentially floating point arithmetic, for which the mapping from an FPGA to the required structure is rather inefficient when compared to the direct gate-implementation of an ALU for that operation. The bitstream required in order to maintain the instance of such a direct implementation of an operator versus the FPGA equivalent will always be far smaller, as all that is required is potentially a mode (operation select) as opposed to a bitstream maintaining the entire structure of the ALU. The next level of architectural flexibility is based off this coarse-grained FPGA idea, providing a regular collection of arithmetic operators with a mesh routing fabric. The approach of combining both high and low flexibility components gives such products a considerable advantage in cost-benefit in their targeted domain versus FPGA alternatives.

Most reconfigurable processors at present require the engineer to manually select and replace sections of source code with ISEs that utilise the various reconfigurable fabrics available to the architecture. The addition of design automation in this process can, if it provides solutions of equal quality or better than an engineer in a manual process in the same time, provide a great improvement in design cost.

2.2 Design Space Exploration

Hardware / Software Co-design [2] is the phrase used to describe the inclusion of both hardware and software elements simultaneously in the design of an embedded system. It was the original idea from which more recent topics such as Design Space Exploration (DSE) [8] were inspired, and fundamentally forms the basis on which this thesis rests. A number of engineering methodologies [2] for the combined system-level design of hardware and software have been published. Such methodologies tend to involve manual analysis of the constraints of a particular problem, and really do not provide much in the way of design automation. Through a structured and methodical design process centered around the preservation of design flexibility and efficiency the designer performs all of the trade-offs themselves. Whilst this approach offered a considerable advantage over classical software engineering for embedded system level integration, it does not attempt to address the dwindling efficiency of the engineer themselves. With ever increasing numbers of design trade-offs available, the engineer is swamped with complexity.

Design-Space Exploration [8] is an attempt to properly formalise the trade-offs inherent in an embedded application design. Constraints such as execution speed, power use, cost, and complexity are made functions of axes of design, which represent the potential dimensions in which a design may develop. There is a space defined by these dimensions, but by no means is this space straightforward to explore. A large degree of trial-and-error is required in the process, placing a premium on making small changes to a design and observing the effects that has on its various performance metrics. In addition, the various levels of hardware and software co-design spread the space across multiple levels of abstraction, further increasing the complexity of attributing performance to particular "features" of the design. What is more, performance functions over the space are often exceptionally discontinuous and non-linear for continuous regions of the space. Whilst original efforts in manual design space exploration have been very promising, generally these efforts were reduced to a mechanised brute-force evaluation of contiguous regions of the design space at-once, with a degree of controlled iteration and movement within the space provided by the engineer atop the design process. It soon became apparent that there were more effective means of automating this process by taking the engineer out of the loop in order to make a wider guided evaluation of the space.

Meeting speed constraints means that no further increase in speed is useful for its own sake. In certain cases excess speed may be "spent" in reducing other design costs. So long as hardware and software together meet the minimum speed requirements, application deadlines will be met and the system built around the core will be able to communicate and process data. No stalls will occur due to system-level deadlines missed by the core, and power will have been reduced overall. For example, if execution speed exceeds requirements, the clock speed of the ASIP may be reduced, reducing power and hence energy consumption. These secondary concerns have additional design spaces of the configurable core available to be explored for

satisfactory areas; for example clock gating [3], dynamic voltage scaling [9], and unit pruning. The "second order" effects of core extension are not always beneficial and often hard to predict with any accuracy. Adding more logic to a core can for example increase the critical path and force a reduction in the overall clock speed. Such a complicated web of non-orthogonal trade-offs forms a space which can only be explored efficiently through the aid of iterative automated means.

2.3 Instruction Set Extension

Instruction Set Extension (ISE) is the process of adding new instructions to a baseline core in order to improve the performance of the core with respect to a particular application or set of applications (domain). The approach requires work in a number of distinct areas: those of architecture (instruction set design), microarchitecture (the hardware implementing the architecture), and compiler (in order to map applications to the new architecture). These three problems have been addressed manually for some time, with engineers using profiling and manual inspection of application code to determine the best application of ISE. RISC architectures have formed the basis of most ISE-driven approaches for a considerable time. Their basic architecture is able to efficiently cover the non-ISE component of the application without expending excessive amounts on resources attacking things like dynamic instruction-level parallelism. Most high-performance execution is performed by ISEs and not by the general purpose component of the design, which makes dynamic OLP-exploiting hardware redundant.

ISE affects all the main axes of design concern (acceleration, area, power, energy, engineering cost). The guiding metric in deriving extensions is often still application execution speed; designers will add ISEs that "cover" the hottest (most frequently executed) sections of their application code. The intention is that by partitioning of the application code into areas covered and not by ISEs, sections of microarchitecture can be dedicated to the servicing of these new ISEs. This is an example of the application of Amdahl's law; by covering the dominating areas of program code with acceleration the useful effect of the optimisation is maximised for a given application. This approach is ostensibly one of design-space exploration, but in a less classical sense since the opportunities for extension are not bounded so much by design space constraints as by the data-flow structures present in the application code and their potential for mapping to specialised microarchitecture, often referred to as an "Application Functional Unit" (AFU). This aside, there are some constraints which can be thought of as defining the design space when considered with the structure of the application code; these are:

- The number of register file ports for input and output; this ultimately defines the bandwidth to and from the ISE hardware, giving the number of words that can be read to and written from an ISE per cycle. This is commonly referred to as the "I/O Constraint".
- That the data-flow covered by an ISE is able to be scheduled; more specifically that

there are no mutual dependencies introduced by new instructions (i.e. an instruction that both reads from and writes to another instruction, ISE or otherwise). This is commonly referred to as the "Convexity Constraint", as it is represented by the convexity of the data-flow graphs commonly used to define ISEs structurally.

- That the number of ISEs defined can actually be encoded in the space available in the instruction word fields. E.g. there exist opcodes or sub-opcodes free for defining as instruction set extensions, and the number of these is statically finite. This is somewhat relaxed where dynamic hardware reconfiguration is possible, as opcodes can be re-used during execution.

- That the power and energy consumed by ISEs does not excessively overload or drain the supply available to the core.

- That the resources employed by ISEs, in cells or silicon area, does not exceed the maximum allowable by the particular technology and budget in question.

These constraints are derived from different levels in the design hierarchy; convexity being a requirement of the compiler, encoding space being a requirement of the architecture, I/O being a requirement of the microarchitecture. Some constraints are set purely by the cost of producing a given design, and a further meta-constraint is that the time cost of deriving the instructions does not exceed whatever limit is set by the desired time-to-market and human resources budget. Given that the finally selected architecture (instruction set) impacts all elements of the hardware-software co-design, care must be taken to fix each constraint only as it is found to be either necessary or beneficial to the outcome. This represents a "Phase Ordering" problem in the design flow: What constraints should be fixed and what should remain mutable at what stage in the design process. The proper division of the design space into (preferably orthogonal) concerns is essential for determining an answer to this problem.

2.3.1 Abstract Problem Definition

For the sake of both manual and automated analysis, it is necessary to define an abstract formalism of the data-flow which will be examined for ISE candidates. Since the aim of ISE is to take a section of code and convert it to a structural representation, a structural graph representation is a good formalism for this purpose. Application code and ISEs are themselves represented as a Data-Flow Graph (DFG). Application code DFGs are derived directly from basic blocks from within the compiler representation of the application. DFGs are defined as $G = (V, E)$, wherein V is the set of vertices representing operations and E is the set of data-flow edges connecting the operations in G. Vertices of DFG are referred to as "nodes", which may or may not be coverable by the particular ISE methodology in question. For example, throughout this thesis memory operations are not coverable, and so will never be included in an ISE. Other nodes

are not represented in the DFG, specifically those involved in control-flow. For this reason, all performance figures given in this thesis with regards to this model are concerned only with the data-flow portion of an application, not the control-flow.

When calculating the impact of an ISE formalised in this fashion, the difference between the sum of the latencies of all the nodes in an ISE minus the sum of the latencies of all the nodes in the critical path of the ISE constitutes the speedup in cycles. The software and hardware latencies of a particular node reflect the number of cycles the node will take to execute on the baseline RISC processor and as a synthesised functional unit in an ISE, respectively. The hardware latency is generally lower than the software latency in standard-cell approaches, but in FPGA the extension logic can sometimes be slower than the baseline processor. The speedup of implementing a particular DFG G as an ISE is calculated as follows:

$$\lambda\_sw(G) = \text{Sum of all software latencies of nodes} \in G.V$$
$$\lambda\_hw(G) = \text{Sum of all hardware latencies of nodes} \in \text{critical path of } G.V$$
$$speedup\_cycles(G) = \lambda_{sw}(G) - \lambda_{hw}(G)$$

The total software cycles taken by an application in the data-flow domain is calculated in a similar manner. First the application is profiled to get per-basic-block execution frequencies. For each basic block, the number of software cycles is calculated by summing the software latencies of all the nodes in the DFG for that basic block. This value is multiplied by the profiled execution-count for that basic block and is added to the total. Taking the same approach but including the speedups calculated as above, the ISE-accelerated cycle count can be calculated. This is the method used in all experiments in this thesis. In all experiments in this thesis other than that of section 6.2 the node latencies are set to those of the EnCore processor [10]: both hardware and software latencies are the same, due to the baseline and extension being implemented in the same standard-cell technology.

2.3.2 The Software Emulation Fallacy

When performing ISE for any given baseline core, we are weighing up the cost of the new extensions (in area or power) versus the benefit (in acceleration or power). It is very important to keep in mind what you are using for a baseline, and the continuum of design from that baseline to any other design point currently under consideration.

Wherever a baseline has simple RISC-like operations which are not covered by hardware functional units, software emulation is usually used to provide these operations to compilers. An example of this is the GNU libfpe (Floating Point Emulation Library) provided as a part of the GCC compiler infrastructure. The libfpe software provides floating point operations in terms of a series of integer operations. Most architectures will then be able to use to provide floating point calculation in the absence of floating point functional units. When the cycle-counts for software-emulated operations are included as the software latencies of nodes in ISE,

the resulting speedup when considering a single ISE design point is grossly misrepresented. In order for ISE identification algorithms (or indeed manual ISE) to honestly represent the merit of ISE, the baseline for extension should always consider the software latency of each operation represented as being the integer ceiling of an individual hardware functional unit that could be included. ISE as represented in the literature (see section 3.2) does not generally consider the simple instructions which could be added, rather opting to cover as large an array of complex arithmetic as possible in order to obtain a design point of high merit.

It is important when designing an ASIP to perform some degree of design-space exploration outside ISE, both before and after the ISE is performed and often with a degree of iteration. Citing the speedup obtained by complex ISEs when software-emulated operations are included in the software latency calculations will invariably provide an extremely high speedup where these operations are included in ISEs. Including a scalar hardware functional unit to cover the software emulated functions would provide a large degree of this speedup. Even the design-space of a single scalar functional unit is complex and results in a number of potential solutions; a simple integer multiplier itself has several potential structures for implementation at different cost/benefit points. For this reason and others, ISE alone is not a holistic solution to designing ASIP, but instead one powerful technique within a number of other techniques which overlap with regards to their impact on the function of cost versus benefit. In this work, software latencies are always assumed to be as if there were a scalar hardware functional unit; this avoids misrepresenting the speedups by including the benefit from simple scalar extensions before addressing the complex approach of ISE.

2.3.3 Separation of Concerns

As discussed earlier, ISE design is a DSE problem at heart, and one with various separable concerns; acceleration, power, energy, area, code size, and engineering time. It has been demonstrated that the orthogonalisation of concerns [11] is necessary for a thorough exploration of the potential design points.

With regards to acceleration, the following properties of an ISE design will contribute to its efficacy:

- **Operation-level/Spatial parallelism.** Parallel instances of arithmetic hardware are used in order to perform multiple operations at-once, as allowed by dependencies.
- **Aggregation of clock period surplus present in most arithmetic functions.** In particular, bitwise functions have a hardware latency far below the clock period in most cases.
- **Issue latency between data-independent ISEs.** Scheduling ISEs to have the minimum

possible distance between independent instances will allow these to be temporally parallel.

- **Reduced register-transfer overhead, due to the increased locality of communication within the functional unit used to represent the ISEs.** Wherever a value is passed between nodes in an ISE, register pressure is reduced. The opposite may occur with wider ISEs, which may actually increase the pressure through requiring a large number of live registers for input and output.

Power and energy performance will depend heavily on the microarchitectural implementation selected, and the size of ISEs used. Some work (see section 4.2.1 for a review) has been done by others to characterise the power and energy concerns of ISEs. The following properties of an ISE design point will effect these concerns:

- **Number and complexity of arithmetic units used in ISE microarchitecture.** Dominant in non-sub-micron designs, power will depend on the number of concurrently processing arithmetic units. This is naturally at odds with the spatial parallelism described under acceleration above, and adding more arithmetic units will always increase power. Power though, is only one factor of energy. Addition of more arithmetic units may decrease energy through acceleration. This trade-off is investigated in later sections 4.2, 5.5, and 6.3.

- **Number and width of flip-flops used in ISE microarchitecture.** Dynamic and static power are the dominant factors in processor power consumption, and the registers used in making a synchronous circuit are the source of these factors. Dynamic power is the major contributor in designs above 90nm, with static power becoming dominant below this technology node. Deeper pipelines are a major contributor to the power and energy consumed by a design.

- **Clock Gating.** The granularity of clock gating is of particular importance to ISE, and will have a large impact on the efficacy of maintaining both low power and energy consumption when using ISE.

- **Power Gating.** whilst harder to apply than clock gating and not yet investigated at all in the context of ISE, power gating should prove useful in the future for shutting down ISE when not in use.

Code size is generally impacted positively by ISE where the baseline architecture is RISC with a single instruction word size (by far the most common case). Each instance of an extension instruction should cover at least two operations, so wherever an ISE is used it will bring down the overall number of instructions directly. The decrease in register pressure covered under acceleration above will also contribute a reduction in code-size where spill code is removed.

Engineering time is not really a function of any point in the design space, but is still a relevant design concern for any practical application of ISE. The use of ISE in a purely manual design methodology is fairly impractical for all but the simplest efforts, due to the effort involved. Both time-to-market deadlines and human resource limitations will govern the amount of time that can be spent designing any application of embedded computing. Design automation is absolutely critical in order to make ISE an industrially viable technique; the production of extended architecture, microarchitecture, and compiler by automated means is a major goal of ISE research. Automating as much of the design process as possible frees up human resources for more creative tasks, decreases the time-to-market, and delivers a higher quality through reducing error from the human element. SoC are becoming more and more complex, with the human designer being placed higher and higher atop a pyramid of abstractions allowing them to govern the design through high level specifications and constraints. The transition from manual to automated design echoes similar developments elsewhere in computing and manufacturing across history, and is a sign of increasing technological maturity.

2.3.4 Amdahl Limit

When considering the impact of a particular ISE technique with respect to the application code that is being examined, it is useful to have a measure of the maximum possible acceleration possible. Amdahl's law provides an effective tool using which we can address this problem directly: By subtracting the runtime of coverable nodes from the total runtime and considering the original runtime divided by the reduced runtime. This ratio is then the acceleration that would be obtained if all the area coverable by ISE were reduced to zero latency; effectively infinite acceleration for the hardware element. Whilst this is definitely an idealistic performance measure, it does represent an asymptote which is not possible to surpass without addressing the problem of making more of the application coverable by the ISE methodology in question.

S_{all}: Number of software cycles consituting the entire DFG. S_{cov}: Number of software cycles constituting DFG vertices coverable by ISE. $coverable(V)$: Sub-set of all dfg vertices (nodes) in V coverable by ISE. V: Vertices of DFG. $Accel_{limit}$: Amdahl limit for ISE acceleration for the given DFG represented by V. $S_{all} = \sum_{v \in V} \lambda_{sw}(n)$

$$S_{cov} = \sum_{v \in coverable(V)} \lambda_{sw}(v)$$

$$Accel_{limit} = S_{all}/(S_{all} - S_{cov})$$

2.3.5 Micro-architecture

When considering extension logic rather than a full-custom solution, there must be an interface between the extension logic and the baseline logic. In this case we consider only the extension of a RISC pipeline, which can be achieved in multiple ways:

- Coprocessor; extension logic is implemented as a coprocessor with a separate register file and control flow. A coprocessor is suited to extensions covering large contiguous blocks of code, such as whole functions. Coarsest granularity of extension and highest overhead in accessing extensions. This might not be considered as ISE so much as just "extension", but coprocessor operations are often referenced by special instructions on the master core.

- Loose coupling; extension logic is implemented in a block of logic lacking control-flow via memory mapped I/O control and DMA. Suited to extensions covering large areas of memory with similar or identical data-flow operations. Medium granularity of extension and medium overhead in accessing extensions. Extensions may be accessed via ISE or reading and writing to memory-mapped extension logic.

- Tight coupling; extension logic is integrated directly into the pipeline of the host core, via the register file and extensible sections of the decode stage in the pipeline. Finest granularity of extension and lowest overhead in accessing extensions. Extensions are always ISE in this context, operated using special instructions added to the baseline ISA.

In this research, we only consider the third type of integration: extension logic tightly coupled with the baseline core's pipeline. Keeping the logic inside the host core allows smaller sections of code to be usefully executed in extension logic due to the lack of overhead in accessing the extension logic. The following descriptions cover the kinds of extension logic which may be used in such a system with tightly coupled extensions. These can also be used with the coarser-grained interfaces, but are particularly suited to pipeline integration due to their lack of control flow and relative simplicity compared to higher-function extension.

Pure Combinational ISE

Pure combinational implementation implies that there is no clocking of the internal logic of the ISE, with this instead being governed external to the extension logic with the baseline core asserting the values on the inputs of the ISE, waiting a number of cycles, then registering the output for write-back to the register file. The ISE itself is just a collection of combinational functional units connected by wires in an isomorphic fashion to the DFG it is intended to implement. Resource sharing may be achieved by the insertion of multiplexers on the wires between functional units, to change the shape of the graph being implemented by routing data between functional units on a per-opcode basis.

Algorithms to automatically derive fine-grain ISEs from application code have used the single-ISE pure combinational implementation as a model for estimating the speedup obtained from a given candidate ISE. This estimation faces some error in the face of both wire delays and multiplexing delays due to routing and resource sharing, but this is generally considered to be minimal when compared to the dominant latency of the functional units themselves.

Purely combinational ISEs are the simplest of all ISE implementations, drawing virtually no static power as registers are not employed in their implementation. The lack of registers in purely combinational ISE does cause limitations with regards to their potential throughput, as the data initiation interval (DII) of such an ISE is equal to the number of cycles (or integer ceiling thereof where this is non-integer) that the ISE takes; generally the integer ceiling of the critical path latency. Whilst the area is not bloated with the addition of registers, the lack of such can lead to an implementation which does not achieve an efficacious trade-off between the acceleration afforded and the cost (in area, power, and energy) of implementing the ISE. More sophisticated approaches can yield better results, as is now discussed.

Pipelined ISE

The addition of pipeline registers to combinational ISEs allows for designs with issue latencies lower than the critical path of the entire ISE. This can be especially useful:

- When the ISE is to be executed in the body of a potentially parallel loop, the DII of the loop body may be decreased by up to a factor of the number of pipeline stages.
- When two or more data-independent ISEs are to be executed with an issue latency that is less than the critical path latency of the first ISE, their execution may be overlapped in the same hardware module. For this reason, resource sharing should often be combined with pipelining, so that structural hazards are not introduced.

The addition of registers will add to the power requirements of the circuit. This is a space which can be explored and traded off as required by a designer.

Configurable Compute Accelerators

Configurable compute accelerators [12] are a particular realisation of the idea of coarse-grain reconfigurability, specifically targeted at obtaining the greatest speedup for the minimum die area and with the maximum effective flexibility. CCA are effectively a microarchitecture for implementing reconfigurable ISE. The CCA itself is much like the microarchitecture one would expect to implement a single ISE, except it may have any of its potential combinations of subgraph and node operation induced as an ISE. This function of dynamic reconfiguration is achieved through the use of a LUT connected to the CCA, which for a given CCA opcode and arguments looks up the correct configuration of the CCA and induces it. The majority of the existing work [12, 13, 14, 15, 16] on CCA has concentrated on the use of CCA with transparent dynamic translation hardware. The authors of the work state that this is because it is possible to implement hardware to recognise sequences of instructions which may be dynamically translated into a configuration line in the CCA LUT, and used from that point onwards to accelerate the application without translation overhead.

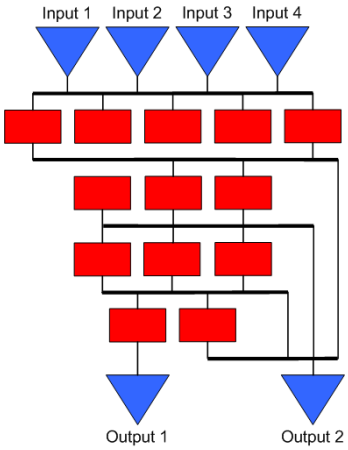


Fig. 2.2: Domain-specific CCA derived for Audio-based Benchmarks, in [13].

Work has been done to establish an initial foray into the DSE of CCA configurations [15], but this was neither guided by heuristics nor automated. Exploration was also only performed over a small number of applications, application domains, and DSE parameters. It was determined that even for this small manual exploration the effort expended was well worth the results: CCA performance for application specific accelerators was far higher than CCA designed with pure generality in mind. Domain specific acceleration with CCA is also well motivated by this study, as it is shown that for a relatively small overhead over the application-specific CCA, all applications in a benchmark domain may be accelerated to similar levels by a single domain-specific CCA. See figure 2.2 for an example of a domain specific CCA, as produced for the audio domain in [13].

Work to determine the usefulness of exploring the additional axis of operator bit-width [16] has shown the CCA to be particularly efficient in the case where there is low-bit-width arithmetic and exploitable OLP. Low-width operators may be combined through relevant carry propagation into larger widths to deal with both thin and wide data in the same reconfigurable unit. This approach is adopted by the previously mentioned Stretch S6000 [17], the ISEF of which is effectively a massively over-specified non-transparent CCA.

ISE Memories

As mentioned earlier one of the desirable properties of ISE is the reduction of register pressure that can be obtained through encompassing large numbers of operations that would otherwise be passed by register with single ISEs. There are two further instances that through using single-cycle memories with ISE, pressure on the register file can be reduced:

- Where values are to be passed between ISEs (and are not otherwise required to be used by baseline RISC operations), then it is possible to use a local scratch-pad in the ISE microarchitecture to store the values rather than passing them via the register file. This

reduces both the amount of pressure on the main register file and the number of register file I/O ports required to implement a given ISE.

- Where the same value is used by successive ISEs (such as a constant), it may be stored after it is transferred to the ISE the first time such that successive instances do not require it to be transferred again.

In the face of an interrupt driven environment such as might be expected in a modern embedded OS like Linux, consideration must be given to what happens to the internal state of ISE when a context switch or interrupt is encountered by the extended processor. This is because the interrupt may change the state of the ISE memory, causing incorrect results to be produced. For this reason interrupt code should either be prevented from modifying the state of ISEs, or the state should be included in process control blocks and maintained for any context switch. ISE memories are not explored here, in favour of greater depth of analysis in other areas.

2.3.6 ISE Example

By way of an example, consider the following C code:

```
#define SIZE 4

int a[SIZE][SIZE] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12}, {13,14,15,16} };
int b[SIZE][SIZE] = { {16,15,14,13}, {12,11,10,9}, {8,7,6,5}, {4,3,2,1} };
int c[SIZE][SIZE];

matmul(a,b,c)
  int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];
{
    int i,j;
    for(i = 0; i<SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
        {
            c[i][j] =
                (a[i][0]  * b[0][j]) +
                (a[i][1]  * b[1][j]) +
                (a[i][2]  * b[2][j]) +
                (a[i][3]  * b[3][j]);
        }
    }

}

int main(int argc, char **argv)
{
    matmul(a,b,c);
    printf( "c: %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d\n",
            c[0][0], c[0][1], c[0][2], c[0][3], c[1][0], c[1][1], c[1][2], c[1][3],
            c[2][0], c[2][1], c[2][2], c[2][3], c[3][0], c[3][1], c[3][2], c[3][3]
          );
    return 0;
}
```

After converting the application to a DFG, the main kernel of this small program is the basic block at the innermost part of the loop (executed 16 times). Figure 2.3 illustrates the DFG form of this inner loop basic block.

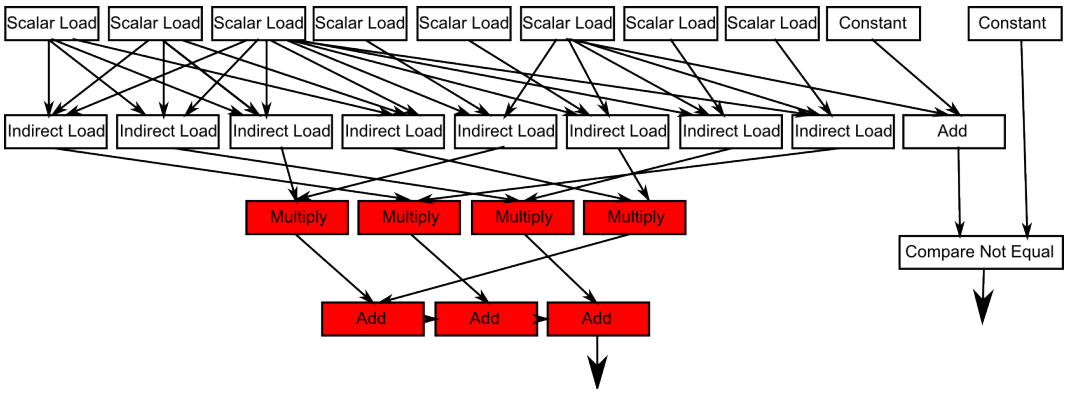


Fig. 2.3: DFG of Matrix Multiplication Inner Loop Body, with one possible (8-in, 1-out) ISE candidate hi-lighted. The ISE would take 15 cycles in software, or 6 cycles in hardware. Arithmetic transformation of the ISE could reduce this to 5 cycles; GCC has not balanced the adds into a binary tree.

Even with this simple example, we can make observations on the various facets of ISE design. With hardware and software latencies for multiply and add considered as three and one cycle respectively, the major contribution of the ISE will be acceleration through operator level parallelism. There is no room in the latencies for serial overhead to accumulate between the two operators in the critical path. The critical path latency for this instruction is six cycles, running between the leftmost multiply and across the bottom row of additions. This could be reduced to five if some arithmetic transformation had occurred, since the three additions are in serial. The software execution speed of this ISE is fifteen cycles, meaning that for the area covered the ISE is roughly a factor of three faster than its original software representation. Note that the same reduction in cycles would have been achieved without including the addition operations in their current state, due to their serial arrangement. This is therefore also a simple counterexample to the assumption that "bigger is better", made in some recent ISE work [18]. The smaller 4-multiply ISE would use less area, and less power, whilst providing the same acceleration as the maximal clique [18] represented by the ISE in figure 2.3. Assuming the minimal 4-multiply ISE, the DFG is reduced from 33 to 24 cycles; a speedup of 1.375x.

The Amdahl limit for this example is the sum of all the software latencies (33 cycles), divided by the same sum minus the number of cycles coverable by ISE (16 cycles); giving a factor of 2.06x. This means that for the given DFG, and allowing only arithmetic nodes to be covered by ISE, the acceleration can never rise above 2.06x. Software transformations can help to increase this factor, and these are discussed later. It is difficult to discuss the area, power, and energy concerns of this example in any detail due to not yet introducing a microarchitecture. This will be covered in chapter 4. The identification of an ISE in such a simple example is a relatively trivial task for both manual and automated means. The unconstrained design space of this example with only 9 nodes to cover is still 2^9 or 512 design points, although not all of these points will satisfy constraints. This should serve as an illustration of how complex

the identification problem becomes when there are 100 or more coverable nodes, as in many application kernels.

2.4 Automated Synthesis

The task of taking a high-level specification of a system and lowering it to a design which may be fabricated has historically been the remit of engineers and their tools. As time goes on, the ability of the tools to perform work which would otherwise have been performed by the engineers has progressed further and further, to the point now where few of the low-level details are actually controlled directly by the engineers. Instead of lowering the design directly, the high level specification is taken by engineers and converted to a machine-readable format (such as Verilog). Automated synthesis tools then take the machine-readable design and lower it depending on the constraints and libraries provided by the engineers. Several different domains of automated synthesis are available to engineers when producing an ASIP, and these are described during this section.

2.4.1 Automated Instruction Set Extension

The automation of ISE exploration has been actively studied in recent years, leading to a number of algorithms e.g. [19, 7, 20, 21] which derive the partitioning of hardware and software under microarchitectural constraints. Work is still being performed in defining the full space of exploration even in purely arithmetic ISA design [22]. Work to include better models in tools has allowed for better decisions about the performance and feasibility of extensions [23], but further work is required.

Most current approaches to automated ISE incorporate two phases:

1. Identification; whereby formalisations of basic blocks are analysed to produce ISEs in the form of DFG. Current algorithms use a combination of register file ports, bandwidth to other on-chip-memories, number and size of operations to constrain this problem, effectively a search technique. In order to guide the search, an abstraction of the speed of such operators in hardware is used to deduce the delay of the operation in software (the sum of all its nodes) and in hardware (the sum of the nodes in the critical path).

2. Selection; whereby the identified ISEs of the previous phase are organised in terms of their performance, via Pareto-optimal ordering [24], greedy selection of the as many best performing templates as will fit in the coding space, or a knapsack formulation based on cost-benefit.

Algorithms differ in the result quality and runtime generally by trading off greed for speed. Some approaches like the Tensilica XPRES [25] compiler will identify a massive number of non-orthogonal ISEs in order to provide a large degree of freedom in the selection. On the other

hand, algorithms like the Atasu [20] and ISEGEN [7] formulations focus on modelling some aspect of the microarchitecture (most often delay), and then greedily identify the largest and best performing template for each basic block. This process is then repeated for the remainder of the basic block. This is an example of the trade-off between greed and speed, as it provides space reduction in both the identification and selection phases. In addition, selection phases have the benefit of orthogonal templates; the performance may simply be summed over the set selected.

This approach is somewhat flawed in the face of wire-delays, and in larger templates the algorithms will become less and less accurate at predicting the actual delay of an ISE. This flaw is as a product of the automated ISE algorithms currently blindly targeting ASIC implementation of the resulting extensions, a process which is wildly affected by a number of design concerns outwith the control or speculation of current automated ISE algorithms. The mapping of ISEs to an existing CFA would permit far more accurate analysis of the delays inherent in mapping a particular extension. This is because the hardware is already realised and may be modelled in terms of its actual, empirically determined performance.

The exploration approach of using a range of tools, iteratively operating on a canonical system-level ADL is described as "Compiler-in-Loop Design-Space Exploration" [26]. It was originally motivated [27] through the discovery that iterative and methodical exploration of ASIP design is beneficial in decreasing time-to-market. CoSy [28] and LISATek [29] tools feature in many such frameworks; figure 2.4 illustrates such a combination.

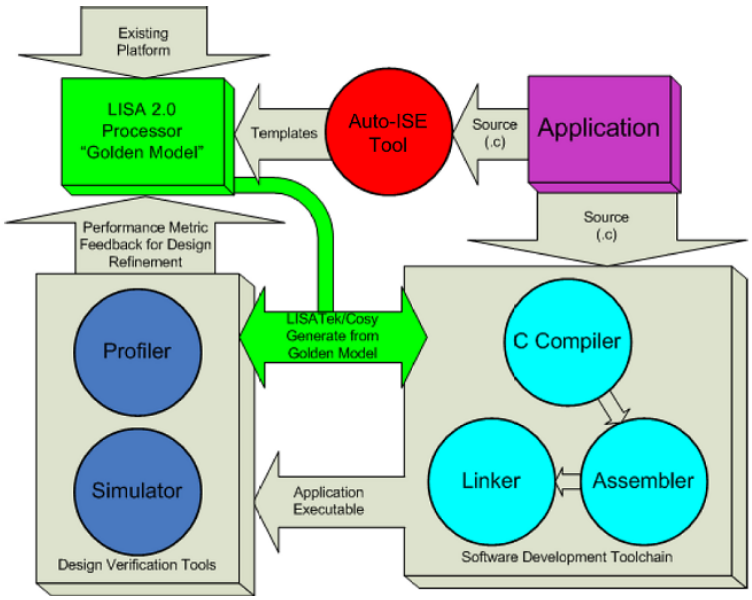


Fig. 2.4: The Compiler-in-loop methodology for ASIP design space exploration.

General formulation as DFG-subgraph-inducement or DFG Partitioning algorithm. Discussion of semi-greedy approach.

2.4.2 ISEGEN Algorithm

The ISEGEN AISE Algorithm, first presented by Biswas *et al.*in [7, 30], is an adaptation of the Kernighan-Lin circuit partitioning algorithm. The goal of the algorithm is to partition a given DFG into hardware and software cuts. The algorithm only addresses the identification phase of the process, assuming selection will be performed after ISEGEN is run over the entire application. ISEGEN includes a compound heuristic which includes the standard ISE merit model (see section 2.3.1), the expected constraints (convexity, I/O), and additional metrics to represent what the authors describe as the "designer's objective":

- Large Cut; solutions are weighted towards larger cuts, with the algorithm preferring growth into areas where growth potential is higher. This corresponds to growth in a vertical direction.
- Independent Cuts; solutions are weighted towards cuts containing multiple disconnected components. This corresponds to growth in a horizontal direction.

This compound heuristic is weighted by a vector to select a particular linear multiple for each component. The heuristic is used to provide a merit to every node which could be toggled in a given iteration of the algorithm. The algorithm allows intermediate iterations in search to violate constraints, in the hope that the algorithm will eventually settle upon a valid solution. ISEGEN is particularly suitable for extension and use in the work of this thesis, because it is flexible in the heuristics used to both explore the design space and ultimately determine the ISEs for selection. Heuristics may be extended arbitrarily with procedural code, but we must be careful to keep the runtime of the heuristic low as it sits at the very core of the ISEGEN kernel.

Put simply, the algorithm is comprised of three nested loops. The outermost loop sets the working cut to the previous best cut (or the original DFG), performs the middle loop, and then checks to see if the result has improved. If it has not, the loop is ended, else it continues up to a pre-defined number of iterations to attempt to improve the result further. The middle loop executes so long as there are unmarked nodes. A marked node is a node which has been toggled from software (standard instruction set) to hardware (part of an ISE), or vice-versa, for the current execution of the middle loop. Each iteration of the middle loop first calls the innermost loop to determine the merit of toggling every unmarked node. The node with the highest toggling merit is toggled and marked. The best cut is updated to equal the current cut if the latter meets design constraints and is of improved or equal merit to the former. The process continues until the outermost loop detects no improvement in the best cut.

Algorithm Definition

The following pseudo-code is transformed from the description in [7] in order to make it both easier to understand and to match the implementation undertaken for this work. The original description has ambiguities which make the algorithm somewhat recondite on first inspection, and the implementation made for this work has a small difference in order to improve the efficacy of the search.

The algorithm is comprised of two merit functions ($M(C)$ and $M_{toggle}(n,C)$), two sub-functions ($SetInitialConditions()$ and $CalcImpactOfToggle(n,C)$), and a main function ($ISEGEN(C,DFG)$).

ISEGEN is comprised of three loops nested within one-another. The very innermost loop iterates through all unmarked nodes (a node being marked once it is selected as the best node from said loop) and determines the toggle merit for each using the compound heuristic discussed earlier. The node with the highest toggle merit ($M_{toggle}(n,C)$) with respect to *working_C* is toggled from hardware to software or vice-versa depending on its original state. The next loop out repeats this process of selecting and marking nodes until all nodes are marked. Nodes are toggled in the working cut and the best cut obtained in the current outer-loop iteration is updated, if the working cut is valid per the constraints specified. The very outermost loop repeats the mark-and-toggle approach up to five times, unmarking the nodes and storing the best cut obtained from the inner loops when a better cut is observed as per the cut merit function ($M(C)$). Three cuts are therefore generally active at any time in the algorithm: The innermost loop uses *working_C* which may violate constraints but performs the vast majority of the search guided by $M_{toggle}(n,C)$, the middle loop uses *best_C* to store the best valid cut obtained from the innermost loop, and the outermost loop uses *last_best_C* to store the best valid cut obtained. The algorithm can be seen as a process of promotion, with cuts flowing from the innermost loop to the outermost if they first meet constraints and second are the best cut observed. Allowing *working_C* to violate constraints allows the algorithm to escape local maxima, being drawn away by the heuristics of $M_{toggle}(n,C)$ to explore further areas than would be approached if the entire search was guided purely by $M(C)$.

The merit function governing the promotion of cuts between iteration levels is the cut merit function, $M(C)$. It is defined as follows:

$$\lambda\_sw(C) = \text{Sum of all software latencies of nodes} \in C$$
$$\lambda\_hw(C) = \text{Sum of all hardware latencies of nodes} \in \text{critical path of } C$$
$$M(C) = \lambda_{sw}(C) - \lambda_{hw}(C)$$

The weighted compound heuristic $M_{toggle}(n,C)$ is used to govern the order of selection of nodes for toggling from hardware to software or vice-versa. It is comprised of five sub-heuristics, each contributing a merit which when combined with the weightings gives a single scalar weighting to each unmarked node in the DFG under analysis. The toggle heuristic is

comprised as follows:

$$savedcycles(C) = \begin{cases} M(C) & \text{if C is convex} \\ -\infty & \text{if C is not convex} \end{cases}$$

$$io(C) = (C.I_{ise} - N_{in}) + (C.O_{ise} - N_{out})$$

$$convexity(n,C) = \begin{cases} +num\_neighbours\_in\_cut(n,C) & \text{if } n \notin C \\ -num\_neighbours\_in\_cut(n,C) & \text{if } n \in C \end{cases}$$

$db\_down(n) =$ minimum distance down of n from barrier nodes.
$db\_up(n) =$ minimum distance up of n from barrier nodes.

$$largecut(n,C) = \begin{cases} +|db\_up(n) - db\_down(n)| & \text{if } n \notin C \\ -|db\_up(n) - db\_down(n)| & \text{if } n \in C \end{cases}$$

$CS(n,DFG) =$set of connected subgraphs in DFG excluding that containing n

$$fragcut(n,C) = \begin{cases} max_{cs \in CS(n,DFG)} critical\_path\_latency(cs) & \text{if } n \in C \\ 0 & \text{if } n \notin C \end{cases}$$

$$M_{toggle}(n,C) = (\alpha_1 \cdot savedcycles(C)) - (\alpha_2 \cdot io(C))$$
$$+(\alpha_3 \cdot convexity(n,C)) + (\alpha_4 \cdot largecut(n,C)) + (\alpha_5 \cdot fragcut(n,C))$$

The original algorithm does not have the additional merit check at line 13, which led to that version requiring more iterations in order to settle upon a solution. Testing of this additional check confirms that in all cases this slightly modified algorithm performs faster than the original. The original algorithm description is ambiguous with regards to the toggling of nodes versus the addition and removal of nodes from *best_C* and *working_C*. The cut *working_C* does not appear in the original algorithm description, instead being implied through ambiguous references to "Toggling" a node outwith any cut. The explicit exit from the main loop at line 20 is also not covered in the original description, but is implied through unmarking the nodes only if the conditional statement at line 16 is true. The algorithm would originally have spun through the outermost loop without effect if said conditional ever evaluated to false. NUM_ITERATIONS for the original algorithm was stated in [7] to be five, but generally the algorithm exits before this.

The ISEGEN algorithm is quite a departure from the original Kernighan-Lin min-cut algorithm; the original KL algorithm toggled vertex pairs, with a graph which is initially divided into two same-sized partitions. The result of the original KL algorithm is two partitions of equal

$SetInitialConditions(C, DFG)$
00: $C.O_{ise} \Leftarrow 0$
01: $C.I_{ise} \Leftarrow 0$
02: foreach(node $n \in DFG$)
03: $\quad n.I_{toggle} \Leftarrow Inputs(n)$
04: $\quad n.O_{toggle} \Leftarrow Outputs(n)$
05: endfor

$CalcImpactOfToggle(n, C)$
00: $C.O_{ise} \Leftarrow C.O_{ise} + n.O_{toggle}$
01: $C.I_{ise} \Leftarrow C.I_{ise} + n.I_{toggle}$
02: $\quad n.O_{toggle} \Leftarrow n.O_{toggle}$
03: $\quad n.I_{toggle} \Leftarrow n.I_{toggle}$
04: foreach(node $m \in Parents(n) \cup Siblings(n) \cup Children(n)$)
05: $\quad$ Update $m.O_{toggle}$ and $m.I_{toggle}$ as per rules of [7].
06: endfor
07: Update data-structures for $O(1)$ evaluation of $M(C)$ and $convexity(n, C)$

$toggle(n, C)$
00: if($n \in C$)
01: $\quad$ remove n from C
02: else
03: $\quad$ add n to C
04: endif

$mark(n)$
00: $n.marked \Leftarrow true$

$unmark\_all(DFG)$
00: foreach(unmarked node $n \in DFG$)
01: $\quad n.marked \Leftarrow false$
02: endfor

Algorithm 1 The ISEGEN algorithm main function.

$ISEGEN(C, DFG)$
00: $SetInitialConditions()$
01: $last\_best\_C \Leftarrow C$
02: for(i=0, i $<$ NUM_ITERATIONS, i++)
03: working_C $\Leftarrow$ last_best_C
04: best_C $\Leftarrow$ last_best_C
05: while($\exists$ unmarked nodes $\in DFG$)
06: foreach(unmarked node $n \in DFG$)
07: Calculate $M_{toggle}(n, working\_C)$
08: endfor
09: $best\_node \Leftarrow$ Node with Maximum M_{toggle}
10: $toggle(best\_node, working\_C)$
11: $mark(best\_node)$
12: $CalcImpactOfToggle(best\_node, working\_C)$
13: if(working_C satisfies constraints AND M(working_C) >= M(best_C))
14: best_C $\Leftarrow$ working_C
15: endif
16: endwhile
17: if(M(best_C) $>$ M(last_best_C))
18: last_best_C $\Leftarrow$ best_C
19: unmark_all(DFG)
20: else
21: i $\Leftarrow$ NUM_ITERATIONS
22: endif
23: endfor
24: C $\Leftarrow$ last_best_C

size with a minimised edge weight connecting them. ISEGEN search (and the derivative used here) does not result in partitions of equal size. The major similarity between the K-L min-cut algorithm and ISEGEN is the iterative improvement of the solution based on heuristics, starting from a trivially determined base case.

Computational Complexity

The work published on ISEGEN claims that the algorithm has a worst case complexity of $O(|V|.|E|)$, the reasoning being as follows:

1. The Merit function $M_{toggle}(n,C)$ may be calculated in $O(p)$ time, where p is the maximum number of neighbours (parents, children, and siblings) of a node. All of the $M_{toggle}(n,C)$ components may be calculated in constant time with exception of the $M(C)$ and $convexity(n,C)$, which take $O(p)$ time.

2. The loop at lines 6:8 of $ISEGEN(...)$ calculates $M_{toggle}(n,C)$ for up to all the nodes in the graph, e.g. $|V|$. The complexity of this loop is therefore $O(p.|V|)$. Since $O(p) \in O(|V|)$, $O(p.|V|) \in O(|V|^2) \in O(|E|)$ (note this is the original reasoning; see later comments).

3. The complexity of $CalcImpactOfToggle(n,C)$ called at line 12 of $ISEGEN(...)$ is $O(p+|E|) \in O(|E|)$ due to the updating of data structures at line 7 of $CalcImpactOfToggle(n,C)$.

4. The loop at line 5 of $ISEGEN(...)$ is iterated a maximum of $|V|$ times, with the body being $O(|E|)$ complexity; therefore this loop has complexity of $O(|V|.|E|)$.

5. The outermost loop has a constant-bounded number of iterations (at most 5), hence not affecting the overall asymptotic worst case runtime, therefore the algorithm has a worst case runtime $\in O(|V|.|E|)$.

Whilst the logic of this reduction appears sound, the claim of $O(|V|^2) \in O(|E|)$ is not valid in this context, and the reasoning behind this reduction is not given in any of the published literature. Due to the acyclic topology of DFG, the number of edges in a graph is not an asymptotic upper bound for the number of vertices squared. The assumption that $O(|V|^2) \in O(|E|)$ is for a strongly connected graph, which acyclic graphs invariably are not due to the very property of being acyclic. Therefore, the runtime complexity of ISEGEN is considered to be $O(|V|^3)$ throughout this thesis; a runtime complexity which is still polynomial and hence not subject to the massive intractability of the underlying space which is $\in O(2^n)$.

2.4.3 HDL Synthesis and Analysis

The synthesis from HDL (or from a higher level as appropriate) is a matter of taking the HDL description in a form which may be mapped to hardware, and lowering it until it is in a form which may be used to either fabricate the design described or program it into a reconfigurable

fabric such as an FPGA. HDLs normally have a subset of their semantics that for a particular synthesis technology is deemed to be ”synthesisable”. Care must be taken by engineers attempting to write a synthesisable design that they use only the language constructs which may be lowered to structural representations.

There are a wide array of tools available to help in this process, generally working from either the Verilog [31] or VHDL [32] languages at the top level and producing either an FPGA bitstream or a GDS-II schematic at the bottom level of the flow. Tools from Synopsys and Cadence usually feature in such a flow targeted at producing standard-cell implementations of designs, and the work of this thesis follows this trend by making use of the DesignCompiler and associated DesignWare libraries offered by Synopsys.

Most ASIC designs undergo a degree of iteration with regards to the synthesis flow. In order to facilitate feedback in the design process, various analysis can be used to create pre-fabrication reports on aspects of the design being synthesised. DesignCompiler produces reports on timing, area, clock gating, and power. The latter is achieved through combination of the PowerCompiler component of DesignCompiler, and switching information obtained through simulation. ModelSim from Mentor Graphics provides HDL simulation functionality throughout this thesis. When performing evaluation of HDL throughout this thesis where timing, area, power, or energy is referenced it has invariably been derived through synthesis using the DesignCompiler and a commercial 130nm standard cell library implementation.

2.5 Resource Sharing

To avoid bloating the die area with large numbers of extension instructions, it is important to identify and exploit commonality between instructions and, where possible, to share hardware resources when this represents a good trade-off between die area and execution time. Brisk *et al.*[33] have explored an approach based on finding the longest common sub-string, in order to determine which parts of a pipelined data path may be shared. This work was extended by Zuluaga *et al.*[34] in order to introduce parameters to control the process of merging data paths for resource sharing. The latter work has a focus more on parameter exploration, allowing for integration into a design space exploratory framework. Other approaches to resource sharing include [35], which utilises the graph-theoretical concept of cliques to accelerate the resource-sharing process. All (re-)configurable microarchitectures are inherently resource-sharing, as the same functional unit may be mapped to a number of distinct ISEs over time. The only difference between a resource-shared ISE microarchitecture which is *not* dynamically configurable and a resource-shared ISE microarchitecture which *is* configurable, is the means of changing the control signals governing the multiplexor selection and enable lines. The various costs involved (latency, area, and power) will generally govern whether a resource-shared ISE microarchitecture will be made configurable.

2.6 Compiler Transformations

Compiler transformations attempt to find a better representation of the semantics which have been input to the compiler as the code under compilation. These can occur at all levels of abstraction: source code, intermediate representation (SSA, RTL, etc), or Assembly Language. Different transformations apply at different levels, and some can be applied at multiple levels with varying efficacy. The combination and order of transformations is one that has undergone considerable research since compilers were invented, and continues to this day.

Early efforts to combine compiler transformations and ISE have been targeted at transforming CDFG towards a more efficient arithmetic structure [36]. This operates post automated ISE (AISE), so does not directly contribute to the design space search but improves upon the result.

In [37] it is shown that an exploration of *if-conversion* and *loop-unrolling* source-to-source transformations is successful in enabling better performing AISE. This work utilises control-flow transformations to move larger regions of the target application into the AISE algorithm at-once. The work in [37] demonstrates that new search methods and heuristics can be developed to control the application of transformations, with respect to the new set of goals inherent in ISE as compared to code generation. Transformations once targeted at the back-end would attempt to limit increasing basic block size due to register pressure. Now in ISE the drive is towards the largest possible basic block size for analysis. Note that the use of these transformations to increase the block size will also increase the run-time of the ISE algorithm, which for ISEGEN is subject to runtime $\in O(|V|^3)$. These must therefore be applied judiciously, or blocks may be made intractable to identification.

Source to source transformations have the useful property of being highly portable and widely applicable. Transformations from C to C have been used in a variety of contexts, optimising applications at the highest level so that they fit better in a variety of performance criteria. In [38] source to source transformations are used to optimise the performance of I/O operations. In [39] and [40] the focus of the optimisations is energy efficiency, and minimising redundant or wasteful operations. Formal verification is another attempted application of such transformations [41], whereby a program is iteratively transformed until it matches a specification. Of considerable relevance to this thesis, are cases where transformations have been used to explore the design space of multi-core optimisation for compute-intensive embedded applications. Examples include the work done locally in [42] or elsewhere in [43, 44].

There are tools and frameworks available for the construction and exploration of source to source transformations. Notably ROSE [45], Transformers [46], and COLOTool [47]. These provide the basic structures and transformations in order to allow a user to integrate them for the optimisation of C and C++ applications.

A empirical study of how combining adaptive compilation and machine learning is effective in exploring the design space of source to source transformations, is covered in both [47] and [48].

Transformations are essential to explore the data and process level parallelism inherent in any application. This section has demonstrated that compiler transformations should form an integral part of any strong framework to automatically extend an ISA.

2.7 Summary

This chapter has covered a variety of work necessary to understand and motivate the research performed in this thesis. The broad spectrum of instruction set architectures has been covered in order to demonstrate the context in which embedded processors exist, and their main differences from general purpose processors. Three somewhat overlapping classes of embedded processor have been defined, with the main contribution of this thesis falling into the domains of extensible and configurable cores. The main properties of these domains have been outlined, along with their potential for application-specific hardware/software co-design. The next chapter covers more specific details of embedded processors touched upon in this chapter. The methodology by which processor designs may be defined through iterative exploration is outlined: Design Space Exploration, which in this thesis is concerned with producing a hardware/software co-design to fulfil the functional and performance requirements of a particular application.

Instruction Set Extension (ISE) is introduced, which considers a hardware/software co-design as two partitions and facilitates the movement of function between the software and hardware partitions in order to benefit performance in a number of axes. The major constraints on ISE have been presented, which are either related to costs due to the extensions (area, power, energy) or constraints on the original core (register file ports, encoding space, scheduling limitations). An abstract definition as used in algorithmic exploration was defined, allowing the performance of ISE to be calculated based upon a simple linear equation and hence provide a basis for determining the merit of an ISE. The need to explore design spaces other than just ISE has been detailed, with particular reference to software emulation as an example source of misleading performance statistics. The need to address processor customisation as a holistic problem was stressed. Different areas of concern have now been identified (acceleration, area, power, energy, engineering time) and properties of ISE which are relevant to these were detailed. The Amdahl limit of ISE which is a theoretical maximum obtainable speedup through ISE was defined based upon the abstract model; this gives an asymptote of acceleration performance which cannot be surpassed (or realistically even reached) with just ISE.

Microarchitectural interfacing between host processors and extensions were covered and divided into three groups defined by the degree of communication overhead brought about by their distance from the host core's register file. Different microarchitectural implementations of ISE were covered; combinational, pipelined, and reconfigurable structures and their properties have been discussed. The latter may of course be combined with either of the former. The use

of scratch-pad memory in ISEs was touched upon, but this is not generally explored further in this thesis. A simple example of ISE identification based upon the abstract model and a matrix multiplication benchmark has been detailed to better illustrate the mechanics and specifics of the problem.

The techniques of automated synthesis used to automatically produce a hardware design based upon application and direct specifications has been covered. A major relevant area of automated synthesis now covered in detail is automated ISE, its separation into identification and selection phases, and the ISEGEN algorithm to perform ISE identification. The further processing of HDL descriptions of hardware into actual fabrication definitions has been described also. Resource sharing of hardware components has been outlined as a useful technique to reduce the cost of fabrication without excessively impacting the performance of the design. Finally, exploration of software transformation has been introduced as being important to both the quality of the executable code and to automated synthesis where application code is used, as in AISE.

This thesis now moves on to a more detailed look at some of the work related to the efforts undertaken herein.

3 RELATED WORK

"The best way to get a good idea is to get a lot of ideas."

– Linus Pauling

A number of approaches similar to that explored in this thesis have been explored by other people in the fields of DSE, ISE, and ASIP design. This chapter attempts to give an insight into those similar works which are relevant to the work undertaken herein.

3.1 ASIP Design Space Exploration and Co-Design Frameworks and Languages

DSE was introduced in section 2.2, and covers a great deal of alternative techniques and design methodologies. The concept of taking design as a space of choices is applicable to a number of approaches, and has spawned a number of languages and frameworks to allow for such work to be performed. Means of exploration vary from altering numerical or otherwise enumerable parameters, to direct modification of high-level structural and behavioural descriptions of a design. The common feature in all cases is a degree of abstraction between the fine-grain detail of the underlying ASIP design and the method by which the designer explores alternatives. The following are languages and tools which can be used to explore the design space of extensible processors to some degree.

3.1.1 Verilog

Perhaps the most pervasive of the RTL languages which may be used for design-space exploration is the Verilog Hardware Description Language, described in [31]. The Verilog language provides enough flexibility to be used for simulation, synthesis, and verification of digital circuits; later extensions to the language have also included support for analog and mixed-signal applications in the form of Verilog-AMS. The most recent incarnation of pure Verilog is the 2005 revision (IEEE Standard 1364-2005), which largely applies small refinements to the 2001 revision as covered in [31]. Despite the succession of new editions of Verilog, many designers prefer to use the older editions of the language such as Verilog-95, as the synthesisable subset is more explicit and hence less ambiguous in its mapping between language and synthesised structure. For example, the EnCore [10] processor is written entirely in Verilog-95 despite having been designed and implemented between 2006 and the present. More recently, the SystemVerilog language has been introduced as the successor to Verilog. SystemVerilog adds new types, software engineering semantics such as interfaces, and most significantly an object-oriented programming model. DSE using Verilog can be approached in a number of ways; utilising preprocessors, language constructs, or manipulation of behavioural descriptions. Di-

rectly manipulating structural descriptions is rather too low-level to be considered in design space exploration, but it certainly plays a major role in producing synthesisable designs and is often the subject of both preprocessor and parameterised manipulation.

Verilog itself defines a number of preprocessor directives allowing for parameterised duplications, inclusions, and definitions of any code. These are combined with the ability to perform calculations through macros at compilation time, providing a rich interface for DSE to be performed on either behavioural (high-level) or structural designs. A variety of alternatives for the preprocessor are available, and most synthesis tools also incorporate a preprocessor pass in their compiler. Often the separate preprocessors are used to perform multi-level preprocessing, wherein parameters and definitions are propagated to the source before it is processed by the synthesis tool. This allows for the design to be pre-processed to a parameterised form still containing macros and definitions where appropriate, so that it may be understood following initial parameterisation. In addition, the two-pass approach allows for more complex macros to be introduced, but can sometimes be rather complex to manipulate accurately which can lead to errors.

Verilog provides parameters which allow for a module to be modified based upon parameters passed to it on instantiation, which are effectively defined as constants within that module. This can be used in a way very similar to the preprocessor directives already described, with the major difference being that the parameters are a part of the language semantics and not the preprocessor. Default parameters can be provided and then overridden as required for reuse of a particular design, or DSE in general. The choice of using preprocessor directives or parameters is one subject to ongoing debate, but the general consensus is that preprocessor directives are better suited to system-wide constant definitions, whereas parameters are better suited to constant definitions local to a module [49].

ISE design exploration and implementation can be performed in Verilog, and the tools covered later in this thesis all use Verilog as a target language when constructing synthesisable models of the ISEs produced.

3.1.2 VHDL

The major competitor to Verilog is VHDL (Very high-speed integrated circuit Hardware Description Language), and the two languages share a lot of similar semantics and functionality. VHDL used to cover a higher-level set of behavioural abstractions than Verilog, but this has become less true as both languages have been further developed. VHDL includes both behavioural and structural semantics for describing hardware, a variety of data types, and modular engineering features in the same way that Verilog does. VHDL is generally rather more verbose than Verilog since its original purpose was in specification, whereas Verilog was originally intended for simulation. Like Verilog, VHDL has undergone a series of revisions culminating in the most recent IEEE Standard 1076-2008 [32]. The basic design units in VHDL are the

entity and *architecture*, which together are equivalent to the Verilog *module* in the sense that they declare ports and contain definitions of how that component should behave and how it is structured.

With regards to DSE, VHDL can be both preprocessed and parameterised in a similar way to how Verilog can. The VHDL preprocessor is not generally included in the compilation of the VHDL itself, and must be run prior to compilation. A number of VHDL preprocessors exist, and many have equivalent features to those of the Verilog preprocessor directives allowing a similar approach to parameterised DSE. Parameterisable components in VHDL can also be created using generics, which can be used in the same way that parameters can in Verilog with regards to reusable designs and design space exploration. Differentiation is made in VHDL between constants and values used to parameterise modules, whereas both are parameters in Verilog. The choice of whether to use Verilog or VHDL in DSE or indeed ASIC design in general is largely down to the existing IP and tools available to the engineer, and which language they have the most experience in.

3.1.3 SystemC

At a higher level than the HDL languages discussed so far, SystemC is an extension of C++ via a library of classes and macros, defining an event-driven simulation kernel through which hardware can be evaluated. Verilog and VHDL are suited to describing the lower layers of design abstraction all the way down to gate-level models. SystemC is better suited to system-level modelling. The most recent standard for SystemC is the IEEE 1666-2005 standard [50]; this represents the culmination of efforts by the Open System C Initiative (OSCI) who are generally in charge of the definition of the language. Despite the fact that it is essentially a C++ library, SystemC is often referred to as a language due to the semantics of using the libraries themselves. There are methods to perform high-level synthesis from SystemC to lower-level descriptions such as Verilog or VHDL, and these can be mapped further to ASIC (GDS-II) or FPGA (bitstream) hardware implementations. This is generally not used where performance is critical, as the degree of control over resulting circuit structure is less than that available in Verilog or VHDL unless RTL semantics are used. When using RTL semantics, simulation is typically slower than that of Verilog and VHDL simulators and there is a considerable syntactical overhead. SystemC is well suited to the integration of existing components into a system level design, and may be combined with Verilog models for this purpose [51], yielding more accurate timing feedback than if the entire design is defined in SystemC.

For DSE, SystemC can be used as a functional language to perform co-exploration of system-level organisation and algorithm design. SystemC can also make use of the C preprocessor or C++ semantics to enable parameterisable DSE. Exploration will only allow for cycle-accurate performance modelling when functional description is used: Using high-level synthesis of a functional model to determine timing is not a reliable way of determining per-

formance if it is eventually implemented structurally in an RTL fashion. Improvements are being made to commercial SystemC high-level synthesis, so it may be more suitable for high performance hardware implementation in the future. Several case studies have been made using SystemC for DSE. One concludes that it is well suited to exploration of SoC designs using Transaction Level Modelling (TLM) [52]. TLM provides a high-level approach to modelling communication between components, allowing for greater simulation speeds than RTL. SystemC is better used for DSE where the design is to be explored at a system level, where the design space is largely defined by the number and organisation of pre-defined components. SystemC has been used to facilitate DSE of application-specific ISE for specific applications; for example in [53] and [54].

3.1.4 SA-C

Single-Assignment C (SA-C) otherwise known as "Sassy" [55] is a language based on C which attempts to incorporate the features from existing imperative and functional languages which enable accurate dependency analysis and hence mapping onto parallel hardware. Sassy is intended to exploit both coarse-grain (e.g. loop-level) and fine-grain (e.g. operator-level) parallelism. The language has explicitly parallel loop structures (e.g. *foreach*), and has removed both pointers and recursion in order to ensure the tractability of a structural mapping. Arrays are the mainstay of data storage in Sassy, and the language includes semantic constructs for concise creation and manipulation of arrays, which are promoted to first-class objects. The support of multi-dimensional arrays enables image and video processing. Array windowing semantics are included in order to allow algorithms implemented in Sassy to detail the memory scope required. Scalar variables are considered as wires, not memory locations; the Von-Neumann model of computing is deliberately avoided in Sassy in favour of an approach allowing direct circuit generation.

The circuits derived from Sassy are emitted in VHDL, utilising parameterised libraries. Handshaking control signals occur both between the components emitted as a Sassy circuit, and between accelerators and the core they are hosted by. Loose coupling introduces a number of cycles overhead versus a more tightly coupled solution. In general the further an accelerator is from the data-path of the host core, the more cycles will be wasted in handshaking and data transfer. For this reason, large portions of code (and data) must be offloaded to accelerators outwith the host core in order to actually gain an advantage through their use.

Loops and arrays are closely related in the Sassy language, which is what leads to the ease of mapping large-scale parallelism from loops to a reconfigurable architecture utilising FPGA as streaming coprocessors as in [56]. This approach is not one of instruction set extension, but is very closely related due to the use of DFG as an intermediate abstraction, for both program structure and application-specific reconfigurability. It would be trivial to adapt existing ISE techniques to operate on the DFG representation in the current Sassy compiler, and the

process would not be plagued by the problems inherent in performing AISE on C source; e.g. referential ambiguity. Sassy has been touted [55] as appropriate for a variety of highly parallel computing platforms including SMP and vector processors, the latter of which can be considered a close relative of application-specific ISE. The SA-C compiler is unfortunately not available for free download and so can not be modified for use with ISE and AISE without a large degree of effort re-implementing the entire language. DSE using Sassy is largely replaced with transform space exploration, as the mapping from Sassy source code to FPGA is direct in the current implementation. Compiler pragma's may be used to explore a number of different transformations upon the intermediate representation, including but not limited to strip mining, unrolling, and fusion [56].

3.1.5 Handel-C

Another extended subset of C intended for hardware generation is the Handel-C [57] language, originally designed in the Oxford University Computing Laboratory and later developed by Celoxica, Catalytic, Agility, and more recently Mentor Graphics. The most notable language features of Handel-C are arbitrary bit-width expressions and variables, plus communication and parallelism semantics borrowed from the Communicating Sequential Processes (CSP) language [58]. The SA-C language is designed with image processing explicitly in mind. Handel-C has been designed for more general hardware applications. The academic Handel-C implementation [59] differs from SA-C in that it does not map to a number of coprocessors; instead it maps the entire application to a single FPGA circuit which is coupled to and controlled by a transputer [60]. A simple example also exists in [59] of constructing an entire ASIP via transformations over a Handel-C program, so the language has potential for various hardware-software co-design applications. Handel-C has been used for image processing, neural networks, video processing, database servers, and other applications. Compilation in Handel-C is performed by a set of transformation rules, borrowed from CSP; a program is converted to IR form and then sequentially transformed by these rules until it reaches a so-called "normal form", representing the final state of the program as it is to be mapped into hardware. Control flow is converted into state machines wherever possible, and the normal form of the program is able to be immediately converted into a net-list which may then be compiled to hardware. Variables become flip-flops and expressions become combinational logic, which is different from the SA-C approach wherein variables are considered as wires and storage is via array memory only.

DSE using the Handel-C language is similarly achievable as in SA-C: the language itself can be used to perform algorithmic exploration, and the transformations which are performed over the resulting language can be controlled to a degree to allow different reductions of the same Handel-C program to distinct hardware. A preprocessor and the language itself may also be used to form a parameterised space by which a design may be evaluated in various axes.

The use of Handel-C for constructing an ASIP is covered in [59], but the transformation steps taken are not automated and require a considerable degree of forethought and intervention in the design process. The separation of the hardware element and the software element is not implied by the original program, and is instead governed by the set of transformations used to convert the original program to a lowered form.

3.1.6 ROCCC

The Riverside Optimizing Compiler for Configurable Computing (ROCCC) [61] is a C or Fortran to FPGA compilation framework allowing for properly structured loop nests in programs to be transformed to FPGA circuits. ROCCC takes C as input and emits a combination of VHDL (to program the FPGA) and C (to execute on a host processor); a combination which the authors of [61] term a "Configurable System on Chip". ROCCC is built on top of the existing SUIF [62] and MachineSUIF compiler frameworks, utilising the existing compiler and intermediate representation for the hardware analysis. This integration of hardware generation software with existing compiler infrastructure is a common theme in automated hardware-software co-design, where the language used was not designed for the purpose (e.g. C and Fortran). Loop unrolling is employed to expand loop bodies so that when they are converted to hardware, effective use is made of available loop-level parallelism. Note that this is converting loop-level parallelism to operator-level (OLP). Area estimation of the hardware resulting from unrolling is used to govern the number of times the loop is unrolled before conversion to hardware. Profiling is used to determine which of the loops in an application are "hot", so that hardware construction is targeted only at those sections of code which will benefit application acceleration. ROCCC is intended for data-flow heavy applications, as the acceleration provided by its hardware generation only benefits data-intensive loop sections. There is no provision for general control flow as in Sassy [55] or Handel-C [57]. Loop statements are the only control-flow construct covered by hardware, and these are converted into the control logic of the resulting accelerator, governing the buffers and data-path contained therein.

DSE utilising ROCCC can be performed using C preprocessor directives such as definitions and macros; additionally the source code itself can be manipulated in order to get feedback on the performance of the application in a hardware-software co-design. SUIF compiler optimisations are employed in order to improve the suitability of the IR for hardware synthesis, although this is not made controllable for transformation space exploration purposes. The COLOTool [47] is an extension of SUIF which enables source-source transformation, which could be combined with the ROCCC to achieve transformation exploration (although this combination is not covered in the literature). Work has been performed to explore the design space of Discrete Wavelet Transforms with ROCCC in [63]. Accelerators produced by ROCCC are somewhat similar in structure to the CFA described later in this thesis, in that they contain a pipelined data-path to cover the code offloaded to the accelerator.

3.1.7 SPARK

The SPARK framework is another C to VHDL tool-chain [64], also building on existing compiler technology which was originally developed to extract and map parallelism. In a manner similar to SA-C [55], the SPARK system does not allow pointers or recursion. These language features both inhibit the construction of hardware by requiring mapping to a full memory space and stack. Since the compiler technology behind SPARK was developed by the same people who developed SPARK itself, they have had a high degree of control over its customisation to suit the high-level synthesis application it has been ultimately applied to. The SPARK compiler uses DFG and Hierarchical Task Graphs (HTG) to form its intermediate representation; these are innately excellent representations for hardware synthesis as they allow for a direct structural mapping. No translation from SSA form to structural form is required, since the application is immediately converted to a structural form from the AST. Hardware in the SPARK framework is generated as an extension to an existing processor, utilising FPGA technology to implement DFG that are controlled via memory-mapped registers. These DFG are maintained by state machines which are also generated by the synthesis process, but the host processor is the only component capable of actual control flow.

The original SPARK compiler technology was heavily oriented towards exploiting parallelism, and contains a swathe of transformations which are intended to improve the quality of the IR and increase parallelism. CSE, IVA, Folding, Function inlining, Operator Chaining, Loop transformations, Percolation, and Trailblazing may all be applied in order to explore the transform space of the source code. As per other C-based high level synthesis frameworks, preprocessor directives or C semantics may be used to parameterise the source and perform parametric DSE. Algorithmic exploration may also be performed manually, which will indirectly effect the structure and hence cost/benefit trade-off of the accelerator hardware. The DFG and HTG forms used in the SPARK IR would be very suitable for AISE analysis, but the authors of SPARK have not yet investigated this potential. The distance of the accelerators from the core adds overhead in data-transfer and control, which could be alleviated by moving the system to a tightly coupled approach such as ISE.

3.1.8 DWARV

The Delft Workbench Automated Reconfigurable VHDL (DWARV) generator [65] is yet another C to VHDL framework, designed with the intention of targeting the MOLEN polymorphic processor (described later in section 3.3). The DWARV generator is constructed in a similar fashion to ROCCC and SPARK, in that it is based upon existing compiler technology (SUIF2 [66]) and intended to exploit operator-level parallelism. The DWARV tool translates code to Hierarchical DFG (HDFG), then performs a hardware-software partitioning of the code and emits pragma-annotated C, denoting the parts of the source that should be implemented

as hardware. The HDFG is similar to the combination of HTG and DFG used in SPARK. The annotated source is then processed by the MOLEN tool-chain to produce the lowered hardware-software co-design. Limitations on the implementable subset of the source language (C) include only allowing single-dimension arrays, disallowing all compound types, and not including floating point operators. Control flow may be implemented within the accelerators produced, but these are constrained to *for* (iteration) and *if* (selection) constructs. DWARV is generally better suited to arbitrary application domains, whereas other C-VHDL synthesis frameworks are generally better suited to DSP.

Suitability of DWARV to DSE is identical to that of SPARK or ROCCC: the C preprocessor and the C language itself may be used to perform a degree of DSE or algorithmic exploration. Source-level transforms may also be used, whether these be manually applied or evaluated through an exploration framework such as COLOTool [47].

An empirical comparison of ROCCC, SPARK, and DWARV in the context of FPGA-based reconfigurable design is covered in [67]. It should be noted that the work performed was done at Delft, who are the creators of DWARV. The conclusion is that DWARV covers the largest subset of C, with the fewest restrictions on how this subset may be used. DWARV does not allow for designer-applied transformations to be used, which limits the DSE potential of the DWARV framework somewhat. ROCCC is claimed to be the hardest to target code for, as it covers the most restricted subset of C and the windowing strategy with regards to arrays supposedly presents further difficulties. It is wise to note that none of these frameworks allow for seamless translation of arbitrary C code to VHDL; good performance requires a degree of knowledge on the part of the software engineer of the underlying microarchitecture. DWARV claims to target software engineers without any knowledge of the hardware, which has led to it being the easiest to work with according to [67]. Despite these observations on the ease of use of DWARV, the study of [67] concludes that the best performance is to be obtained with SPARK, due to the transformations applied to hardware and FPGA-local storage. SPARK requires the most knowledge about the underlying hardware when retargeting.

3.1.9 LISA

Architecture Description Languages (ADL) are an extremely potent tool for performing DSE over a number of architectural alternatives, when designing an ASIP. The LISA [68] language is one of the most widely used ADL in use today, allowing a description of the entire processor including both behavioural and structural elements to be created as a “Golden Model”. The golden model may then be used with the LISATek tool [29] developed by CoWare [69] as the basis for generating a number of deliverables that are generally required in an ASIP design:

- Synthesisable Verilog or VHDL model of the core, allowing ASIC or FPGA implementation of the core described.

- Instruction Set Simulator, able to execute binaries compiled for the machine description given.
- Compiler based upon the CoSy [28] compiler construction system, allowing C/C++ to be compiled for the architecture. The work in [70] describes the generation process.

The complexity of description required depends on the degree of accuracy required in simulation, and whether or not the model is to be synthesised. Cycle-accurate simulation and synthesisable RTL can be achieved by describing the pipeline structure. The bare minimum description requires details of the instructions, their mnemonics, encoding, and operation semantics. This allows for a basic compiler and instruction-set simulator to be generated. More recent additions to the LISA language (now at version 2.0) include the ability to describe RFU's [71], allowing for dynamic reconfiguration to be included in the processor specification.

LISA is a powerful tool for design space exploration, as much early work can be performed using a purely behavioural description. The "golden model" can be slowly lowered to a more specific structural specification once features are determined. DSE of reconfigurable processor elements is performed in [71], and earlier efforts to use LISA to perform DSE on standard processors proved successful also [72] [73]. A book exists describing the process of performing architectural DSE using the LISA language [74]. AISE efforts can generate LISA descriptions of new instructions, and the LISATek tool-chain can be used to perform empirical evaluation of the extended processor. The author of this thesis has encountered various problems with the quality of the synthesised Verilog and VHDL, and so the language was not adopted for the work described in this thesis. More recent work on LISA has focused on improving the quality of the RTL generated [75].

3.1.10 MESCAL

The Modern Embedded Systems, Compilers, And Languages (MESCAL) group [76] at the Gigascale research center do not present a single software framework, but instead present a methodology for the design of ASIP. The methodology was derived from consultation with "leading industrial experts" and empirical evaluation of the software frameworks available. The methodology breaks down to a number of points, described in [77]:

- **Judiciously Using Benchmarking**; *The use of benchmarks to determine the performance of an architecture is essential to determining its practical use and performance.*
- **Inclusively Identifying the Architectural Space**; *Identifying the mutable features of the architectural design space is important to ensure all options are evaluated.*
- **Efficiently Describing and Evaluating the ASIP**; *The means by which design space exploration is facilitated effects the amount of work expended in describing the options*

available to the designer. Using an ADL such as LISA [29] in order to both describe and evaluate the ASIP can provide a large improvement in productivity and accuracy over more manual methods.

- **Comprehensively Exploring the Design Space**; *Leaving no stone unturned with regards to the scope of DSE is very important in order to ensure that the best design is obtained.*

- **Successfully Deploying the ASIP**; *Deploying an ASIP commercially includes providing documentation, compiler, and simulator at a minimum. The quality of all this supporting work is very important to the success of deployment for an ASIP; again ADL such as LISA [68] supporting specification-driven generation of all these additional deliverables can prove invaluable.*

One example of work utilising the MESCAL methodology is that of [78], wherein the methodology is followed in the development and use of an Integer Linear Programming based tool for exploring an FPGA multiprocessor design space. A network of processors connected using a hierarchy of buses and FIFO is constructed and evaluated, and this represents a single design point in the exploration undertaken. The tool developed takes an application as a task graph, and attempts to maximise the throughput of the application-specific multiprocessor design with respect to the application.

3.1.11 Lime & Liquid Metal

The Liquid Metal project at IBM is a recent attempt to extend Java with further semantics to aid in the mapping of the language to a spatial hardware implementation. The language that is currently the focus of this project is called Lime, which is a high-level object-oriented language backwards compatible with Java [79]. The long term goal of the Liquid Metal project is to "JIT the hardware", meaning that on a hybrid CPU/FPGA system, the FPGA will be reprogrammed in real-time given the same kinds of dynamic information that existing JIT compilers rely on in order to optimise code as it runs. The Lime language can currently be compiled to run on a standard Java VM or on FPGA hardware; a combination can also be used, with the intention that the transition between the two execution modes be seamless. The Liquid Metal Runtime (LMRT) is the software which synchronises the two domains of execution, and handles the loading of FPGA sections on the fly.

Lime introduces new types similar to those in Kava [80], but with less complex type rules and compatibility with existing Java. Most notable is the *value enum*, which provides an immutable bounded value with a default value (these cannot be null). Iterators may be derived from the enumerable types, and can be used with *for-each* semantics to provide explicit definitions of parallel code. Arrays may also be referenced using the *value enum* type, ensuring

accurate static size and bounds analysis, and enabling bit-level analysis. Lime also allows for user-defined operators, such that new operations can be defined through method definition.

The Lime tool-chain includes a compiler which converts Lime source-code into either byte-code for a JVM or Spatial IR (SIR), which is an intermediate representation particularly suited to hardware synthesis. SIR is a DFG-based IR, perfectly suited for exploiting spatial parallelism (OLP) where this is identified. Nodes of the SIR graph are *filter*s, which are connected by edges representing communication channels (implemented by buffers). Each *filter* represents a single method from the original Lime program, edges between these represent method parameters and return value.

A high-level SIR compiler lowers the SIR from the form presented by the Lime compiler and performs some optimisations, mostly exposing spatial parallelism. Layout is then performed to map the lowered SIR to FPGA resources as appropriate, and finally the RTL is emitted and synthesised in order to obtain bitstreams for the FPGA.

The Lime language may be used to explore the design space of the FPGA implementation used in the mixed-mode execution of the applications written in it. Whilst code transformations are employed in the tool-chain, further source-level transformations can be explored manually by an engineer. The object-oriented nature of the language combined with the static semantics and analysis also make for effective tools with which to implement parameterised DSE. The SIR used for the Lime language could be retargeted for use in ISE, and the additional semantics in Lime could be as useful to automated ISE design as they are to automated FPGA data-path construction.

3.1.12 Trimaran

Compiler and simulator technology is a critical part of many DSE frameworks, and one of the most commonly used systems in DSE is Trimaran [81]. The Trimaran infrastructure consists of:

- **The OpenIMPACT compiler**, which translates source code to the Lcode IR.
- **The Elcor compiler**, which translates the Lcode IR combined with a machine description in MDES format to the REBEL IR, or to native code in one of the available backend architectures.
- **The Simu simulator**, which takes a machine description in MDES format with an application in REBEL IR format and produces performance statistics.

The Lcode IR is a machine independent assembly form which exposes the instruction level parallelism of the underlying application. Classical loop-level transformations such as unrolling, and super- and hyper-block formation are used to optimise the OLP available in the Lcode IR.

The REBEL IR is immediately formed from the Lcode IR in the frontend of the Elcor compiler, and is processed by classic scalar optimisations (DCE, CSE, CP, etc), then mapped onto the machine-description given through the MDES information supplied. Automated mapping of ISEs is possible by defining the acyclic DFG representing the ISEs, and the opcode format in the MDES format. Elcor processes the REBEL intermediate representation and replaces all isomorphic graphs with the new custom instruction. This feature makes Elcor and hence Trimaran a very attractive option to AISE developers and those wishing to perform DSE within the space of acyclic fine-grained ISE. Vectorisation can also be automatically exploited, assuming the vector capabilities can be defined within the MDES format. Clustered architectures are supported too, and the compiler provides automated mapping for these. Once the Elcor compiler is finished mapping the code, it can emit it either in a native instruction set or in REBEL IR format. The latter format can be used in the Simu HPL-PD simulator to get early performance statistics, through parameterisation and simulation of the HPL-PD parametric architecture. Whilst the HPL-PD architecture does not cover all possible architectures, it gives an easily targetable polymorphic architecture which may be used for early feedback on the suitability of architectural features to a particular application, and hence is useful for early DSE. DSE in the Trimaran framework is easily facilitated using the MDES machine description format; essentially an ADL in its own right, MDES allows for a range of machine features to be defined and then used in the compilation and simulation process. An example of DSE using the Trimaran framework is in [82], which performs exploration of VLIW ASIP with coarse-grained functional units.

3.1.13 Other Languages and Frameworks

This section has so far covered only a select few of the languages and frameworks which may be exploited for the purpose of design space exploration. These have been selected to give a cross-section of the available approaches to hardware-software co-design. Some other notables include:

- **Streams-C** [83]; Another implementation of streaming extensions to C utilising the Hoare CSP model. Streams-C is constituted of a number of annotations (hidden from the standard C compiler by comments) and C library functions. The version covered in [83] is intended to target FPGA accelerator cards on the PCI bus in standard PC hardware.

- **Optimus** [84]; A more recent descendant of languages such as Handel-C and Streams-C, Optimus comes from the same group at IBM that developed Lime [79] and is intended to be used to efficiently target streaming applications towards FPGA implementations. Optimus is not a language but a synthesis framework which operates off the same SIR intermediate representation as the already discussed Lime language. The SIR in [84] is

generated from the StreamIt [85] language rather than from Lime, although it could be generated from a number of other languages.

- **EXPRESSION** [86]; An ADL which may be edited directly or edited by a GUI, which allows for architectural DSE in a similar manner to LISATek [29] by generating a simulator and compiler for the architecture specified.

- **LLVM**; An extremely well documented and powerful compiler framework including the Apple-sponsored *clang* C compiler, which provides a much higher caliber of static analysis than has previously been available in free compilers such as GCC. Contains intermediate representations in a DFG structure, which in itself is essential to the automated hardware-software design process. LLVM will almost certainly feature in future projects to perform hardware-software co-design, as it is maturing rapidly into a reliable platform for compiler development.

- One of the earlier but mature attempts to formalise DSE for an application-specific VLIW architecture is [87], from work at HP Labs in 1996. The work focuses on parameterising a VLIW architecture and exploring the parameter space with regards to a number of kernels and applications as benchmarks. Exhaustive search is performed to determine the space, and the resulting performance graphed to provide insight into the trade-offs. A similar approach is adopted in many experiments throughout this thesis.

3.2 Automated Instruction Set Extension

When the process of identifying and selecting ISEs is made algorithmic through computerised processes, it is referred to as Automated ISE (AISE). The following are alternative approaches to AISE than the ISEGEN algorithm already covered, which are included to paint a picture of the diversity present in this field.

3.2.1 Linear-Complexity MISO Identification

One of the simplest approaches to AISE is given in [88] and later in more detail in [89], wherein the MaxMISO algorithm is presented. A MaxMISO is a Maximally expanded Multiple-Input-Single-Output graph, which is generated by selecting a single node to output from and iterating backwards over the fan-in to that node and its predecessors. This process is repeated until a barrier predecessor which is unimplementable (e.g. LD/ST) is encountered. Input port constraints are not considered by the algorithm due to the idea that these should be relaxed, having been identified as a bottleneck in previous works. The output port constraints are never violated by this algorithm due to the inclusion of only a single output node; essentially ensuring the output port constraint by construction. A similar effect may be noted with regards to the

convexity constraint, in that the algorithm only produces nodes which are topologically defined by a tree expanding backwards from the output node; at no point is a solution considered where there is a hole between one node and a predecessor of it. The linear complexity of this algorithm is due to only considering each node once for inclusion or exclusion, and further many nodes in the graph are not even considered in most cases. This low complexity leads to a sub-optimal exploration of the design space, especially in the situation where application basic blocks present parallel data-flow where there is more than a single output. Vector-like operations are not well exploited by this algorithm as these require multiple outputs as well as inputs, which means many great opportunities for acceleration are missed. This deficiency is motivated in [89] by the supposed lack of register file I/O in embedded processors; in reality more modern embedded processors allow a large number of I/O ports in order to exploit larger, more effective complex instructions. This algorithm suffers from a similar problem as that of [18], in that it exploits properties of the search space to reduce the search complexity. The algorithm is not cognisant of the actual performance of the ISEs identified, and instead assumes that maximally-sized candidates will confer the greatest advantage. An extension of the MaxMISO idea is presented in [90], wherein a similar algorithm to that of [89] and [88] is presented. The only differences between the algorithm of [90] and the work of Pozzi *et al.* is that in [90] the algorithm is extended to iteratively derive all SubMaxMISO (SMM) subgraphs of MaxMISO which have greater potential to satisfy realistic input port constraints, with the intention that these subgraphs be further combined to form the ISEs for the application.

3.2.2 Linear-Complexity MIMO Identification

A natural evolution of the MaxMISO idea is to allow multiple outputs, which is the strategy employed in [91]. A MIMO is a Multiple-Input Multiple-Output subgraph of a DFG, with the property that it is constructed as the union of a number of MISO graphs. This hierarchical construction is what contributes to the linear complexity of the approach in [91], since the combination of a number of disjoint MaxMISO will produce an ISE which is convex by construction. In order to preserve the linear complexity of the algorithm, this hierarchical approach is adopted wherein each node is treated as the root of a MISO, and a spiral search is performed [92] on the DFG to produce MIMO from combining MISO defined by each node. The spiral search is based on the concept of the Archimedean spiral line, which is defined by a point moving with constant speed along a line rotating around one end at a constant angular velocity. This concept is used in [91] by making the seed node O (the node from which the search grows the MIMO) the center of a spiral, and mapping all other nodes onto integer levels above, below, and parallel to the seed node based upon the original graph topology (and hence dependencies which could lead to a convexity violation). The graph is constructed by considering nodes on the level intersecting a spiral outwards from O, and nodes which respect a certain property P are included in the graph. MaxMISO or SMM can be combined if the dif-

ference in their level is either zero or one, because this guarantees convexity of the combined MIMO. During construction, the resulting MIMO is always convex; this limits the potential for combination of disjoint portions of the graph, and this essentially is the limiting factor of this algorithm. The linear complexity is ensured by only allowing graphs to grow outwards from a pre-determined point. The search stops once no further nodes can be added without violating the convexity constraint, and has no consideration of register file port I/O at all. In addition, the point O from which each graph is grown is arbitrary and difficult to correlate with the ultimate desired properties of the graph; a random selection seems to be favoured in the literature [91]. Whilst the algorithm is linear in complexity, the solutions are likely to perform poorly and on I/O limited architectures will often be invalid. Where graphs from an application present large open areas of data flow without any barrier nodes, and where architecture has no I/O limitations (e.g. where I/O to ISEs is pipelined), this algorithm will perform reasonably well. Where data-flow is fragmented by memory or other barrier operations, and where I/O is limited per-ISE, this algorithm is likely to perform poorly. Addition of further heuristics and constraints to this search is largely impossible as the algorithm relies on topological concepts in order to guarantee correctness by construction. This is an extension of the ideas of [89], and suffers from the same problems as [18]; using structural properties of the graph to reduce the search space leads to inflexibility of the algorithm when new constraints and optimisation objectives are required.

3.2.3 Integer-Linear Programming Methodology

The Atasu AISE algorithm [20] generates DFG templates through conversion of DFG to a set of constraints in ILP, followed by solution of that program. The algorithm was implemented by the author of this thesis in an early effort at implementing AISE, as a tool built into a CoSy compiler using the *lp_solve* library [93]. Each DFG from the application (taken from expressions in [93] as these represent side-effect free sections) is converted into an ILP program, representing the design space available and the two constraints used to ensure that resulting ISEs are implementable: register file I/O and convexity. In addition to the constraints, a goal function is also expressed. For this algorithm, the goal is the estimated serial time of execution in cycles of the instructions covered by the template, minus the estimated critical path of the template. This is the same model as is presented in section 2.3.1, and is common to many AISE algorithms because it is considered generally accurate. The constraints in [20] are expressed in boolean, and must be translated to integer-linear form using a variety of equivalent operations involving integer-linear operations such as addition and inequalities. This in effect translates a boolean search problem in $O(2^n)$ into a linear search problem in an equivalent sized space, which may then be approached by the advanced constraints-based search of the ILP solver. The quality of the solution is in part down to the quality of the ILP solver, and ultimately the search space remains exponential. The freeware *lp_solve* solver performed sufficiently for the smaller

DFG approached in [93], but larger templates require the more sophisticated algorithms of a commercial solver such as CPLEX. The ILP approach to AISE is not strictly deterministic for this reason, and the result if found is only guaranteed to satisfy constraints; the quality of the result is largely dependent on the efficacy of the ILP solver. It is also difficult to extend the approach to include further heuristics in the objective because of the complexity this adds to the ILP. Following the generation of templates from basic blocks, in [93] the templates are checked for isomorphism with one another using the *NAUTY* [94] graph isomorphism library, then ranked using the product of their estimated usage and per-execution gain as per [20]. The top N of these instructions are then recorded alongside their performance estimates for inclusion in results, where N is defined by the encoding space available. Whilst the majority of the work in this thesis is with regards to the ISEGEN [7] algorithm, the latter transform work is done using the original Atasu ILP [20] algorithm implementation from [93], due to framework availability.

3.2.4 Fast Clustering AISE Algorithm

Verma *et al.*have taken a new approach to the problem of partitioning a graph into hardware and software partitions. An algorithm is proposed [18] which imposes a requirement of monotonicity upon the merit function (classically, and in their algorithm, limited to speedup in cycles only). Through the monotonicity of the relationship between speed-up and graph-size, the algorithm [18] is able to greatly reduce the underlying search space that the algorithm must operate over. A DFG of the application is first processed to find its clusters (groups of operators which may always be contained together in one ISE), and turned into a cluster graph. The cluster graph represents all nodes of the original DFG in classes; if nodes x and y are in a class together, then in an ISE which includes x, y can also be included without breaking convexity, and vice-versa. The resulting cluster graph compresses each class into a single vertex, and an edge is placed between every pair of classes which could be merged into a single ISE without violating convexity. This cluster graph has significantly fewer edges than the original basic-block DFG. Following the production of the cluster graph, it is then processed for maximal cliques (maximally-connected sub-graphs of the cluster-graph which are not sub-graphs of another maximally-connected sub-graph). These maximal cliques correspond to maximal valid subgraphs of the original basic-block DFG.

An issue with the monotonicity requirements of the algorithm are that extension of the merit function requires monotonicity in any further merit consideration. It is also questionable that the monotonicity of even this simple merit function can hold in all cases. For example, in most RISC processors (as are required by the algorithm), the clock-frequency is normalised around a single arithmetic instruction's latency. The clock-period defining operation often given in the literature is multiply-accumulate. Regardless, this operation will have a hardware latency very close to the software latency. There is the possibility that wiring added to route in a new

node in an already crowded graph will add to the total latency of the operator so much that the operator itself is now slower in hardware than software. If this effect is repeated (serially) then the possibilities are that ultimately the larger a graph gets, the slower the graph gets than software execution. Whilst this problem is endemic to all AISE approaches at present, the tendency to produce the largest possible graphs will run into wire delay problems more often than approaches which equally favour smaller ISEs. Empirical examples of the fallacy of this "bigger is better" approach follow later, in sections 5.4, 5.5, and 6.3.

For the purposes of this thesis, a variety of metrics for the merit of the instruction set extensions generated are expected to be used together. This will produce merit functions which are non-linear and non-monotonic; at this point this algorithm will no longer be able to support the intended merit functions and will fail to produce a good quality of ISE. For this reason, despite the smaller design space and execution time of this algorithm it has not been adopted for the research discussed later in this document.

3.2.5 Polynomial-Complexity Identification and Selection

The work of [95] converts the problem of ISE generation into one of combined identification and selection, through exploiting the relationship between the graph-theoretical properties of vertex dominators and convex sub-graphs. All DFG subgraphs within a particular I/O constraint are enumerated using the concept of k-vertex dominator nodes, wherein a set of k vertices dominate a number of nodes. The value of k is determined by the number of inputs or outputs to a graph. An additional constraint is imposed in order to aid in search, in that any input to an ISE w must have a node v within the ISE such that there exists at least one path from the DFG root (artificially introduced single predecessor node to all DFG inputs) to v that includes w but no other input of the ISE.

The algorithm uses the idea that the set of inputs and outputs to a cut define the convex cut itself, and this is proven in [95]. This in itself proves that the search space represented by convex ISE is actually polynomial in the number of inputs and outputs, rather than exponential as when identification is considered as an arbitrary partitioning problem. By selecting a set of inputs and outputs, a cut is defined through calculating the k-vertex dominator nodes of each output and including every node on the path between those nodes and the output in the cut. The construction of a cut from inputs and outputs selected guarantees that the inputs will remain as selected, but new outputs may be introduced by the process. Through exhaustive enumeration of the potential inputs and outputs using k-vertex dominators to define the cuts between, this approach explores the space in $O(n^{N_{in}+N_{out}})$ time. This naive approach is not entirely suitable for larger basic blocks [95], so incremental construction utilising additional pruning is introduced to approach larger DFG. The complexity of the incremental algorithm is the same as the naive algorithm, but pruning can be employed to reduce the value of n. Cuts including forbidden nodes are immediately discarded, and cuts with output v wherein an

ancestor w of v is a forbidden node can immediately exclude all ancestors of w from the set of potential inputs. The incremental search recursively selects all outputs o, and then an input from the set of k-vertex dominators of each selected o, then another output, and so on.

This algorithm of whilst lower in complexity than some other approaches has a high worst case memory requirement. As the cuts are being explored, all non-pruned combinations of input and output will be stored on the stack due to the recursion employed. The exhaustive enumeration of all valid ISEs within a DFG can be combined with arbitrary heuristics to select a combination of those ISEs for implementation. This enumeration places a large computational burden on the ISE selection process to evaluate each of the candidates for inclusion. The ISE selection process can be made equivalent to the knapsack problem. The problem is NP-complete and hence optimal solution becomes intractable for larger problem sizes, i.e. the number of ISEs provided to the selection stage. This approach therefore has a high memory requirement, and additionally seems to transfer the burden of NP-complete runtime complexity to a later stage in the ISE generation process. For this reason, the algorithm of [95] is avoided in favour of the ISEGEN [7] algorithm in the work of this thesis.

3.2.6 Tensilica XPRES

The Tensilica XPRES compiler [25] separates the customisation of an ASIP into three areas which effect the ISA and microarchitecture somewhat orthogonally: VLIW, Vector Operations, and Instruction Fusion. Customisations are defined in the TIE (Tensilica Instruction Extension) language, which is processed by the TIE Compiler to produce a compiler tool-chain, simulator, and RTL which is used to extend one of the XTensa (e.g. XTensa LX [96]) processors. The XPRES compiler is able to analyse an application written in C or C++, producing a range of alternative designs represented as TIE descriptions organised in a Pareto-optimal [24] trade-off function between the cost (area) of a design and the resulting performance improvement (generally acceleration).

The AutoTIE approach utilised in the XPRES compiler is covered in [97]. The approach differs from other major AISE techniques by including all three of the mentioned areas of customisation. Other approaches only really utilise the Instruction Fusion approach which is ISE as defined in section 2.3.1. The AISE methodology used in [97] has some similarity to the approach required when using [95], in that thousands or millions of potential ISEs (and other architectural customisation options) are generated before they are combined into the space of potential design-points. These design-points are evaluated through performance estimation and pruned to form the Pareto-optimal function of cost versus performance. The Fused Instructions (ISEs) differ from those generated in approaches already covered in that DFG are considered with loop-dependency information, which innately reduces the search space from that considered by most AISE algorithms. Combining the loop information into the architecture customisation space means that the DFG are not blown up in size by transformations such

as unrolling which are generally required to obtain good results with other AISE algorithms.

Customisation options are generally abstracted as *resources*, with each resource representing a microarchitectural component such as a functional or other hardware unit. Resources are organised into classes like an OO language, with some base classes representing groups of standard operations and new classes used to represent ISE. Resources are parameterised with additional properties such as the vector length they have to process (hence duplicating the operation a number of times), and the element size that they operate on (hence optimising the data-path for a specific width). ISEs can be made vectorised by defining a fusion of operations as a new class, and then parameterising it with a vector size greater than one. This allows for smaller ISEs to be expanded across loop iterations without loop transformations, and is part of the reason why the Tensilica approach favours a large volume of smaller ISEs. Hardware resources are shared between ISEs as much as possible, further improving the quality of the designs produced. A further optimisation is the extension of the memory interface with arbitrary width memory operations. TIE therefore has a coverage of extension greater than most other techniques, most of which (e.g. [95], [7], [20]) class memory operations as barriers to ISE identification. Memory operations are not fused to other operations, but their extension and customisation does underly a significant portion of the impressive performance obtained by XPRES, since memory operations represent a large portion of the unapproachable barrier in other AISE techniques.

The efficacy of the Tensilica XPRES approach therefore motivates the automatic DSE of processor (architecture and microarchitecture) features other than covered by scalar ISE, due to the efficacy of this technique in addressing a large range of applications.

3.2.7 Other Algorithms

The ISE problem has been studied by a great many people, with a number of relevant formulations having been published between the early nineties and the present day. This section has attempted to address a cross-section of the broad spectrum of algorithmic approaches for the fine-grained identification and selection problems. There exist other notable works which deserve at least a brief mention:

- An attempt to analyse the limits on ISE acceleration performance is performed in [98], addressing the question of how much mapping just data-flow to ISE (and leaving control flow to the baseline core) can accelerate applications. Unrolling and other OLP-enhancing transformations are employed, with the result that in general a maximum of 6x acceleration is generally achievable under absolutely perfect circumstances.

- The separation of instruction-set extension algorithms into those that solve the problem *exactly* (e.g. locate the optimum ISE) and those that solve the problem *approximately* (e.g. attempt to locate a good ISE without searching the entire space) is covered in

[21]. Various algorithms are proposed and evaluated, and the intractability of the exact approach is demonstrated.

- The encoding of custom instructions is considered as the main criteria for selection from a set of candidates in [99]. The EXPRESSION [86] framework is utilised to compare a heuristic method of encoding-based selection versus an ILP formulation of the same problem. The heuristic method performs much faster than the ILP formulation, and produces results of roughly equal merit. The authors of [99] note that the variety of encodings employed may increase the complexity and hence critical path of the decoding unit, although they claim that since they replace existing complex instructions it is a like-for-like trade-off. This claim is not empirically proven, so it is hard to determine exactly what the trade-off looks like.

- The question of whether or not architectures designed to exploit operation-level parallelism such as VLIW can actually benefit from ISE also designed to exploit the same is addressed in [100]. The results demonstrate that VLIW issue width and register file size can both be reduced where AFU are employed, and that AFU have considerable use to accelerate an application even in the face of OLP exploiting architectures such as VLIW.

- A fairly standard processor extension approach utilising rapid prototyping is taken in [101], with the interesting difference that the evaluation is performed for Prolog benchmarks amongst others. Most ISE approaches use only small C benchmarks to evaluate their approach.

- Customising a processor for not just one but many similar applications is a valid real-world problem which when solved can increase the profit margins for a particular ASIP. The work of [102] attempts to address the acceleration of whole application domains using AISE. The work also presents methods of generalising AFU hardware to cover a wider array of subgraphs, including wild-carding which causes FU in AFU to cover a wider range of operations than was originally identified. AFU in [102] are multi-cycle and do not contain registers for pipelining, but instead make extensive use of serial gains from combining bitwise operators. Results are between 1.4x and 2.4x at maximum resource usage. The AISE identification algorithms and infrastructure used in [102] were originally covered in [103].

- A divide-and-conquer approach utilising a variant of the popular A* algorithm called "Divide and Conquer A*" (DCA*) is used in [104] to perform a heuristic-based partitioning of a DFG into ISE and software components. Like ISEGEN [7], the algorithm may be parameterised with an arbitrary heuristic for search guidance. Strangely the work of [104] chooses not to quantify application performance enhancement from the extension generated, rather examining code size which is reduced by between 5% and 25%

depending on the number of templates utilised. The algorithm is compared against A* with an equivalent heuristic, and the DCA* algorithm performs considerably better.

- Many AISE approaches (e.g. [7, 20, 21] use simple models of speedup, (see section 2.3.1) whereas the work of [105] attempts to guide the search for ISEs using a cycle-accurate simulator and is effectively a semi-automated iterative refinement technique similar to the Compiler-in-Loop [26] technique popularised by LISA [29] and CoSy [28]. The work of [105] introduces another ADL; the Unified Processor Specification Language (UPSLA), which is claimed to be descriptive enough to model even complex architectures such as the PowerPC 604. Despite the recent (2004) publication of the work [105], ISEs explored are only "Instruction Pairs", i.e. the coalescing of only two instructions into an ISE. This approach yields a best-case acceleration of 28.5%, and code-size reduction of 12.8%, which is interesting as a benchmark for minimalist ISE design. It should be noted that this was achieved using encryption benchmarks, which are commonly found to be the best accelerated benchmarks in AISE studies.

- A general, single formulation of identification and selection is attempted in [106], but the use of the guide and cost functions is not a good fit with many algorithms. A lot of algorithms (e.g. ISEGEN [7], Atasu ILP [20]) tend to use a single merit function and are not divided between identification and selection in the manner described (full enumeration & pruning). There is still scope for a lot of innovation in the instruction set extension problem, and the approaches currently attempted do not entirely encapsulate the potential for the future. This author feels that such a generalisation of the problem only helps in abstract understanding, not in actual algorithm design or in understanding existing algorithms. Whilst the attempt at generalising ISE algorithm design is not entirely useful, the rest of the paper is a good (albeit somewhat incomplete) comparison of techniques not covered here.

One of the main contributions in ISE research not extensively covered here is the higher-level block- or loop-level graph processing. In this thesis the author has decided that a finer-grained approach is necessary in order to properly reap the benefits of reconfigurable technology. Coarser-grained blocks are generally harder to balance for resource utilisation and sharing. The reason is that when you select larger blocks, you have less control over the data-flow within those blocks which for best results should be implemented using resource-shared microarchitecture. A round-up of some of these coarser techniques exists [106], but this is also incomplete. The field of AISE has become so populated in the last ten years that it is nearly impossible for a single study to cover them all; a Google search for "Instruction Set Extension Algorithm filetype:pdf" returns "about 94,900" results at the time of writing. What has been presented here should represent a suitable sampling from this massive space, and in particular includes the most popular cited works.

3.3 Microarchitectural Solutions

3.3.1 Field Programmable Gate Arrays

Several vendors provide reconfigurable hardware which falls under the class of microarchitecture called Field Programmable Gate Arrays (FPGA), which replaced many other programmable logic devices in many consumer and prototype applications. These are perhaps the most popular but misunderstood form of reconfigurable microarchitecture, generally owing to their continued development towards heterogeneity in the cells present in the hardware. At first, the only FPGA cell was generally only a K-bit look-up table implemented in SRAM, in which the table was directly programmed via the bitstream with the logic values to be induced under each of the 2^K possible input values. Configurable routing is combined with this to form the entire space covered by the bitstream used to program the FPGA devices.

Xilinx [107] were the first company to commercially exploit FPGA, providing the first model (XC2064) as early as 1985. This early design contained only 64 cells each containing two 3-bit look-up tables, which was equivalent to fewer gates than most competing programmable logic devices at the time; two years later Xilinx had increased the gate equivalence to 9000, which was competitive. Modern designs by comparison have the equivalent of many millions of gates, in order to keep the FPGA devices competitive in terms of scale with other ASIC and ASIP approaches available. More recent developments involving coarser-grained programmable blocks are perhaps the most misunderstood aspect of FPGA. Units such as multipliers, floating point units, RAM and ROM are integrated directly into the programmable fabric of the FPGA in order to increase the performance of those functions.

A range of different FPGA are available to buy depending on what domain (e.g. DSP, Cryptography) the fabric is intended to be programmed for. This approach was probably motivated by early work [108] which demonstrated that creating an array of different FPGA topologies increased the efficiency of FPGA with regards to particular applications. Earlier still, work [109] had concluded that using heterogeneous combinations of LUT size also improves performance. The introduction of hardwired logic into FPGA to provide improved performance from coarse-grained configurability is a logical continuation in this trend of heterogeneity. In this manner the line between FPGA and reconfigurable ASIP systems is forever being blurred, especially with the introduction of full processor cores into the reconfigurable fabric. A range of interconnects is possible between these integrated processor cores and the reconfigurable fabric, covering the full spectrum from coprocessor to tight coupling described earlier in section 2.3.5.

Instruction set extension is possible, and the performance benefits have been the subject of academic interest for some time, e.g. in [110]. FPGA technology has a well-deserved reputation for high cost and energy consumption, with relatively low performance when compared to an ASIC solution. This is because they represent the most flexible of the reconfigurable archi-

tectures, and this reconfigurability must come at a cost. It is unlikely that even with the advance of coarser-grained FPGA devices that these will ever reach the same levels of performance that standard-cell design can achieve. The flexibility and re-programmability does though afford the devices significant advantages, versus the "single-shot" approach of ASIC fabrication. Once an ASIC design is produced, errors discovered cannot be fixed without a re-spin of the design (at significant cost). This is likely to drive the continuing adoption of FPGA in prototyping, low-volume systems, and anywhere else that the cost of an erroneous ASIC design would be greater than the increased cost of FPGA.

3.3.2 MOLEN

The combination of a hardwired processor and reconfigurable FPGA fabric does not necessarily define the resulting architecture absolutely, as the various mechanisms for combining the two are subject to a degree of design decision as per section 2.3.5. One architecture intended to trade off the various strengths and weaknesses of the different communication mechanisms is the MOLEN processor [111]. The MOLEN processor needs to utilise only four additional instructions in order to facilitate the computation of sections of application code on the FPGA fabric, which is essentially treated as a co-processor.

The additional instructions are not associated with any particular application-specific function, but are instead used to configure (*c-set*) and initiate (*execute*) application-specific functions through the instantiation of microcode. Microcode controls the reconfigurable processor (RP), in addition to the movement of data to and from the GPP core and the RP (*movtx* and *movfx*). This constant space of additional instructions avoids the apparent "opcode explosion" encountered in some works utilising ISEs on a one-to-one basis between extensions and functions. The prototype produced in [111] utilises a Xilinx Virtex-II Pro FPGA, using the embedded PowerPC core as the GPP and the rest of the FPGA as the RP.

An arbiter exists in the FPGA fabric also, which performs an initial fetch and decode on each instruction from the stream before determining whether to send it to either the GPP or RP. When the RP is active the arbiter sends instructions to the PowerPC to put it into a wait state, whilst the arbiter feeds the appropriate microcode signals to the RP to execute the mapped function. The major problem with this approach is that the per-operation speed on the original GPP is greater than that on the RP (FPGA), in addition to the overhead in communicating the values via extension registers to the reconfigurable fabric. The implication of this is that sections mapped to the RP must be both large and contain a high degree of OLP, in order to amortize the overhead from running code on the RP.

Code to be mapped to the RP is not automatically identified, but instead the support software relies on pragma's annotated on the original (C) source code to perform the mapping. As per other synthesis techniques, only a subset of the language may actually be mapped to the RP, but this includes some memory operations so the fragmentation often caused by these

are not a major factor. In the latest work on MOLEN [112] the prototype architecture obtains between 1.56x and 3.18x speedup (over the FPGA-embedded PowerPC) on MPEG2 encode and decode. These speedups were obtained with considerable manual effort; first annotating the program, and then constructing custom microcode for the RP.

Reconfigurable processor SoCs organised in a very similar fashion include GARP [113], NAPA [114], and PipeRench [115] amongst others.

3.3.3 Custard

Soft processors attempt to combine the flexibility of FPGA with the more productive programming model used for standard microprocessors, through instantiating an ASIP within the FPGA fabric (not using a hardwired core but the FPGA itself). The CUStomisable Threaded ARchitecture (CUSTARD) [116] is an example of such an approach. CUSTARD has a CoSy [28] compiler allowing for C code to target any of the potential instantiations of the soft-processor when implemented on an FPGA. The multi-threading is implemented in a similar way to Intel's hyper-threading; in effect the register file is duplicated M times for M threads, allowing for rapid interleaving between different contexts without copying the contents of the register file to and from memory. The CUSTARD prototype allows for M to be any power of two, and additionally allows the register width, file I/O ports, and number of registers to be customised within encoding limitations. Certain forwarding paths may also be enabled or disabled based upon the threading configuration used, further optimising the area (and delay) of the instantiated microarchitecture. The architecture itself may be extended with ISEs, which the CUSTARD compiler can both identify and exploit. The exact means of ISE identification is not covered in the literature [116], other than to state it is based on static analysis. Additional custom FU's are added to the pipeline (tightly coupled) alongside the ALU of the original architecture and within the forwarding paths of the processor.

The use of AISE contributes a maximum of 3.55x cycle reduction in the case of the AES benchmark, but the different configurations vary considerably in terms of maximum clock speed reported by the FPGA tool-chain. There is no discernible correlation between features utilised and the resulting clock delay, as in some cases the use of more complex features seems to result in a lower critical path. This is likely a result of the non-linear and non-optimal mapping algorithms used in the FPGA tool-chain; the maximum clock speed varies significantly, from 19MHz to 30MHz in the technology used in [116]. Interestingly, the use of ISE adds practically nothing to the area (slices) utilised by the design; across all cases covered in [116] the difference between the baseline configuration and the configuration utilising ISE is on average 3%. The custom instructions identified are both few and largely comprised of table look-ups, which are not the standard "arithmetic" type operations one would expect of an AISE approach. It is quite odd to see such an advantage gleaned from such a small number of simple ISEs, but this is likely due to the coverage of constant variables with look-up tables.

3.3.4 ADRES

Citing the greater performance of coarser-grained reconfigurable architectures than is achievable using standard FPGA microarchitecture, the ADRES architecture [117] combines VLIW and a reconfigurable matrix (RM) in tightly-coupled combination. The major motivation for this is that word-level configurable units are more readily optimised for their ultimate function at the time of synthesising the hardware, versus the bit-level reconfigurability afforded by most FPGA. Flexibility is lost in this trade-off, however, it is considered that since the applications intended to be run on this architecture are largely word-level also, the additional flexibility lost is not actually generally useful. The hottest (most frequently executed) sections of code are mapped to the RM, with the VLIW maintaining the control-flow and overall function of the remainder of the application.

The paper of [117] presents not only the architecture, but also the tool-chain which targets it. Despite ADRES being tightly coupled via one of several register files, the architecture is referred to as having a processor-coprocessor model. The nomenclature is due to the additional register files present in the RM, and because the RM can utilise the VLIW memory channels to directly access memory without first loading it into the shared register file. Resource sharing is applied between the baseline VLIW core and the RM, in that some of the reconfigurable cells (RC) share their functional units with the VLIW scalar instructions. The VLIW core and RM do not operate concurrently, which disrupts the idea that the RM is a co-processor in any standard definition; in addition the RM does not have standard control flow operations, which may only be executed through the VLIW core. Predication is present in each RC to enable some degree of control-flow in the RM. Multiplexors controlled by configuration memory determine the flow of data between different RC in the RM, the RC having been set to a particular operation by the same configuration memory. The memory hierarchy is shared between both the VLIW core and the RM, since the memory channels are resource-shared between the two components.

Evaluation in [117] is performed by creating a microarchitectural design instance of the ADRES architecture, which in itself is a template for a class of designs and not a specific design. The design chosen is similar to the MorphoSys [118] reconfigurable architecture. For a small selection of four kernels, speedup observed is between 2.8x and 6.4x. Acceleration is obtained by combination of streaming, pipelined loop iterations, and operator-level parallelism. The results obtained appear to be of roughly the same order of magnitude as standard ISE obtains, with a slightly higher worst-case due to the inclusion and exploitation of loop-level pipeline parallelism and streaming of operands outwith the register file.

3.3.5 Annabelle and Montium: Chameleon

Streaming applications cover a wide range of domains, enough so that designing architectures specifically with streaming in mind can impart benefit to an array of potential applications.

Annabelle is one such architecture, constituting a heterogeneous tiled architecture built around a Network-on-Chip, which acts as the main interconnect between the various tiles. One tile processor (TP) developed for this architecture is Montium, a domain specific reconfigurable core. The combination of Anabelle with four tiles containing Montium TP is called Chameleon, and is covered in [119]. The Chameleon architecture is intended to address a wide array of streaming DSP applications from software radio to image processing, through reconfiguration of the Montium tiles on a per-application basis. Each Montium tile contains five ALU with four inputs and two outputs, each having individual register files allowing for state to be maintained between operations. In addition there are ten data memories which operands to the ALU can be read from and written to, and a routing network allowing for communication between the ALU, Memories, and the NoC interface. With four Montium TP in the Chameleon architecture, there are a total of twenty ALU available to be used in a reconfigurable fashion when performing stream processing and DSP. An ARM 9 processor sits on an AMBA bus (connected to the NoC) in the Annabelle architecture to facilitate general control flow and the addition of further ASIC components which accelerate specific functions such as Viterbi decoding. Only the TP are connected to the NoC directly; other components such as the ASIC and memory channels (including DMA) are on the AMBA bus. Each Montium TP constitutes about 1.8mm^2 in a 130nm process, leading to a fairly high area requirement compared to some ISE-based ASIP such as XTENSA [96] or EnCore [10]. ALU only support integer or fixed-point arithmetic, as these are the data types used in most DSP algorithms implemented for embedded devices. The Montium and hence Chameleon architectures place a premium on memory transfers, citing their massive contribution to power consumption when these climb the memory hierarchy and make their way off-chip. This is the main reason why the Chameleon architecture contains so much on-chip memory, as it is assumed the locality of most streaming DSP applications will have enough temporal and spatial locality to exploit such memory. Since the ALU contain no pipeline registers, the critical path of the TP and hence the whole Chameleon architecture is configuration-sensitive, and fell between 140MHz for an FIR filter and 100MHz for an FFT. Power consumption for an FFT butterfly was noted to be of the same order of magnitude for a single Montium TP and an ASIC implementation of the function. FPGA implementations of the same function were between 13x and 20x higher power consumption, largely due to the overhead incurred in implementing the word-level operations in a bit-level fabric. Using the ARM 9 processor alone the power consumption was around 10x higher. The power consumption of the full Chameleon architecture will naturally be higher once the whole system is considered, but this was not directly examined in [119].

3.3.6 QuickSilver Adaptive Computing Machine

The QuickSilver Adaptive Computing Machine (ACM) [120] is the first commercially available "Fractal Architecture", so called due to the topology of the chip routing layout. QuickSilver is

comprised of clusters, with the smallest being a four node cluster containing an arithmetic unit, bit manipulation unit, finite state machine unit, and scalar unit. These four units are arranged around a central Matrix INterconnect unit (MIN) which connects the heterogeneous units to one another and to a parent node. The next cluster size is sixteen nodes, and this is where the fractal layout comes into play: four sets of four clusters are arranged in a topologically identical fashion to how the original four nodes are arranged in a cluster, leading to a total of five MIN components in the sixteen-node cluster. This arrangement can be continued up in powers of four, hence describing the architecture as fractal because it is topologically self-similar, and the root (middle) node of such an arrangement is used for communication with the ACM. The combination of all MIN are essentially a Network-on-Chip, allowing for data to be routed between the various nodes in the ACM. The mapping of software to hardware is performed through an augmented dialect of C called "SilverWare" [121]; the augmentations include spatial and temporal extensions to assist in the mapping. Ultimately the entire application is written using SilverWare, and the tool-chain is used to create a mapping (which may utilise dynamic reconfiguration) which may instantiate and drive the various algorithmic elements on nodes in the ACM. Each node is a fairly large unit, intended to cover a whole kernel. Every node is largely comprised of local memory, constituting around 75% of each node by area [121]. Arithmetic units are similar to the Montium TP [119], in that they contain a number of parallel ALU, essentially able to implement whole DSP-like functions such as FIR and FFT spatially. Bit manipulation units are intended for functions such as code generation, packet discrimination, and linear feedback shift registers. State machine nodes can implement arbitrary state machines as required for protocol implementation. Scalar nodes are essentially scalar microprocessors, allowing for any legacy C-code to be executed within the ACM where this may not be effectively mapped to any other node in the device. All nodes allow for variable-width data to be used. Multiple nodes of the same type can be used to implement a function or state machine which cannot fit into a single node, and the same function or state machine can be time-sliced on a lesser number of nodes if resources demand it.

3.3.7 XTENSA

At the time of writing, Tensilica have produced a vast swathe of pre-customised version of their XTensa CPUs [96], which they now choose to refer to as Data-plane Processing Units (DPU) due to their data-flow centric customisation. As covered earlier when studying the XPRES extension mechanism for the customisable variants of the Tensilica CPUs, Tensilica allow for customisation in three axes: VLIW, Vector Operations, and ISE. The cores reflect this customisability, allowing for all of these configurations and extensions to be rapidly applied to the XTensa processor to produce a suitable ASIP. The logical separation between "configuration" and "extension" here is that configuration covers the inclusion, exclusion, or parameterisation of an existing CPU component, whereas extension involves the construction of a new CPU

component such as ISEs specified in the TIE language. Configuration options include:

- Standard scalar functional units such as multipliers and FPU.
- Cache sizes.
- Use of VLIW and Number of VLIW ways.
- Zero-Overhead-Looping (ZOL).
- Number of pipeline stages (5 or 7).
- Various DSP engines (Vectra, HiFi2, ConnX) acting as coprocessors.
- Number of memory channels (Load/Store units).
- Memory Management Unit.

Extension options are fewer, and include the ISEs specified in TIE, additional register files, and I/O interfaces. The major distinction between a configuration and an extension is that the latter includes some structural information, whereas the former only requires a finite number of parameters to specify for each option. Earlier it was cited that the three customisation axes are largely orthogonal, meaning that they *can* be used to complement one another resulting in efficacy in the order of the product of their individual performances. When including the additional features above the space overlaps considerably, and the onus is on the designer to select the correct set of additional functionality over the baseline core that will best accelerate his application. All of these features are of course covered in the simulators, but some of the above options (e.g. the DSP engines) are not covered in the XPRES analysis and so require manual comparison between the XPRES-generated design points and points including these more complex computation engines. When this thesis was written Tensilica are offering two main microarchitectures, which are similar but distinct in their potential for customisation: The XTensa LX3 [122] and the XTensa 8 [123]. The latter is a more lightweight relative of the former, and does not include quite as many potential customisations. Configuration features exclusive to the XTensa LX3 include pipeline depth, the various DSP engines, VLIW, and variable numbers of memory channels. The XTensa 8 is purposely kept lightweight to occupy the low-power low-cost end of the spectrum, whereas the LX3 covers higher performance areas of the trade-off space; the two overlap in terms of their potential configurations and where this is the case the 8 will outperform the LX3. Tensilica represent the current state-of-the-art in terms of processor customisation and are likely to do so until other manufacturers include the same degree of flexibility in their architectures, without compromising performance through excessive generality as with FPGA. Tensilica's offerings have found use in various consumer embedded electronics, but also in supercomputer design due to their exceptional performance advantages in application-specific roles. The Lawrence Berkeley National Laboratory recently

proposed an XTensa-based supercomputer for computing weather dynamics [124], obtaining over ten petaflops (10,000,000,000,000,000 flops) using 3.84 million cores. The construction costs and energy consumption are both orders of magnitude (10x or more) less than comparable machines using Intel or AMD GPP, demonstrating that customisable cores have worthwhile application outside of embedded devices.

3.3.8 Stretch

The XTensa microarchitecture is open to extension outwith the options advertised by Tensilica, and some companies have chosen to partner up with Tensilica in order to develop more diverse ASIP solutions than those currently available. The XTensa DPU do not natively contain instruction set reconfigurability, and so the Stretch company has developed a technology dubbed the Instruction Set Extension Fabric (ISEF) in order to allow dynamically defined data-flow computation to be covered by single instructions. The latest in this series is the Stretch S6000 [17] SoC, the main processor of which is essentially the XTensa LX processor [122] core with a "Second Generation" ISEF tightly coupled into the dual issue VLIW pipeline. The ISEF itself consists of 4096 ALU capable of 2x4 bit standard arithmetic operations including multiplication; these can be chained up in order to perform operations of arbitrary bit widths. In addition there are 64 8x16 bit multipliers which may also be chained into larger widths, 64KB of single-cycle-memory "IRAM" split into 2KB chunks, a DMA channel to load the memory-mapped IRAM, and multiplexors and shifters to transfer data around the ISEF. The intention is that an entire loop kernel body can be mapped onto the ISEF including memory accesses, but the ISEF is still fed by a 32-element 128-bit register file (which may be sub-element partitioned arbitrarily). The IRAM therefore should be used to store intermediate values, stream data (such as a single macro-block of pixel values), filter coefficients, and various constants; all of these may be fed to the unit via DMA hence freeing up the processor for other operations whilst the ISEF is communicating with memory. It is not stated in the literature how the DMA "frees up" the main CPU, but it is likely that this just means that the ISEF can be calculating one iteration whilst the values are being loaded for the next. VLIW implies that there is not out-of-order execution allowed, and tight coupling implies that the ISEF will not operate independently from the main pipeline. The S6000 is intended for Audio and Video applications, and a programmable accelerator is provided in combination with the ISEF-extended XTensa core as a co-processor. The programmable accelerator provides a range of additional high speed functions including the Tensilica HiFi2 audio engine [96], plus Encryption, Entropy Encoding, and Motion Estimation accelerators. This range of application specific functionality present in the S6000 SoC allows it to encode a single high definition h264 stream or four standard definition streams in real time at 345MHz. The real strength of the architecture is in providing an off-the-shelf solution for media streaming using modern codecs, demonstrating the efficacy of partially reconfigurable ASIP when addressing a whole domain of applications.

3.3.9 Other Microarchitectures

- The PACT eXtreme Processing Platform [125] (XPP) which constitutes a coarse-grained reconfigurable accelerator is combined with the static data-path LEON processor in [126]. The XPP is normally used in a loosely coupled fashion, but in [126] it is used to provide a reconfigurable instruction set extension microarchitecture by tightly coupling it to the LEON data-path. The result is somewhat similar to the Stretch [17] devices.

- The RAW Processor [127] is a tiled multiprocessor in which the interconnect is directly exposed to every contained processor via the RAW ISA. Any CPU in the RAW Processor may reconfigure and utilise the communication network via this ISA, to facilitate the flow of information between cores. Network access is a first-class member of the instruction set, and can be used as a source or destination for an instruction instead of a register. The RAW Processor is intended to be a proof of concept solution to the problem of increasing chip resources and wire delay; the tiled chip and network combination is infinitely scalable in terms of the wire delay as the longest link is the distance between two adjacent cores. In addition to the dynamic network, parts of the RAW processor's cores can be statically connected via a static network in a manner similar to FPGA place and route in order to create software circuits.

- Kress arrays [128] are a generalisation of systolic arrays, sometimes referred to as reconfigurable data-path units (rDPU). Communication between elements in an rDPU is done between nearest neighbour, The number and type of PE is determined at fabrication time; the routing and selection of operations is done at configuration time. For example, the work of [128] has an rDPU which contains every integer and bitwise operation accessible in C on every PE. Kress arrays are sometimes referred to as "anti-computers" due to their lack of control flow, but this is perhaps a misnomer as all existing implementations rely on a host core for control signals.

Of course the contents of this review cannot be all-inclusive; almost every institution or company which has approached the issue of next-generation reconfigurable or application-specific microarchitecture has generated a new design. This review is intended to cover a sub-sampling of the space of explored options to demonstrate the commonality and diversity simultaneously present in the spectrum of designs.

3.4 This Work In Context

The umbrella hypothesis for this thesis is that: "The efficacy of ISE can be increased by improving the microarchitecture, identification algorithm, and software form". In particular we are looking to reduce the cost-benefit ratio when using ISE with respect to engineering time, acceleration, and energy.

3.4.1 The Need For Predictable Microarchitecture Cost and Benefit

In order to allow an algorithm to explore trade-offs in a design which will ultimately undergo synthesis, some generalisations must be made. A good algorithm to explore the space of ISE designs must have some model of the costs it is trying to minimise, be that the time taken to run the application, the area of ISE implementation in silicon, or the energy expended to run the application. Existing academic ISE algorithms discussed in this chapter [88, 91, 18, 20, 89, 90, 92, 95] do not have any way to predict the costs of their resulting implementation as they have no model of the microarchitectural implementation. The Configurable Flow Accelerator introduced later is an attempt to provide an easily modelled microarchitecture, the cost(s) of which may be integrated into the algorithm identifying ISEs as in later section 5.5. The CFA is an explicitly-microcoded and instructed variant of the CCA discussed in section 2.3.5. The CCA identifies common sequences of instruction to implement on the fly as a single new instruction, however due to the instruction stream being a lowered form free of much dependency information it cannot achieve the same degree of efficacy as a static approach. This is the reason for the differences between the CCA and the CFA. Moreover the additional hardware required for a CCA to dynamically identify, store, and replace instructions on the fly is not required with a CFA.

3.4.2 Reducing Engineering Time

Prior work regarding the construction of algorithms for AISE have taken note of the trade-off between the run-time of the algorithm and the quality of the result produced. The least (linearly) complex algorithms such as MaxMISO [88] and MaxMIMO [91] are cheap in terms of runtime but the results produced are not good: exploration is performed in terms of false constraints put upon the search space in order to obtain a lower run-time. A similar problem is true of the more complex clustering approach of Verma *et al.*[18]: the requirement for the merit function to be monotonic with respect to the number of DFG nodes in a cut is only true when you consider the problem definition and not the reality of the intended use of the result. Complexities arising from the implementation of larger ISEs (such as routing, or other non-linear effects not modeled by ISE algorithms) contribute directly to undermining the hypotheses embodied by these algorithms. This work instead looks to take an already-proven algorithm [7] based on the guidance of heuristics representing realistic engineering concerns, and looks to improve its runtime by further making it aware of pathological conditions in its state that would normally cause fruitless search. Taking a good algorithm and improving its runtime without forcing additional constraints upon it that might otherwise disrupt the quality of its result.

3.4.3 Reducing Area

Having introduced a new microarchitecture for implementation of ISE, and having that microarchitecture explicitly exploit inter-ISE (spatial) resource-sharing by default, further reductions in area via resource sharing are still desirable. The original CCA work [12, 13, 14, 15, 16] explored inter-ise sharing explicitly, as this was what the unit was designed for. It was not shown that any attention was given to resource sharing intra-ISE, which could further allow for a reduction in area without increasing runtime. Such efforts would add considerable complexity to the CCA dynamic scheduling hardware, and would not reduce the size of the CCA as much as increase the size of ISE possible to execute on it. With the CFA static-analysis approach, we are able to focus on application-specific optimisation: optimising the area for the statically induced ISEs of a known application. The later section 4.3 looks at this trade-off and presents an algorithm for it.

3.4.4 Improving Acceleration

The original paper on the ISEGEN algorithm [7] gives a single static weighting vector to calibrate the heuristic used to guide the search. Ad-hoc observations made during the initial implementation of this algorithm demonstrated that this was not in fact the case, motivating further examination of the vector. We therefore perform a parameter-sweep exploration in section 5.2 to locate a better static vector to provide a baseline in other work performed here. This provides both an accurate measure of the efficacy of the original algorithm, and a methodology for further experiments to calibrate and evaluate new ISEGEN heuristics, as in sections 5.4 and **??**.

Pipelining has been explored in other work with regards to ISE identification [18, **?**, 129], but it has been with regards to the pipeline scheduling of inputs and outputs for a single large ISE rather than the identification of multiple ISEs which are able to be scheduled in an overlapping fashion. These works are interested in trying to make sections covered by ISE as large as possible, hypothesising that larger ISEs are better for runtime. Whilst this "bigger is better" mentality is occasionally true for acceleration, there are again many occasions where it is too costly to pursue with regards to other constraints such as area, and energy. Previous efforts also involve increasing the effort required by a rather massive degree due to increasing the size of the problem being processed by AISE. For example in one attempt to schedule I/O over multiple cycles **??** the AISE algorithm had to be run repeatedly, once for each I/O constraint. Here we hypothesise that rather than identifying single large single ISEs, multiple smaller ISEs can instead be identified to increase the acceleration available at a given I/O constraint. Section 5.4 explores this hypothesis.

3.4.5 Reducing Energy Requirements

Exploration of low-power ASIP microarchitecture ([130], [27]) has been a subject of academic and commercial interest for several years. In [131] the energy saving effects with regards to the register file are examined when forwarding is employed, and up to 25% is found to be saved. More directly relevant work includes [132], which was performed for FPGA and soft cores. Whilst the analysis of the energy effects in FPGA fabric is useful, the question of what effects ISE has when using standard cell libraries is also an important one. The energy effects of combinational ISE in a standard cell technology are modeled in [133], and the work examines energy savings in utilising a few (up to seven) large combinational ISEs. The ISEs produced [133] are for some reason extremely low power in comparison to the host core (supposedly less than 1% of the execute stage alone), which is claimed to be due to the simplicity of the AFU circuits in comparison to the synchronous core (described only as "ARM-like"). No absolute energy measurements are made in [133], preferring instead to cite the factor of change.

Energy estimation for extensible processors is explored in [134], but the approach is a statistical one which must be trained. As mentioned in [133], where RTL models are available as opposed to the C models used in [134] a statistical approach is not necessary as the hardware may be synthesised and analysed directly. Both [133] and [134] are concerned with building power models based on sub-components of ISE microarchitecture, as opposed to the approach taken here in section **??**. The work in this thesis directly measures the power and energy performance of the whole system in order to evaluate the energy performance of the CFA microarchitecture. Here a new energy heuristic is integrated directly into the ISE identification algorithm, in order to identify ISE targeting energy reduction as well as acceleration.

3.4.6 Software and Hardware: Chicken and Egg

When we are designing application-specific hardware directly based on the structure of application software, manipulating the structure of the software becomes akin to manipulating the structure of the hardware. As with hardware, a single function in software might have multiple possible realisations internally whilst still maintaining the same external behaviour.

Compiler transformations as discussed in section 2.6 are automated means of manipulating the source code of an application, which in the context of AISE would also mean changing the ISEs generated. Some early work was performed [37] to determine the effects of if-conversion and loop-unrolling transformations. The work presented here in section 6.2 formalises the number and sequence of transformations as a space to be explored, and performs a large-scale sampling of said space. With a single starting source, many thousands of alternative versions are produced in order to determine the effect that source transformation may have.

The use of floating or fixed point in a particular design is also a matter of transformation, and since the trade-off has not been studied with regards to ISE, we do so here in section

6.3. The experiment is to determine whether ISE is disruptive to the traditionally held rules of thumb regarding the choice of number format.

3.5 Summary

This chapter has taken a cross-section of the state of the art in processor DSE and the microarchitectural basis upon which this can take place. Both academic and industrial offerings have been covered, demonstrating both the intellectual and commercial interest present in this field.

This thesis now moves on to the introduction of a new microarchitectural solution and design methodology, to be used in the remainder of this thesis where appropriate.

4 THE REAL WORLD: ENABLING AND OPTIMISING HARDWARE SYNTHESIS

"Reality is merely an illusion, albeit a very persistent one."

– Albert Einstein

This chapter introduces the CFA microarchitecture, and a mechanism called "CFA Staggering" for improving the cost-benefit of CFA designs through temporal partitioning. *The CFA is demonstrated to be a cost-effective design for ISE implementation. Staggering is demonstrated on average to reduce the area of CFA implementation by 37% for only an 8% reduction in acceleration.*

4.1 Introduction

As the previous chapters have covered, there are a very large variety of design options which may be combined in various ways to produce a near-infinite number of design points. Automatic DSE in this space is therefore currently prone to a considerable amount of ad-hoc pruning, largely by throwing out large classes of microarchitecture entirely from the exploration process in favour of a smaller subset of options. High-level constraint decisions such as which classes of microarchitecture to include, cost limits, and the application itself are the major top-level inputs of an engineer to DSE [26]. The DSE process is able to instantiate a particular instance of a design, synthesise it, and retrieve important statistics which are then used to determine the merit of that design [97]. For this reason, in order to perform DSE for the work in this thesis, both the exploration process and the domain over which exploration is performed must be defined in such a way as to allow for this to proceed without human intervention. For our purposes, ISE is just a restricted form of DSE concerned only with the construction of a suitable set of ISEs in the proposed microarchitecture. The aim of all the work contained in this thesis is to make the process of ISE more efficacious, and this has to start with the choice of microarchitecture.

The "Combinational Flow Accelerator" (CFA) microarchitecture introduced here has several differences to the unclocked combinational logic implied by many ISE approaches. Inter-ALU connections in a CFA are made at clock boundaries (referred to as echelons), rather than in an ad-hoc unclocked fashion. This means that no multi-cycle latency ever exists, as all paths within a CFA are single-cycle which is easier to synthesise and verify. Even in work such as [23] where pipelining is used to mitigate I/O constraints, the pipelining effort is targeted at increasing ISE I/O and so generating larger ISEs. Here, we are interested in an architecture that is able to obtain comparable throughput with smaller and hence cheaper ISEs. Reconfigurability

is a core component of the CFA, although in this thesis reconfigurability is evaluated only as a mechanism for implementing resource-sharing. Previous work performed [102] for the combinational ancestor of the CFA, the "Configurable Compute Accelerator" (CCA) has shown that a selection process similar to that used herein is readily adaptable to perform domain-specific acceleration rather than application-specific. The CFA is aimed to build upon an array of prior work including the CCA and the ISEGEN algorithm [16, 14, 15, 102, 7], as has been covered in previous chapters. Having the potential for domain-specific acceleration already proved for CCA, we investigate further methods of making ISE more cost-effective and generally efficacious via CFA.

Whilst the CFA has not been specifically studied before, it is a worthwhile candidate for use in the remainder of the research so long as it has cost within an acceptable range of the baseline core that it is intended to extend [10], and is comparable to acceleration expected of ISEGEN and other ISE approaches [7][20][21]. Performance optimisation for embedded processors is subject to an exponential decay in the return on area when using commercial ISE [97]. We therefore confirm in section 4.2, that the CFA also exhibits this trend, as a due diligence concerning the viability of the technology in the real world. Commercial and industrial products require further costs to be considered than just the gate area consumed. With the ever-increasing demand for massively multifunctional mobile devices such as mobile phones, power and energy consumption are now also first-class considerations. The effects of CFA-based ISE on power and energy are hereafter evaluated, in order to demonstrate that the CFA is efficacious in those domains. As per area, the power and energy consumption should not dwarf the baseline or this approach can never be taken seriously in a realistic context. Observations made in this first empirical section are used to guide later efforts in improving the efficacy of the CFA automated design process.

The echelon-based structure within the CFA enables a particularly straightforward process of temporal resource sharing; sharing hardware resources between operations at a different time in the same ISE, as opposed to between different ISEs. Temporal resource sharing is sometimes referred to as intra-ISE resource sharing, whereas spatial resource sharing is referred to as inter-ISE resource sharing. Section 4.3 covers this temporal partitioning approach, called "CFA staggering" due to its similarity to the loop staggering approach for increasing OLP. The effects of this technique on the performance of the CFA design is studied in the latter section of this chapter.

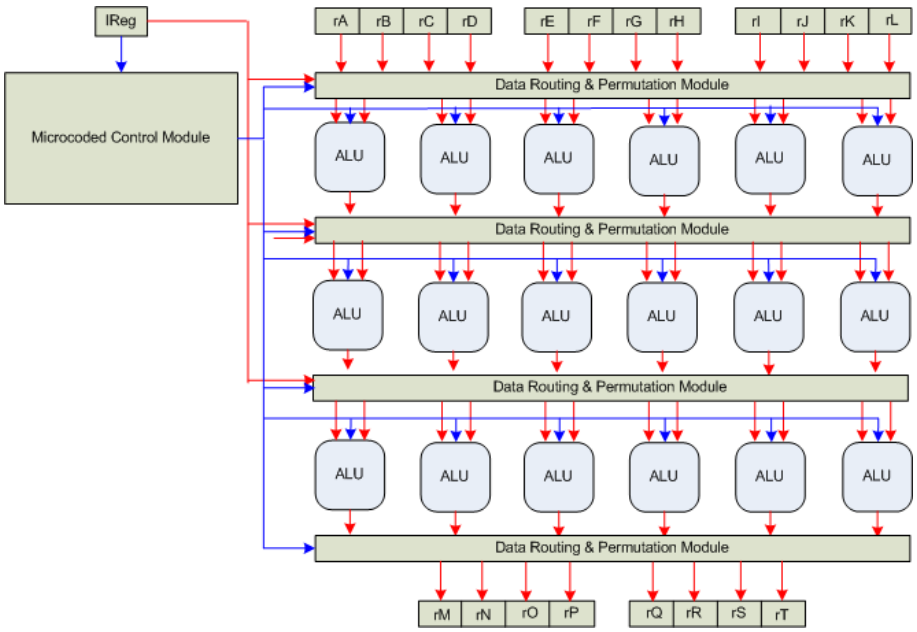


Fig. 4.1: Very generalised CFA, illustrating the Verilog modules and connection in the microarchitecture. The red arrows indicate the data plane, and the blue indicate the control plane. Boxes labelled rA-rL are input registers, grouped into the 4-element vectors used in the EnCore [10] extension interface. Boxes labelled rM-rT are output registers, again grouped as per the EnCore. ALU are single-function; although control exists to select sub-functions such as operand type, each ALU node performs only a single arithmetic operation.

4.2 Configurable Flow Accelerators

The microarchitecture selected for this work is the CFA, a variant of a previously studied [16, 14, 15] reconfigurable ISE implementation. The CFA and the process by which it is constructed are now introduced. *The CFA is demonstrated to be a cost-effective design for ISE implementation.*

4.2.1 Introducing the CFA

Industry's demand for flexible embedded solutions providing high performance and short time-to-market has led to the development of configurable and extensible processors. These pre-verified application-specific processors (ASIP) build on proven baseline cores while allowing for some degree of customisation through user-defined instruction set extensions (ISEs) implemented as functional units in an extended microarchitecture.

The CFA discussed throughout this document is a specific microarchitectural construction similar to the CCA, but with some critical differences. The term CFA has been coined to distinguish this work's contribution from that of Mahlke, Clark *et al.*[16, 14, 15] concerned with the CCA. Figure 4.1 illustrates the microarchitecture of a single, very generalised CFA. As with CCAs, multiple CFAs may be utilised in a single processor data-path in order to better cover acceleration opportunities. CFAs do not include dynamic translation hardware [12], preferring

to allow for a reconfigurable ISA accessible at the architectural level. The baseline instruction stream is used by CCA to extract complex operations dynamically. The instruction stream, however, provides a lowered and hence less efficacious representation for complex operation extraction, than higher-level compiler intermediate representations [135]. In place of the CCA LUT, the CFA has a similar construction called the "Microcoded Control Unit" (MCU). The MCU is mutable on the fly using special instructions to move horizontal microcode from memory into the MCU. In this way the MCU may be initialised with microcode embedded in the executable binary, inserted statically at compile time.

More explicitly, the CFA is a reconfigurable functional unit (RFU) comprised of a number of "echelons", in sequence. Each echelon contains a number of single-function 32-bit scalar ALUs, where every input of each is programmable on the fly via the beforementioned MCU. Figure 4.1 demonstrates a CFA containing three echelons, each having six ALUs of a single cycle latency each. Where ALUs are multi-cycle (e.g. Multiplication, Division), units are pipelined and their routing skips over multiple echelons until they reach a point at which they can return their result to a permutation module. The permutation and routing modules consist of a number of registered multiplexors, one for each input to an ALU that follows.

An echelon is therefore N 32-bit M-input multiplexors, followed by N registers, followed by ALUs with total of N inputs, each attached to one register. M equals the number of inputs to the multiplexor layer, and N represents the number of outputs. The echelon comprises a single clock cycle. In order to prevent the critical path of an echelon from exceeding the clock period, the number of inputs to a multiplexor is capped, and hence there is a worst case for the latency possible. Moreover, multiplexors from N=2 to N=limit are synthesised up front in order to determine the range of latency. The ALUs are all pre-synthesised with input latencies equal to the largest possible multiplexor delay, to guarantee that we can meet the clock period.

In order to configure the CFA for an instruction, we therefore require several lines of microcode: one for each layer of the CFA that is used. The microcode includes fields to control each of the multiplexors and the register-enable that they feed. For example in figure 4.1 we would have four lines of microcode to control each of the permutation layers. The first of these would be taking 12 inputs (grouped here as three four-element vectors as per the EnCore extension interface), amounting to 4 bits $(ceil(log2(numberofelements)))$ per multiplexor, of which there would be twelve. Setting each of these fields would route the selected echelon input to the appropriate ALU input. Registers are also required to be enabled, which in this case adds an additional 12 bits to the line. The first line of microcode for figure 4.1 is therefore (12*4) + 12 = 60 bits.

Between the ALU-internal pipelining and the permutation and routing modules we have the balanced pipeline that constitutes the CFA. Where there are multiple results expected from an ISE running on the CFA, or where the result is expected to be returned in fewer cycles than the CFA pipeline constitutes, there are additional unclocked multiplexors from the results of

such operations to forward their result directly to the final stage of the CFA.

CFAs by their very programmable nature implicitly share resources between ISEs; a single CFA is intended to cover many different extension instructions. Configuration memory in the MCU selects the routing and permutation required at each stage in the CFA pipeline. CFAs are designed to allow for exploiting temporal parallelism as well as spatial: CFAs are pipelined to allow single-cycle initiation interval, whereas CCAs are combinational and have a multi-cycle issue latency.

CFAs are constructed by a greedy algorithm similar to the greedy algorithm used to solve the knapsack problem. Modifications are made because the "cost" of two ISEs merged into a single CFA is not additive due to resource sharing, as illustrated in algorithm 2.

Algorithm 2 CFA Construction Algorithm, used to convert ISEs as DFGs into a CFA model under area and latency (represented as multiplexor input width) constraints, and select a good combination for implementation. As implemented in the *uarchgen* tool.

---

1. Instantiate an empty list L of the CFAs to be included in the final hardware model.
2. The list of ISEs is converted to CFA Object Models, and ordered descending by their impact on application acceleration (merit), to produce the list S of candidate single-ISE CFAs.
3. If a limit is imposed on the maximum number N of ISEs, the list S is reduced to only the top N by merit.
4. Iterate through the list S in order; for each CFA $C \in S$:
   (a) Merge C with all CFAs in L to create a new, list M.
   (b) If no CFAs yet exist in L and if C meets area constraints, add C to L and move to next $C \in I$.
   (c) Remove any CFAs in M which has area (including sum of CFA $\in L$) greater than the area constraint, or a permutation and routing module with input width greater than the width constraint.
   (d) If M is empty, all merged CFAs either broke area or input width constraint, so add C to L.
   (e) If M is not empty, Locate the merged CFA in M with the least area increase from merging with C. Add this back into L, replacing the original CFA merged to create the new CFA.
5. Take the now-populated list L and emit each CFA Object Model as a Verilog structural model with associated synthesis flow scripts and test-bench.

The width constraint of algorithm 2 must be set to ensure that the latencies of the permutation and routing modules (see figure 4.1) do not have excessive latency (and hence require further pipelining). In practice this limits the size of individual CFAs, but if a single CFA breaks the input width multiple CFA will be generated instead. Throughout this thesis, the width constraint is set to 38, as this has been found to perform well with the 130nm standard

cell libraries used. Without this limit, the CFA latencies for ISEs can wander outwith those prescribed by the model described in section 2.3.1, due to excessive multiplexing delay.

Algorithm 2 and the CFA structure are related, as the echelon structure of the CFA allows for the resource-sharing (CFA merging) used in the DSE process to operate in a very simple fashion: Where an asynchronous design would have to deal with real-numbered latencies during resource sharing, when two CFAs are merged all that needs to happen is iteration through the echelons of one CFA adding ALU from the other where these are not already present. The runtime of the CFA "Merging" process is therefore very low; linear in the number of ALU in the largest of the two CFAs merged. When constructing a single-ISE CFA such as before merging in algorithm 2, there is a directly implied assignment of ALU operand source and sink. ALU source operands are mapped to the first echelon where all inputs are available. An ALU sink operand is mapped to the echelon existing at the cycle resulting from adding the source cycle to the number of cycles (integer) that the ALU takes to produce a result. Prior to CFA construction all ALU types which are to be used (e.g. Add, Multiply, etc) are taken from DesignWare and synthesised. Pipeline registers are added to ALU where necessary and timed to preserve the clock frequency desired. Input and output delays are included in ALU module synthesis constraints to allow for multiplexing time. Each ALU hardware latency (in cycles) is therefore known in advance. This preparation allows the CFA construction to operate in the knowledge that the design will close timing, removing the need for feedback from CFA DesignCompiler synthesis. The I/O delay allowance generally increases the area of each ALU by a little to produce a faster implementation, but is necessary to close timing in the face of multiplexing delay. Having pipeline registers pre-balanced for the clock frequency required removes the need for further re-timing when the CFA is synthesised, and so makes DesignCompiler synthesis faster; fast synthesis is a desirable property when attempting to perform DSE.

Additional optimisation of CFA merging could be undertaken by relaxing the ASAP mapping ALU to echelon, but the emphasis here was placed on creating a fast baseline automated C-to-gates flow:

An application is taken and processed by ISEGEN for suitable ISE candidates. A subset of these ISE candidates, which fulfil area constraints when combined together into CFAs are selected. The effect on cycle count is calculated for the CFA-based ISEs selected, via the model of section 2.3.1 The CFA(s) are then emitted as a synthesisable Verilog extension to a baseline core. Gate-level synthesis combined with RTL simulation and power analysis tools are then used, to determine the area and power performance of the generated microarchitecture. Further calculations detailed later in this section are performed based on the cycle counts and power measurements to produce energy consumption results. Figure 4.2 illustrates the flow used for the experiments in this section, and indeed in future sections where synthesis is used to determine power and area. The flow is entirely automated, performing design space exploration over

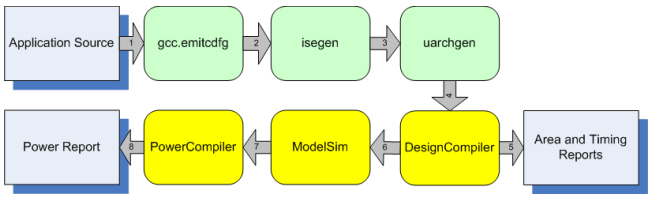


Fig. 4.2: Tool-chain Flow for Experiments: Green indicates tools developed for the work of this thesis, by the author. Yellow indicates existing commercial tools taken to complete the flow.

multiple constraints in order to locate a suitable microarchitecture. CFA can be (re-)configured post-synthesis, although this feature is not the focus of this thesis. CFAs are treated here simply as a microarchitecture for the implementation of a statically known set of ISEs. This section is concerned with validating this basic design flow.

Power consumption is an extremely important factor in todays high-speed mobile embedded devices. The use of Instruction Set Extension (ISE) has been frequently proposed as a potential solution for acceleration of application execution, and a fair amount of work has been done to determine the energy performance of standard combinational ISEs.

Exploration of low-power ASIP microarchitecture ([130], [27]) has been a subject of academic and commercial interest for several years. More directly relevant work includes [132], which was performed for FPGA and soft cores. Whilst the analysis of the energy effects in FPGA fabric is useful, the question of what effects ISE has when using standard cell libraries is also an important one. The energy effects of combinational ISE in a standard cell technology are modelled in [133], and the work examines energy savings in utilising a few (up to seven) large combinational ISEs. The ISEs produced [133] are for some reason extremely low power in comparison to the host core (supposedly less than 1% of the execute stage alone), which is claimed to be due to the simplicity of the AFU circuits in comparison to the synchronous core (described only as “ARM-like”). Due to the lack of any absolute power measurements [133] in favour of relative comparisons it is hard to make any concrete comparisons between that work and this. In [133] the energy saving factor is found to be just above the acceleration factor. Energy estimation for extensible processors is explored in [134], but the approach is a statistical one which must be trained. As mentioned in [133], where RTL models are available as opposed to the C models used in [134] a statistical approach is not necessary as the hardware may be synthesised and analysed directly. Both [133] and [134] are concerned with building power models based on sub-components of ISE microarchitecture, as opposed to the approach taken here which involves directly measuring the power and energy performance of the whole system in order to evaluate the energy performance of the CFA microarchitecture. In [131] the energy saving effects with regards to the register file are examined when forwarding is employed, and up to 25% is found to be saved. The reduction in energy due to reduced memory access and register file pressure is not considered in the work presented here, but all ISEs with

a depth of two or more operations should make such savings; they remove the need for at least one temporary variable to be maintained in a register. The energy model contained herein is therefore somewhat conservative in predicting energy benefits from CFA-based ISE.

It has been found in this work that the use of CFAs can make energy consumption either better or worse, largely depending on the area of the CFAs used. In the best case for energy, 47% is saved from the baseline energy consumption in addition to an increase in speed of 81%. This confirms that it is possible to have both acceleration *and* an increase in energy efficiency when using automatically synthesised CFAs to implement ISE in standard cell technology.

4.2.2 CFA Design Space Exploration Methodology

In order to demonstrate the performance effects of ISE when combined with a standard microprocessor, a selection of tools have been utilised. The combination of such tools into a flow is given in figure 4.2. The locally developed tools, specific to ISE and CFAs are:

- *gcc.emitcdfg*; A modified version of GCC which takes applications as input and produces XML-formatted DFG representing the application. The GNU Profiling extensions to GCC are used to determine the per-basic-block execution counts, which are annotated onto the emitted XML.

- *isegen*; An implementation of the ISEGEN [7] algorithm, taking XML-formatted DFG as input and producing XML-formatted partitioned DFG as a result. The partitions divide each software DFG into a number of ISE candidates and the remaining software (baseline) operations.

- *uarchgen*; Performs the selection phase of ISE generation, in the form of algorithm 2 which takes as input the XML-formatted partitioned DFG and the design constraints (area, width, maximum number of ISEs). Once selection is performed the tool emits a fully-synthesisable structural Verilog model of the CFA microarchitecture (see figure 4.1). We thereon have an implementation of the selected ISEs, profiled cycle counts for hardware and software, and a behavioural test-bench to stimulate the CFAs for power analysis.

DesignCompiler, PowerCompiler, and ModelSim were used for the synthesis from structural RTL Verilog to gates, the simulation of the RTL model, and power analysis of the gate-level model. The full tool-chain used in this experiment is given in figure 4.2. The baseline core used in this work is a 32-bit five-stage EnCore microprocessor [10] with a maximum clock frequency of 250MHz in the 130nm standard-cell libraries used. EnCore is capable of having instruction-set extensions with twelve inputs and eight outputs tightly coupled with the main core. Power analysis of the EnCore has determined that it has a dynamic power consumption of 70uW/MHz, and static power consumption of 0.45mW. This corresponds to an operational

power consumption of 17.95mW at 250MHz (including caches). The EnCore takes 1mm^2 in the 130nm process used. A floating point unit (FPU) has been synthesised to support the baseline case in each of the benchmarks which utilise floating point. In the process used, the single-precision FPU consumes 26.8mW (107uW/MHz) dynamic power, 0.36mW static power, and 0.132mm^2 gate area. These power measurements are used to represent the power consumed whilst operations not assigned to instruction set extensions are processed. The FPU is included in the model for benchmarks which utilise floating point. To determine the average energy consumption per application with both baseline and extended cores, the number of software-only clock cycles is profiled through a linear model: The latencies of instructions in each basic block are added up and multiplied by the execution count of the basic block. These values are summed over all basic blocks to produce a total cycle count for the application executed in software. This cycle-count calculation is as per the model of section 2.3.1.

The number of cycles removed per-ISE is modeled in a similar manner: The number of cycles taken by an ISE in hardware is subtracted from the number of cycles in software, and is multiplied by the number of uses of that ISE. Seven kernels and one large application were selected as benchmarks for analysis in this manner. The kernels selected were taken from the SNURT [**?**] and UTDSP [136] Suites. The suites were chosen due to their being familiar to the author through previous work with ISE. When dealing with kernels, it is important to concentrate on the kernel itself and not any surrounding "test harness" execution, in order to avoid obscuring the results. The kernels represented here are clean, straightforward implementations, with little overhead from test harness code. Kernels of this sort ought to be very similar from one benchmark suite to the next. Any a-priori optimisation for classical computer architectures would be unsuitable.

These kernels are chosen as a representative set of functions which one may expect to be implementing in a DSP environment. With the ISEs in this thesis covering only data-flow, these are the kernels one would expect to actually glean a benefit from their use. The kernels represent a range of complexity, number format, and OLP:

- *SNURT CRC* (Cyclic Redundancy Check; Integer). Fairly low OLP; simple implementation of the CRC algorithm. Contains significant control-flow which breaks up up sections of dataflow. Largest graph size is 24 nodes.

- *SNURT JFDCTINT* (JPEG Forward-DCT; Integer). High OLP; quite flat implementation of an integer forward-DCT as used in the JPEG standard. A realistic target for ISE, as similar algorithms are used in various forms of media encoding and decoding. The lack of control flow puts a larger strain on the ISEGEN algorithm, as graph sizes are up to 101 nodes - significant when considering the search space size for the problem definition is 2^101.

- *UTDSP FFT 1024* (1024-point Fast Fourier Transform; Floating Point). Medium OLP; A very common algorithm in DSP, straightforward implementation of the algorithm with no optimisation. Largest graph size is 35 nodes: towards the upper end of what an exhaustive algorithm could explore.

- *UTDSP FIR 256x64* (256x64 Finite Impulse Response Filter; Floating Point). Very high OLP; main body of this kernel is very wide and flat, and would normally represent a very good target for vectorisation via SIMD. The algorithm itself is very simple, and provides a good baseline for how well ISE can perform in the face of embarrassingly parallel applications. Largest graph size is 185 nodes.

- *UTDSP IIR 4x64* (4x64 Infinite Impulse Response Filter; Floating Point). Medium OLP; of similar level to the FFT 1024. Again, a very common algorithm in DSP, somewhat similar to FIR but here has a smaller window size which narrows the OLP. Largest graph size is 34 nodes.

- *UTDSP LATNRM* (32x64 Normalised Lattice Filter; Floating Point). High OLP; this is the most complex and largest of the benchmarks here, but does not contain the greatest amount of OLP. The largest graph size is 426 nodes, far outside the realm of something many AISE algorithms could even process. This is included to stress the algorithm to its limits; the largest graph is both wide and tall, representing a considerable amount of complex data-flow.

- *SNURT MULT 10x10* (10x10 Matrix Multiplication; Floating Point). Medium OLP; Conceptually the simplest benchmark here, somewhat similar to FIR in structure. A generic operation that would appear repeatedly in a DSP setting. Largest graph size is 58 nodes.

The application selected is the FAAD AAC (Free Advanced Audio Decoder; Fixed Point/Integer). This larger application has been selected to determine the effectiveness of this synthesis technique on a real-world application, rather than just small kernels.

Each benchmark is first processed by the *gcc.emitcdfg* and *isegen* tools to produce a profile-annotated list of DFG and their partitions into ISEs and software (RISC operations). At this point, the *uarchgen* tool is run without any area restrictions to determine the maximum area required to implement the instructions using the CFA microarchitecture. The *uarchgen* tool (see algorithm 2) is then run again with incremental area constraints, from 0.1mm^2 to the maximum area used in steps of 0.1mm^2. The larger AAC application is sampled at 0.5mm^2 intervals past 8.5mm^2 due to the uneventful continuum of results past that point, and the long time taken by DesignCompiler to synthesise larger designs. Most acceleration has already been exploited before this point so this does not effect the quality of the results. The resulting CFAs are processed by Design Compiler to produce an SAIF (Switching Activity Interchange Format)

forward-annotation file. The test-bench produced by *uarchgen* for each constrained design point is used to stimulate the CFAs under test in ModelSim, which produces an SAIF back-annotation file. The back-annotation SAIF is read back into PowerCompiler, and a power report is produced giving the average dynamic power consumption and static power consumption.

Once dynamic and static power consumption has been determined, the values are combined with the other measurements to produce an overall result for energy consumption for each design point. A description of this energy efficiency model follows.

Energy Efficiency Model Variables

F: Clock Frequency (250MHz)

E_{sw}: Software-only (baseline) energy consumption.

E_{hw}: Energy consumption after extension.

P_{sw}: Combined dynamic and static power consumption for the baseline processor during execution.

P_{sw_cg}: Power consumption for the baseline core whilst CFAs are active (reduced compared to P_{sw} due to clock-gating).

P_{hw}: Power consumption of CFAs during ISE execution.

P_{hw_cg} : Power consumption for CFAs when ISEs are not being executed.

C_{sw}: Number of cycles for software-only execution.

C_{cov}: Number of software cycles covered by CFAs.

C_{hw}: Number of cycles spent executing ISEs on CFAs.

Energy Efficiency Model

$$E_{sw} = (C_{sw}/F) * P_{sw}$$

$$E_{hw} = (((C_{sw} - C_{cov})/F) * (P_{sw} + P_{hw\_cg}) + ((C_{hw}/F) * (P_{hw} + P_{sw\_cg}))$$

The model assumes clock-gating with a single enable signal for the entire extension logic at-once; hence dynamic power for CFAs will be consumed only when units are actively processing extensions. In addition, the EnCore is clock-gated such that the dynamic power is 95% less than peak when CFAs are executing ISEs.

4.2.3 Analysis of the Efficacy of CFA

In this subsection we present and discuss the results produced in the experiment described in the previous subsection.

AAC (FAAD)

Ultimately the goal of ISE synthesis techniques is to enable automated synthesis for large applications, such as those used in consumer embedded electronics. FAAD represents a valid example of such an application, and in order to prove the validity of the above CFA design process it is used here with exactly the same tools and methodology as has been used with the kernels. The first observation made was one outside of the original experiment. Despite presenting around a hundred times more DFGs to the *isegen* tool, the runtime was less than that of some of the "smaller" kernel benchmarks. Since the runtime of *isegen* is sensitive more to the size of individual graphs than to the number of graphs in an application, large applications are tractable assuming they do not contain any excessively large basic blocks. Since unrolling was not employed for this larger application, the size of the basic blocks are not large enough to cause problems for the analysis performed here. The kernels of FAAD are comparable in scope and function to the kernels in the UTDSP benchmark suite. It should then still be tractable to process FAAD with ISEGEN, if unrolling were performed. Manual unrolling was employed for the kernels due to a lack of GIMPLE-level loop unrolling in the version of GCC used. It was not feasible to manually unroll the critical subsections of FAAD in the time available. Unrolling of FAAD would result in higher acceleration and energy savings results, but the results presented here are still valid if not optimal.

The *uarchgen* CFA construction process (algorithm 2) explained earlier in this section is fast due to its partially greedy approach. The design space (figure 4.3) generated by the *uarchgen* has the characteristic shape [97] generally encountered in acceleration-area trade-off. Less than five seconds was required in all cases for *uarchgen* to process each design point and emit results, and the runtime is approximately linearly correlated with the area limit used. At the very low areas, there is not enough area to properly accelerate critical subsections. At around $0.8mm^2$ acceleration of the compute kernels begins to become effective, with the area expendable becoming enough to approach the hottest subsections of the FAAD codec. Due to the large volume of different kernels and potential ISEs, the trade-off between area and acceleration becomes near-linear between $1.8mm^2$ and $5.8mm^2$ with a gradient of $5.2\%/mm^2$. At $5.8mm^2$ (51.3% acceleration) the return on expending more area becomes massively diminished, with the remaining $7.31mm^2$ (to the maximum $13.11mm^2$ of the space contributing only an extra 4.26% to application acceleration (maximum acceleration for all ISEs 55.52%). This is typical of other ISE methodologies [97], and the emergence of this trend in these results is a good validation of the synthesis technique.

Energy results for FAAD do not demonstrate a massive benefit, but there is a subsection of the graph ($0.8mm^2$ to $2.6mm^2$) in which there is a small advantage from the use of ISE. From this subsection of the graph there are two points of particular note:

1. At $1.8mm^2$ the energy consumption is 10.9% improved over the baseline, in addition to

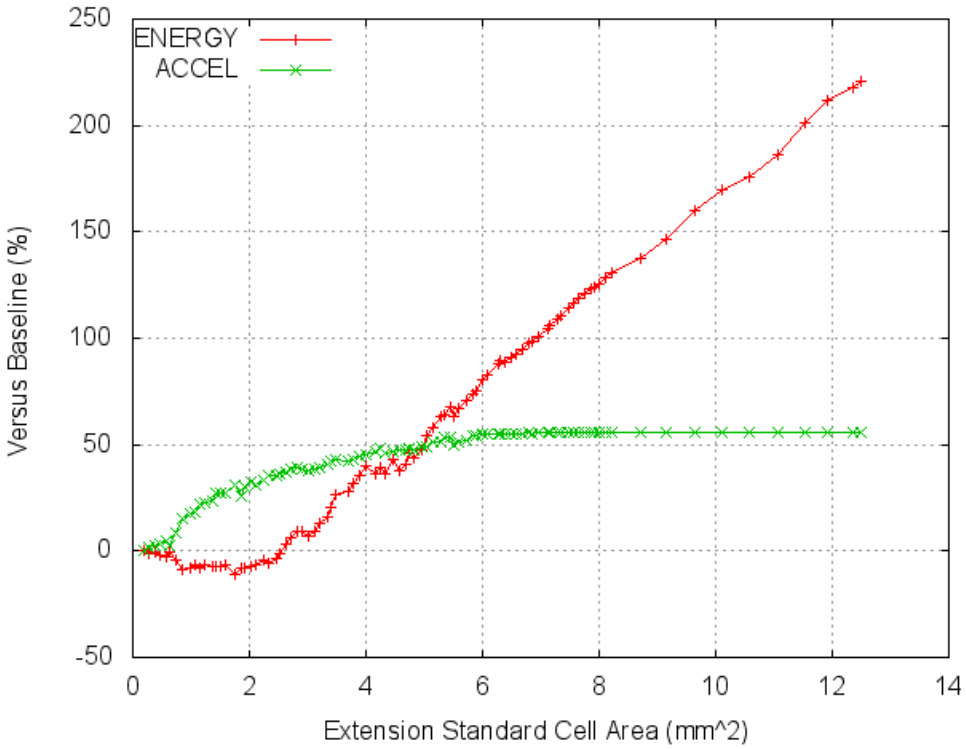


Fig. 4.3: Acceleration and Energy per-area-constraint for AAC Decode (FAAD). The acceleration series demonstrates the classic exponential decay in return (acceleration) on investment (gate area). The energy series demonstrates a net saving in energy from the use of CFAs between 0.8mm^2 and 2.6mm^2. Including energy in our exploration allows us to take this cost into consideration when selecting a design point.

an acceleration of 30.7%. This is the best case for energy consumption.

2. At 2.6mm^2 the energy consumption is 1% improved over the baseline, in addition to an acceleration of 36.5%. This is the best case for acceleration where energy performance is not worsened.

With this synthesis methodology the energy improvements are small for this full application, however, the energy is at least not worsened for a considerable stretch of the graph. One might expect that energy would increase with addition of acceleration hardware, however, the reduction in delay effectively offsets the increase in power until around 2.6mm^2. At that point the increase in power for the larger CFAs constructed past this point is in excess of the energy benefits from lower delay. If lower energy rather than acceleration were desired from a design synthesised in this fashion, the clock frequency could be reduced by the acceleration factor, and voltage could be reduced by the square of the clock frequency reduction. It should be noted again that this work does not involve any scaling of the clock or voltage.

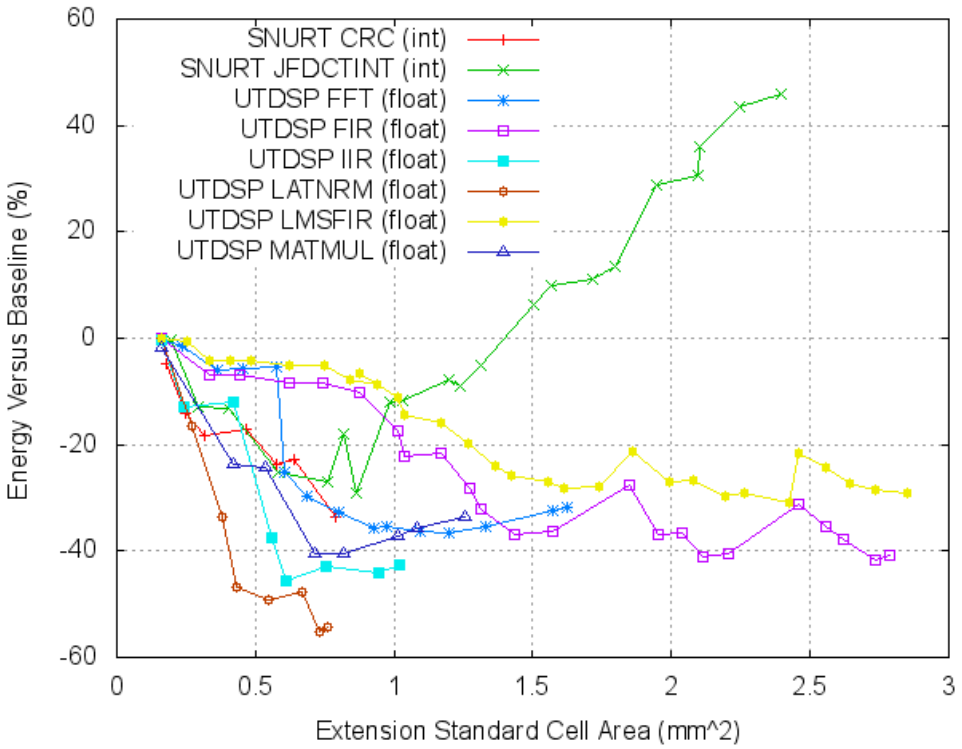


Fig. 4.4: Energy per-kernel and per-area-constraint. The *uarchgen* tool area constraint, is set from 0.2mm^2 up to the maximum for each kernel, in 0.1mm^2 increments. The only kernel not receiving a net energy saving over its entire series is SNURT JFDCTINT. For it, the power requirements of the larger OLP-exploiting ISEs cause the CFA to become an energy burden after CFA area grows above $\tilde{1}$.4mm^2.

Kernels

The energy efficiency of the CFA-based ISE generated for each application and area combination gives the percentage of energy relative to the baseline EnCore when using ISE based upon the CFA microarchitecture to accelerate it. The graph in figure 4.4 gives the energy efficiency of the CFAs produced at each of the application/area constraint pairs. We can see that kernels obtain considerably better improvement in performance than FAAD, but this is largely due to their number format and smaller scope as we now discuss.

The JFDCTINT benchmark is the closest to the FAAD application in terms of the trend observable in its energy performance under CFA-based ISE. It is also a fairly large portion of a real-world application (JPEG encoding), and does not include the FPU. The EnCore consumes a very small amount of power (17.95mW) without an FPU. Extensible cores which consume more power (in the same process as this) will obtain a larger energy improvement factor, assuming they are judiciously clock or power gated as per the energy model of this section.

Those kernels which include the FPU have a baseline power of 45.11mW, which increases the relative energy efficiency of CFAs versus the baseline core. This experiment was previously performed without the FPU consideration, and those floating-point benchmarks which

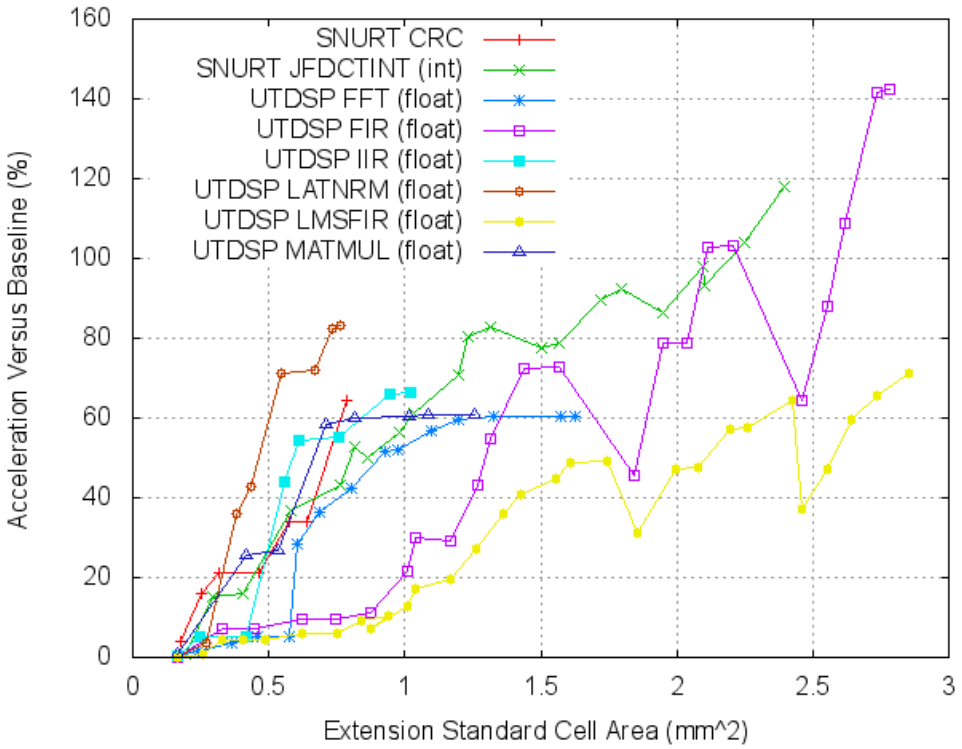


Fig. 4.5: Acceleration per-kernel and per-area-constraint. The *uarchgen* tool area constraint, is set from 0.2mm^2 up to the maximum for each kernel, in 0.1mm^2 increments. As per figure 4.4 the FIR and LMSFIR series share similar features, because they share similar code. The FIR function alone is more trivially accelerated than the additional least-mean-squares processing of LMSFIR, as FIR contains wide OLP. These kernels generate only a few ISEs due to their compact nature. There are not enough ISEs generated to exhibit the power-law which results in the exponential decay of ROI seen in figure 4.3. The selection process (algorithm 2) is less efficacious in the face of fewer ISEs, breaking monotonicity here for FIR, LMSFIR, and JFDCTINT.

now see only positive benefit demonstrated a trend closer to that of FAAD. Even without the FPU consideration, many of the kernels were still much better aided by ISE than FAAD is. JFDCTINT itself receives a considerable positive energy benefit for some portion of the area, so other forces are in play than the inclusion or exclusion of an FPU.

The graph shows that all but one (JFDCTINT) of the smaller kernels receive energy benefit up to and including their area limit. The concentration of OLP-rich computation in the kernels, versus the more sparse OLP in FAAD is a large contributor to the difference in trends. Specifically, the Amdahl acceleration limit described in section 2.3.4 is on average 2-3x higher for these kernels than they are for FAAD. The individual kernels are more regular, leading to a number of similar ISE structures. The regularity is especially strong after the unrolling effort undertaken for these individual kernels, which was not performed for FAAD. The regularity leads to very effective resource-sharing between the resulting similar ISEs and so smaller, lower power CFAs result. FAAD on the is very diverse, containing kernels with very different

data-flow, convolution and encoding being two examples. The resulting diversity of ISE structure inhibits effective resource sharing. FAAD is also very big, with "live" code-size larger than the largest kernel (JFDCTINT) by a factor of around 1000. A larger portion of the FAAD application is therefore not at the maximum "hotness" for the application, as the execution time is spread around the source-base.

An interesting feature we can see in figure 4.4 is the similarity of the trends for FIR and LMSFIR, which share very similar sections of code. We can see the same shape in the trade-off being made between energy and acceleration versus power. In particular, features at 1.7mm^2 and 2.4mm^2 are due to the inclusion of a particularly large ISE that appears in both kernels, but does not benefit acceleration as well as a combination of smaller ISEs. The lack of monotonicity in figure 4.4 is somewhat problematic, and is due to the sub-optimal greedy approach of algorithm 2. Kernels will, however, be the most prone to this lack of monotonicity, as they contain a fewer number of much higher-coverage ISEs. Real applications such as FAAD exhibit a trade-off curve much closer to the monotonic function expected [97].

Deriving from the observations made on energy effects here, and in [132], alongside the model of energy efficiency given earlier in this work we present the following heuristic; any ISE which satisfies the following inequality should contribute a positive benefit to energy efficiency:
C_{ise_sw}: Cycles taken to execute this ISE in software.
C_{ise_hw}: Cycles taken to execute this ISE in hardware.

$$C_{ise\_sw}/C_{ise\_hw} \geq (P_{hw} + P_{sw\_cg})/(P_{sw} + P_{hw\_cg})$$

This will be used in work later in this thesis, to be added to the ISE identification (*isegen*) tool's heuristic in order to optimise directly for energy consumption, rather than acceleration.

For the purposes of acceleration, we can see from figure 4.5 that an iterative evaluation of design points between zero and the maximum desired (or possible) area is beneficial to choosing a suitable design point, as algorithm 2 does not produce a completely monotonic series for area versus acceleration. Larger CFAs do not benefit acceleration much over their smaller counterparts in all cases, and sometimes a lower acceleration results at higher areas outwith the maximum (FFT, FIR, LMSFIR). Energy effects are also sometimes better at lower than the maximum acceleration possible, for example in MATMUL. Whilst the lack of monotonicity in area versus acceleration is a little disappointing, as has already been covered it is somewhat inevitable using the greedy algorithm 2 and a small number of ISEs. We would not expect the graph of energy versus area to be monotonic, however, because energy is not the selection objective. For now we can see, however, that the CFA-based ISEs are capable of reducing energy as seen in other related work, that the acceleration produced is of a level also seen in other related work, and that algorithm 2 functions acceptably and with more efficacy as the size of the problem increases.

Benchmark	Average Area:Power Ratio	σ
SNURT CRC	30.7:1000	0.51
SNURT JFDCTINT	31.17:1000	1.13
UTDSP FFT	26.57:1000	2.16
UTDSP FIR	22.12:1000	2.64
UTDSP IIR	24.44:1000	2.22
UTDSP LATNRM	24.41:1000	2.37
UTDSP MULT	23.92:1000	3.04
FAAD AAC	31.66:1000	0.81

Tab. 4.1: Average ratios of area (mm^2) to power (mW) in the eight benchmarks tested, with associated standard deviations over the samples in each.

Correlation between CFA Area and Power

These results also show a good correlation between the area and dynamic power of CFAs, as shown in table 1. Integer benchmarks CRC, JFDCTINT, and FAAD have a very similar area:power ratio, as do the other floating point benchmarks. This correlation is useful in designing heuristics which are energy-aware, as estimation of area is already reasonably accurate when treated as a "sum of parts" for the individual module areas. This correlation will be used in further work to improve the identification algorithm (*isegen*) heuristics in order to direct the search towards energy-efficient CFAs.

4.2.4 Conclusions

This section has been concerned with determining the various performance effects of CFA-based ISE. A number of conclusions regarding these effects may be drawn:

- This work demonstrates empirically that it is possible to obtain both a reduction in energy consumption and acceleration by utilising ISE in standard cell technology. Energy reduction of 54.4% was obtained in combination with acceleration of 83.3% in one case (LATNRM); similar performance was noted in other kernels.
- Small kernels obtain greater improvements in both energy and acceleration than larger applications.
- There is a near-linear correlation between the die area of CFAs and the power consumed by them when active.
- CFAs which are smaller tend to have better energy efficiency due to the lack of fine-grain clock gating.
- There is a diminishing return on energy savings as the size of the extension logic increases, in addition to similar diminishing returns on acceleration. This is particularly true of larger applications.
- The CFA microarchitecture and construction algorithm 2 employed here are valid for large applications, and produce a more monotonic trade-off curve in such an event.

The CFA construction used in this section is now carried forwards into future sections as the microarchitectural basis of the ISE considered in this thesis.

4.3 CFA Staggering Methodology

CFA designs are sometimes prone to becoming considerably deep, with the functional units closest to the inputs being the most heavily shared and the latter units being under-utilised. In order to address this issue, a methodology for dividing a CFA up into successive temporal partitions and further merging these new subdivisions is now proposed and evaluated. *Staggering is demonstrated on average to reduce the area of CFA implementation by 37% for only an 8% reduction in acceleration.*

4.3.1 Trading off Space for Time

Combinational ISE can make use of a number of distinct techniques for reducing the latency of a particular instruction; most notably accumulating differences between the clock period and operator latency in serial, and utilising parallelism between operators. CFAs do not make use of the serial overhead accumulation because of the echelon-based resource sharing. Instead CFAs may make use of pipelining (temporal) and operator-level (spatial) parallelism. The former temporal parallelism can only be exploited where there are two instructions utilising CFAs executing in an overlapping fashion, which whilst not uncommon is not the case in all situations. In addition, initiation intervals between the two overlapping instructions are not always a single cycle. These observations lead to the idea that temporal parallelism can be traded off with temporal resource sharing, further improving the cost-benefit performance of the CFA microarchitecture for a given application. The later section 5.4 studies the potential for temporal parallelism in ISE, whilst this chapter continues to ignore it in favour of exploring only the acceleration from spatial parallelism as per the common model of section 2.3.1. It was noted during the earlier work performed for this thesis that some CFAs were extremely large compared to their EnCore host core. Overhead is up to a factor of 15x in some cases (e.g. in the presence of a large and diverse application such as FAAD). The temporal parallelism *potential* of the CFAs produced were noted to be often much greater than the potential temporal parallelism of the underlying applications. A method was required to trade off a CFA's potential parallelism for a reduction in area, as potential which cannot realistically be mapped is not useful in a design.

The ISE templates as DFG which are given as inputs to the CFA construction process outlined in section 4.2 will all be meeting input and output port constraints. The process proposed herein is to take each ISE and further partition it. The process is similar to the loop staggering transformation performed as a compiler optimisation for VLIW and other architectures craving OLP and better data-locality [137]. The major difference between loop staggering and CFA staggering is that the former is done in order to increase OLP or data locality in a loop structure, and the latter is done to increase resource sharing and does not improve OLP. An example of CFA staggering can be seen in figure 4.6. The original ISE is taken and partitioned into

sub-ISEs by dividing the critical path into pieces with length targeted at a given cycle interval I, using algorithm 3.

Algorithm 3 Staggering Algorithm.

```
dfg_stagger(DFG, I, SUBS)
00: SUBS ⇐ {}
01: foreach(node n ∈ DFG)
02:     n.input_delta ⇐ cycle at which n reads inputs
03: endfor
05: min_cycle_delta ⇐ 0
06: max_cycle_delta ⇐ I
07: while(∃ unmarked nodes ∈ DFG)
08:     repeat_delta ⇐ false
08:     working_C ⇐ {}
09:     foreach(unmarked node n ∈ DFG)
10:         if(n.input_delta >= min_cycle_delta AND n.input_delta < max_cycle_delta )
11:             if(adding n to working_C will meet I/O constraints)
11:                 add n to working_C
12:                 mark(n)
13:             else
12:                 repeat_delta ⇐ true
13:             endif
13:         endif
14:     endfor
15:     add working_C to SUBS
11:     if(repeat_delta == false)
16:         min_cycle_delta ⇐ min_cycle_delta + I
17:         max_cycle_delta ⇐ max_cycle_delta + I
13:     endif
18: endwhile
```

The complexity of the algorithm is (where V is the set of vertices $\in DFG$) $O(|V|^2/I)$ in the worst case, which represents the case where a DFG is a single dependent string of unary operations with no OLP. This pathological case does not represent the more likely candidates for analysis, which generally form a set of partially balanced trees. Another way of expressing this complexity is (where CP is the critical path of DFG) $O(|V|.CP/I)$, and since CP is almost always less than 20 cycles and N must be at least 1, in the general case the algorithm can be said to have complexity of $O(|V|)$. This linear complexity is a great advantage of the algorithm, which is kept simple so as to promote rapid evaluation of design points in an iterative DSE scenario.

In the example of figure 4.6, the length targeted is a single cycle. Operators longer than a single cycle may cause staggering to produce subgraphs with a latency longer than the target, by up to the latency of the operator minus one cycle.

The relevance of I in algorithm 3 is to the potential temporal parallelism of the underlying

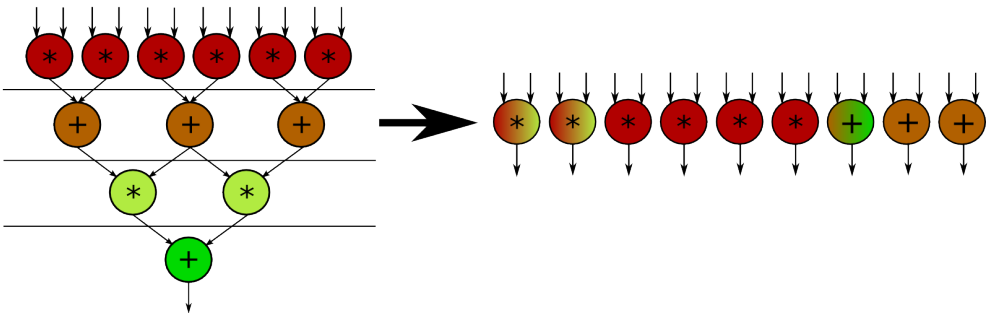


Fig. 4.6: Staggering applied to a simple DFG. The DFG is divided into temporal partitions, as indicated by horizontal lines. The CFA representation is shown on the right, demonstrating the DFG represented in the CFA after construction based upon the temporal partitions extracted from the DFG on the left after staggering.

target application; specifically it is related to the data initiation interval (DII) of a particular ISE within the target application. This algorithm was designed such that where the DII is known for a particular ISE, the value I can be set to balance the result latency of the resulting CFA against the required DII. In many cases, I should be set to the hardware latency of the ISE in question divided by the DII required. This balancing can yield the result wherein the ISE can still be executed to yield the targeted DII, but with a temporally partitioned DFG such that partitions of the ISE would be interleaved in time in order to achieve the desired result. Staggering could therefore be combined with the approach given in section 5.4 in order to maintain low DII for just the ISE where this is useful. ISE overlap may be ignored, and staggering can be used just to improve resource-sharing efficacy and hence area utilisation under the original model of section 2.3.1. This section evaluates such a scenario, and whether or not the CFA staggering algorithm affects the acceleration performance disproportionately for the saving made in area. We also examine the correlation between this trade-off and I. Further work should examine the efficacy of CFA staggering when used with pipelined ISE and when using software pipelining to schedule the resulting subdivided ISE. There was unfortunately not time to investigate such approaches in the course of constructing this thesis.

4.3.2 Comparison to Other Techniques

There exist a number of similar approaches to that taken here, each having a slightly different goal and methodology. The common factor between all of the approaches discussed is that they represent data-flow as a DFG in their analysis.

In [138] the aim is to produce temporal partitioning for data-flow implemented in an FPGA fabric, with the overall goal of reducing the amount of area used without excessively increasing the execution latency of the underlying functions. The work further attempts to reduce the number of temporal partitions required to implement larger data-flow in a given area constraint by temporal resource-sharing of the FPGA coarse grain functional units within a single tempo-

ral partition. The temporal partitioning, resource sharing, and scheduling of the resources are all integrated into a single algorithm in order to try and remove the inefficiencies of a phased approach. Whilst this work is similar to the approach we take here, the problem is not entirely the same. The need to reduce the number of temporal partitions for latency purposes holds the fore in [138]. In the work presented here there is no pressing need to reduce the number of partitions, as the reconfiguration between temporal partitions is instantaneous. In [138], temporal partitions must be activated by a bitstream reprogramming of the FPGA. Here we need only pass a different sub-op to the pre-configured CFA unit. There is of course a small constant overhead in using more sub-ops, as the CFAs must be microcoded for each sub-op before it is used. In [138] the reconfiguration must occur at every switch between one temporal partition and another. CFAs in contrast, by virtue of having multiple sub-ops and persistent microcode for each, amortize reconfiguration time over multiple executions of a sub-op. The main similarity between [138] and the work presented here is in the temporal resource-sharing of the functional units between temporal partitions.

The reality of FPGA reconfigurability is that new temporal partitions cannot in fact be switched between in a few nano-seconds, as is assumed in [138] and other work. In [139], the work attempts to include the significant reconfiguration delays into the analysis and generally reduce the number of reconfigurations as a priority. Both temporal and spatial resource sharing are applied between modules used in operators, dubbed as "cores", to try and fit a given DFG into an area constraint without requiring reconfiguration. The work concludes that in the case of reconfiguration, the time taken to execute the DFG is dwarfed by the time taken to reconfigure the FPGA.

Moving away from the inefficiencies of hardware virtualisation in FPGA, the work of [140] is the closest to the work presented here. The work concentrates on generating ISEs for a pre-existant RFU, through iterative temporal partitioning. The work presented here concentrates on temporal partitioning in order to generate the original RFU, however, the approaches taken are otherwise similar. In [140] the focus is on mapping ISEs which have been identified without strict adherence to the constraints of the RFU and which would be impossible to map without further partitioning. The RFU in question, dubbed AMBER [141] is very similar to both CCAs and CFAs, with the exception that AMBER supports the same set of operations at each functional unit node within the RFU. CCA support a non-homogeneous set per node, but again allow sharing of the same node between a number of operations. CFAs on the other hand allow only one operation per node and instead perform operation selection through having a wider permutation and routing network. The resource sharing algorithm in [140] operates by attempting to map a selected partition to the AMBER RFU; if not enough resources are available two different strategies may be used in order to remove nodes from the oversized partition into a new partition. HTTP or "Horizontal Traversing Temporal Partitioning" starts with the temporally leading nodes of a partition and moves nodes to the new partition breadth-first until

constraints are violated, at which point the partitioning begins again on the remaining portion of the original partition. VTTP, the vertical counterpart to HTTP, is a depth-first version of what is otherwise the same approach. The approach taken in the work presented here is very similar to the HTTP method, but operates without the constraints of an existing RFU because we are attempting to build a new RFU (CFA), not map to an existing one.

4.3.3 Determining the Efficacy of Staggering

The *uarchgen* tool was modified to include algorithm 3, operating after DFG have been canonicalised and before the CFAs themselves are generated through algorithm 2. Figure 4.2 from section 4.2 covers the same ISE design flow as used in this methodology. In order to evaluate the efficacy of staggering, a range of benchmarks with varying complexity are necessary:

- FAAD (Free Advanced Audio Decoder; full application).
- SNURT JFDCTINT (JPEG Integer DCT; kernel).
- UTDSP ADPCM (ADPCM encode and decode; linked kernels).
- UTDSP COMPRESS (DCT-based image compression; linked kernels).
- UTDSP EDGE DETECT (image edge detection; kernel).
- UTDSP FFT 1024 (1024-point fast-Fourier transform; kernel).
- UTDSP FIR 256x64 (256x64 finite impulse response filter; kernel).
- UTDSP IIR 4x64 (4x64 infinite impulse response filter; kernel).
- UTDSP LATNRM 32x64 (32x64 normalised lattice filter; kernel).
- UTDSP LMSFIR 32x64 (least mean squares finite impulse response filter; kernel).
- UTDSP LPC (linear predictive coding; linked kernels).
- UTDSP MULT 10x10 (10x10 matrix multiplication; kernel).
- UTDSP SPECTRAL (power spectral estimate of speech; linked kernels).

For each benchmark, the C source code is run through application profiling, DFG extraction, and ISE generation phases. Each DFG in the unique set is then processed by algorithm 3 with $I = 1$ and the resulting set of staggered DFGs is then canonicalised and synthesised into CFA Verilog models. The performance of the resulting CFAs with respect to their original benchmark is recorded as per the model detailed in section 2.3.1. Subgraphs induced as the result of the staggering process are modelled as individual ISE when using the model from 2.3.1 to determine the performance. This model does not include the temporal parallelism

(pipelined) aspect of CFA performance. The model is used in this methodology to show the acceleration of staggered CFAs when used as a mechanism for implementing combinational style ISE. If the staggering approach described herein does not greatly detract from the combinational performance of the ISE, then it follows that it will also not detract from the temporally parallel performance of pipelined CFAs.

All generated CFAs are synthesised through a Synopsys DesignCompiler RTL to Gates flow, whereupon the area is also recorded alongside the acceleration and staggering interval I associated with them. The libraries used are once again a 130nm standard cell implementation from a popular commercial library provider.

This methodology is repeated with $I = 2$, $I = 3$, $I = 4$, and $I = 5$ in order to study the relationship between staggering latency (I), CFA area, and ISE efficacy. Each setting for I will constrain the construction process to target a different maximum number of cycles for the CFAs being constructed, and will result in a different configuration of resource sharing with respect to the original identified DFG partitions and the CFA design which they ultimately are mapped to. These results are graphed and analysed in the following section.

4.3.4 Evaluation of Staggering Efficacy

Two graphs have been produced from the results of the above methodology to determine the efficacy of this staggering process. The first shows the acceleration in figure 4.7; each bar represents the maximum acceleration obtainable by the ISE methodology, and is subdivided into the acceleration obtained at each staggering level and with staggering disabled. The second graph in figure 4.10 shows the die area at each staggering point and with staggering disabled; these are divided into different bars because the area itself is not monotonic with respect to staggering level.

The most striking feature of the staggering which can be seen in figure 4.7 is that staggering even down to a single cycle does not dramatically limit the performance of the resulting CFAs with respect to the targeted application. In the least negatively affected case (UTDSP FFT 1024), the loss due to staggering between baseline and $I = 1$ is zero. This can be explained due to the great regularity in the FFT instructions, in that at each temporal partition of the instructions contains a number of operations which all start and end at the same point in time, meaning that there is no overhead due to splitting up the operations temporally. An example from UTDSP FFT 1024, of staggering leading to no overhead, is demonstrated in figure 4.8.

The most negatively affected case (UTDSP Edge Detect) has a number of irregular ISEs, wherein a number of operations which start at the same time finish at different times, and there are further operations which immediately depend on the results of the shorter operations. Timing overhead is therefore introduced when these instructions are broken down into temporal partitions. An example of this irregularity-induced overhead is demonstrated in figure 4.9, as taken from the UTDSP Edge Detect benchmark. Even in this most affected case, the overhead

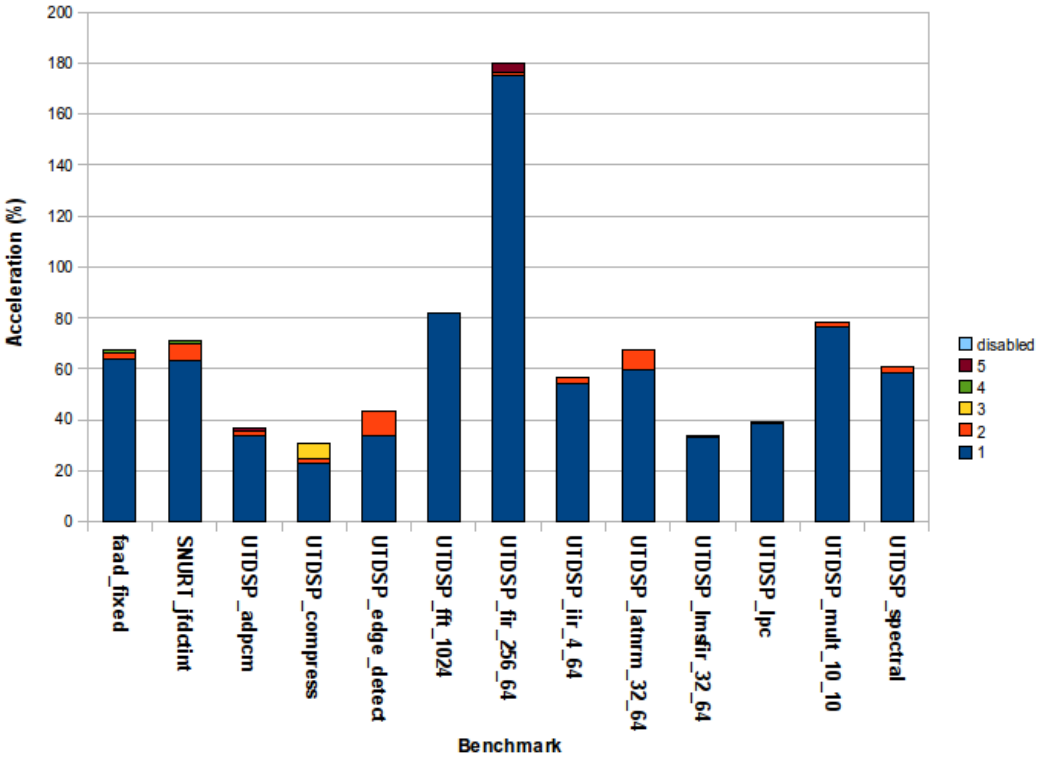


Fig. 4.7: Acceleration for staggering interval of 1-5 cycles, and without staggering (baseline). In all cases the acceleration lost through staggering at the most aggressive interval (1 cycle) is minimal. A maximum of 22% is lost, in EDGE DETECT. This graph demonstrates that staggering at any interval does not excessively stunt acceleration potential.

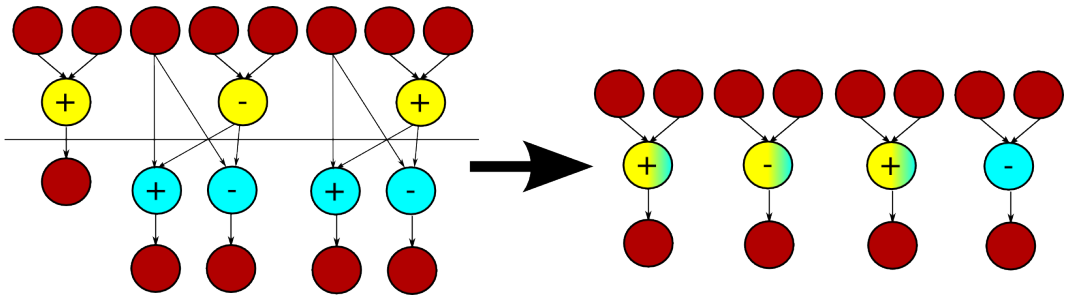


Fig. 4.8: Staggering as applied to an ISE identified in the UTDSP FFT1K benchmark, with $I = 1$. Horizontal lines delineate the temporal partitions produced by the staggering algorithm. The result has no temporal overhead compared to the original.

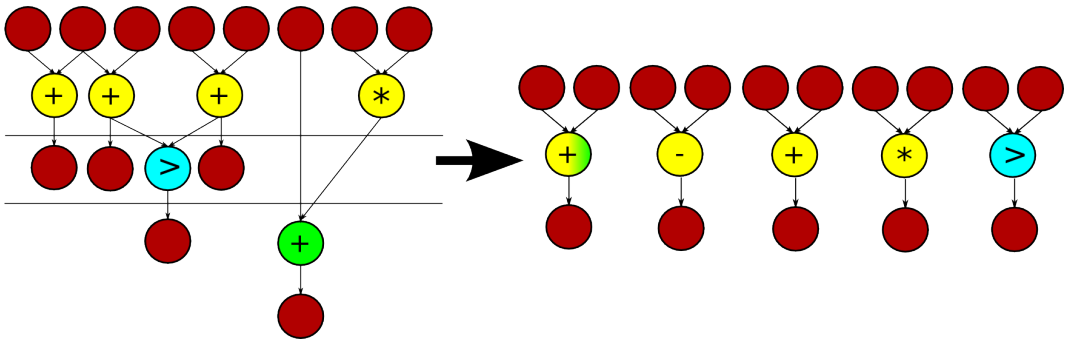


Fig. 4.9: Staggering as applied to an ISE identified in the UTDSP EDGE DETECT benchmark, with $I = 1$. The result has one cycle of temporal overhead compared to the original. The multiplication operation has a 3-cycle latency, and all other operations in this graph have a 1-cycle latency. Horizontal lines delineate the temporal partitions produced by the staggering algorithm. The staggering algorithm at $I = 1$ has missed the opportunity to execute the greater-than operation at the second cycle of the multiplication operation. At $I = 2$ or above the two operations would be in the same temporal partition, and there would be no overhead.

from staggering with $I = 1$ compared to the original baseline is only a relative loss of 22% from the acceleration versus no staggering, when a saving of 29% is made from the extension gate area (0.21mm^2). In this worst observed case for acceleration loss, relatively more is saved from the extension area than is lost from the acceleration.

The large application examined (FAAD AAC Decoder) has a particularly wide array of areas represented in the different staggering levels, however, there is not a vast difference in the actual acceleration resulting from the $I = 1$ staggered and baseline versions of the CFAs constructed. The different CFAs constructed for these two points (two for $I = 1$ and six for the baseline), only differ by 3% in terms of acceleration but differ by a full 9.41mm^2 in terms of their gate area in the 130nm process used. Two effects need to be explained here: The first is the massive difference in area between the $I = 1$ staggered CFAs and the baseline; the second is the very minimal difference in acceleration between same. Both of these effects can be explained through the greater size and diversity of FAAD, and the greater time it takes for the application to execute. The FAAD application is an audio decoding application, which means that it is extremely repetitive over a number of distinct tasks (Decode, IDCT, Re-Encode). In FAAD, many of these tasks share similar but non-identical structure. When $I = 1$ staggering is employed the structure is reduced to simply the number and type of different operations required in each temporal step, with no dependent operators whatsoever within the CFA. This "maximally flattened" CFA is not only the smallest possible with this technique, but is also the best able to exploit resource sharing both inter- and intra-ISE. Different ISEs have their complex structure effectively removed and reduced to the number of each ALU type required, removing the structural complexity from inter-ISE resource-sharing efforts. There is the same effect on structural complexity when considering the intra-ISE equivalent. With $I = 2$ and above, operators which exist in two different temporal partitions will only be shared if they

begin on the same cycle level in a CFA, as per the construction algorithm of section 2. This is the main contributor to the factor of just over two times area increase between $I = 1$ (2.49mm^2) and $I = 2$ (5.6mm^2).

The original purpose of this staggering methodology was to allow for a balance to be struck between the temporal parallelism potential and the temporal resource sharing present in a CFA implementation of a number of ISEs. Insofar as this goal is concerned, it has been proven here that arbitrary scaling of the staggering interval I (related to the DII as discussed earlier) does not have a major negative impact on the non-pipelined acceleration of ISEs. The best and worst cases above aside, the average loss in acceleration between baseline and $I = 1$ across all benchmarks tested is only 8%. On the other hand the area is decreased on average by 37%, or 1.42mm^2.

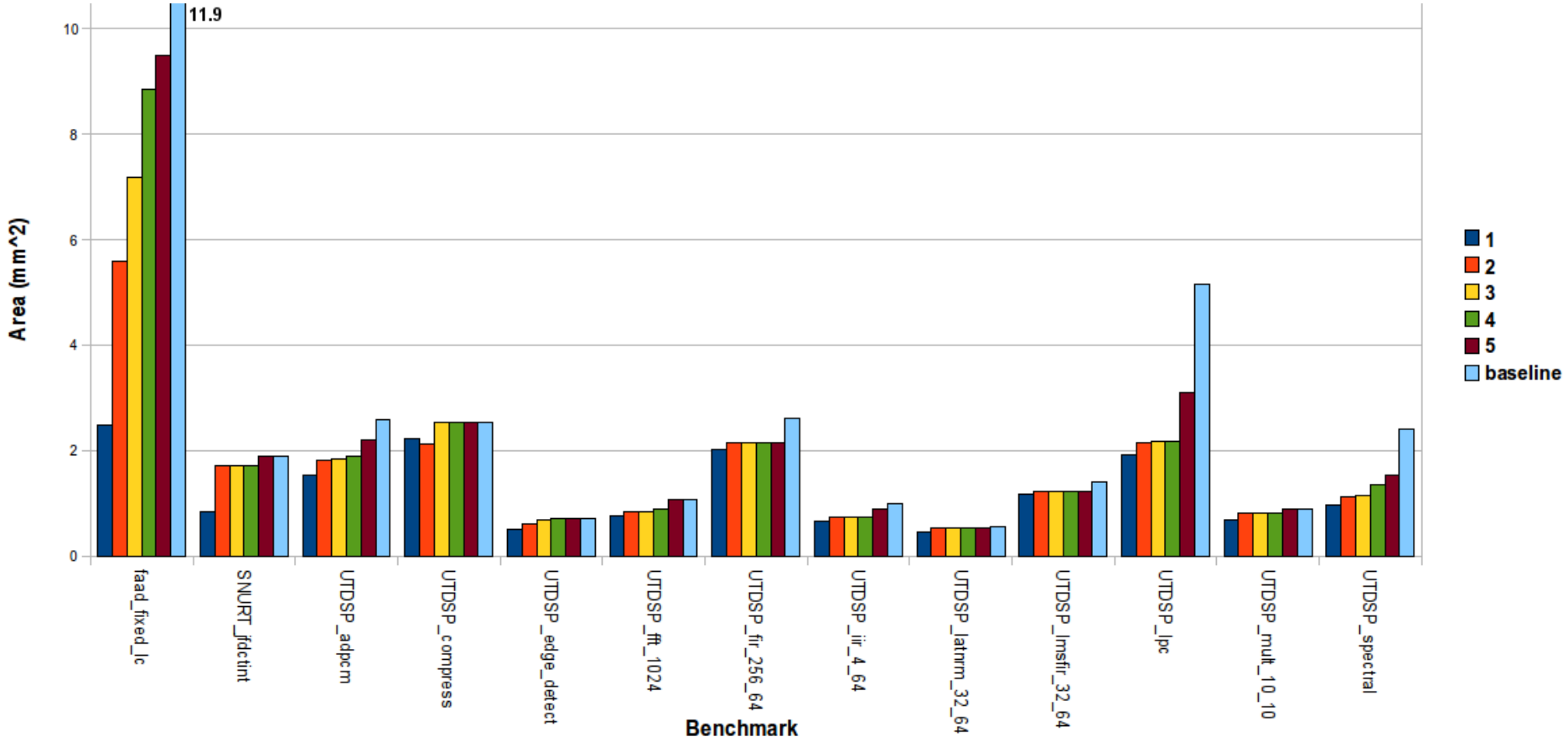


Fig. 4.10: Die area for staggering interval of 1-5 cycles, and without staggering (baseline), when utilising a 130nm process. The large application tested (FAAD) obtains a much greater area saving than the kernels. This is due to the range of complexity apparent in faad which contains a number of distinct kernels. Staggering inherently reduces complexity, greatly improving the resource-sharing in FAAD compared to the more homogeneous kernels. Staggering is more applicable to real applications where CFA are intended to be shared between kernels. FAAD obtains a reduction from 11.9mm^2 to 2.49mm^2, removing 79% of the area. The average lost is 1.42mm^2, or removing 37%. Standard deviation across all benchmarks is 2.54mm^2. Average lost across just the kernels is 0.756mm^2, with a standard deviation of 0.88mm^2.

4.3.5 Conclusions

The study of CFA staggering performed in this section has demonstrated that:

- Staggering can save up to 79% in cell area in the best case (FAAD), losing only 3% from the acceleration afforded; at the opposite extreme a maximum of 22% of acceleration is lost with a 29% cell area saving.
- The greatest area savings are to be had when the staggering interval (I) is set $= 1$, as this allows the maximum degree of resource sharing both intra- and inter-ISE.
- From the point of view of acceleration when considering the ISE are non-overlapping (combinational model from section 2.3.1), staggering does not appear to drastically debilitate the acceleration performance and in some cases the acceleration is untouched regardless of staggering performed.
- Visual examples have been given of the effect of CFA staggering on graphs from applications, which directly support the results obtained herein in both area reduction and the preservation of acceleration.
- CFA Staggering is far more effective in the face of a diverse and ISE-rich application partitioning, owing to the inefficiency of the non-staggering CFA design process in the face of diversity.

Staggering is examined again in section 6.3, where staggering is combined with other new techniques developed in the following chapter to determine how it interferes with other merit objectives.

4.4 Summary

In this chapter the CFA, the process by which one is automatically designed, and a method of compressing the area utilised through temporal partitioning and resource-sharing are covered.

The CFA is demonstrated to be a cost-effective design for ISE implementation. The CFA design achieves both acceleration and energy reduction, achieving up to 2.6x in the former and 1.54x in the latter, albeit in separate cases. The pursuit of both acceleration and energy reduction is a possibility, which will be considered and improved upon in the next chapter. The technique of ISE identification used in this chapter is of the simple combinational heuristic outlined earlier in this thesis in section 2.3.1. The efforts that follow from here concentrate on tuning and modifying this heuristic to produce a better quality of ISEs. There is a relationship between the area of CFAs, the power that they hence consume, and the acceleration afforded. This relationship determines whether and how much an ISE implemented in a CFA will actually benefit energy consumption. This observation will be used in the next chapter to address energy efficiency in the ISEGEN identification heuristic.

In the case of an FPU being contained within the baseline core (i.e. when floating-point benchmarks are considered), beneficial energy effects from CFA use appear to be more pronounced due to the more power-hungry baseline. Two effects are in play with regards to the energy consumption: the time the application takes to execute, and the power consumption of the architecture during execution. When an FPU is included, the power of the architecture executing a "software" partition is greater than that executing one in "hardware". A more in-depth exploration of the effect of number format on ISE/CFA design and implementation is undertaken in the final chapter of this thesis.

Staggering is demonstrated on average, to reduce the area of CFA implementation by 37% for only an 8% reduction in acceleration. Staggering has been shown to be a very effective means of reducing the die area of a CFA implementation, and owing to this ought to also reduce the energy consumption of the design in turn. It is also likely, however, that staggering will interfere with the objectives of identification heuristics tuned for particular axes of concern such as energy efficiency. The energy effects of staggering have not been explored at this point, but will be in the final chapter of this thesis. Up to this point it has been determined that staggering can reduce area by up to 79% with only 3% reduction in acceleration; moreover that staggering is more effective with larger and more complex applications. This chapter has therefore proven the staggering technique to be a good approach to improving the cost-benefit efficacy of the ISE/CFA combination in realistic scenarios, with regards to area and acceleration.

5 BRIDGING THE GAP: IMPROVING ISE IDENTIFICATION

"Strive for continuous improvement, instead of perfection."

– Kim Collins

This chapter is concerned with the improvement of the ISEGEN algorithm through the location of better heuristic weighting vectors, the acceleration of the execution of the algorithm itself through judicious use of newly added early-termination, and the development of new heuristics to target new domains of architectural concern. *A methodology for finding a good static weighting vector for ISEGEN is proposed and demonstrated. Up to 100% of merit is shown to be lost or gained through the choice of vector. ISEGEN early-termination is introduced and shown to improve the runtime of the algorithm by up to 7.26x, and 5.82x on average. An extension to the ISEGEN heuristic to account for pipelining is proposed and evaluated, increasing acceleration by up to an additional 1.5x. An energy-aware heuristic is added to ISEGEN, which reduces the energy used by a CFA implementation of a set of ISEs by an average of 1.6x, up to 3.6x.*

5.1 Introduction

The ISEGEN algorithm has already been introduced and thoroughly described earlier in section 2.4.2, but what has perhaps not been stressed enough is the sheer magnitude of the space over which it operates. Without considering the constraints-imposed pruning such as I/O and convexity, the search space is a massive $2^{number\_of\_nodes}$; graphs from application kernels tend to fall at between 10 and 200 nodes, or a state-space of between 10^6 and 10^{60} points. The latter already falls squarely into intractable territory for a brute-force algorithm. At the extreme end of this scale, encryption applications with control-flow flattened out can have as many as 500-1000 nodes. The number of points in the resulting state-space is between 10^{70} and 10^{221} times the estimated minimum number of atoms in the universe. In such a space, there is ample room for acceleration-equivalent alternatives to be produced for nearly all design points. For any set of ISEs with an acceleration merit M, there will be different sets of ISEs with an acceleration merit within 1% of M. *If you can reach a particular acceleration merit with the combinational heuristic, it is likely that a different heuristic could find a design of equal acceleration merit, but with improved merit in other areas such as standard cell area, power, and energy consumption.*

Previous work such as that of [7] has demonstrated considerable interest not just in the ability to produce a valid result from this massive space, but also in the quality and cost of that result when considering concerns other than acceleration. Examples include power, area, energy consumption, design time, and inter-ISE independence (to enable overlapping ISEs). The original ISEGEN paper [7] made the observation that the algorithm was prone to favouring particular structures wherever they may exist in an underlying DFG, which ultimately makes

the output of ISEGEN more amiable to isomorphism-based resource sharing such as employed by the CFA construction process described in the previous chapter. The ISEGEN identification is referred to [7] as having a behaviour similar to that of expert designers, who would also favour selection of similar ISEs in order to facilitate resource sharing. The ISEGEN algorithm has then already been observed to mimic well the expertise of a trained (human) engineer in producing ISEs which provide ample acceleration and are amenable to resource sharing. We now look at the modification of the algorithm to further improve the efficacy and add further consideration for other design concerns.

At the heart of the ISEGEN algorithm is a compound heuristic, with several different (and potentially conflicting) objectives. In order to weight the different objectives by their relative importance or rank, a vector is used as in many compiler heuristics to form a dot-product with the compound heuristic values as a vector yielding a single scalar "merit". The published work on ISEGEN [7] states that a single weighting vector they give is the optimal. Early work with the algorithm demonstrated to the author of this thesis that in fact better weighting vectors exist. The first section of this chapter is concerned with locating a more effective weighting vector, so as to ensure later comparisons to the combinational results are done with regards to a well-tuned representative baseline.

Before addressing a change of focus for the algorithm objective, repetitive and redundant behaviour in traces of the ISEGEN algorithm were observed during the heuristic weighting analysis, which led to the development of early-termination mechanisms to improve the time taken to produce a result. The second section of this chapter covers the details of this approach, which considerably reduces the algorithm runtime without impairing the quality of its result.

This same heuristic contains in addition to components governing only the "merit" of adding or removing single nodes, a component which gives the merit of a graph as a whole. This component of the heuristic is extensively modified to reflect different design objectives than the simple "number of cycles removed": Energy and Pipelining are both addressed in this manner within the third and fourth sections of this chapter.

5.2 ISEGEN Heuristic Weighting Analysis

The ISEGEN algorithm [7] includes a weighted heuristic used to guide the choice of nodes to toggle from hardware to software during its search for ISE candidates. This work performs a parameter-sweep over the weighting factors of this heuristic, in order to determine the best static vector amongst those explored *and* whether a dynamic approach such as machine learning would be otherwise appropriate. This work has determined that there exist settings that perform relatively well for all benchmarks, but for an optimal performance a dynamic approach is necessary. In this study there was not a single vector which achieved optimum performance on all benchmarks. *The methodology presented for finding a good static weighting vector for ISEGEN demonstrates that up to 100% of merit is shown to be lost or gained through the choice of vector.*

5.2.1 The ISEGEN Heuristic Weighting Vector

The ISEGEN algorithm is effectively a reworking of the Kernighan-Lin circuit partitioning algorithm, allowing for a range of weighted heuristics to be used to guide the search for a suitable partitioning. The algorithm is described in detail in section 2.4.2; the particular weighting of the heuristics as described in that section is not something to which particular attention has been paid, other than to state in the original paper [7]:

"The relations between the weights α_1*,* α_2*,* α_3*,* α_4*, and* α_5 *have been determined experimentally to be as follows:* $\alpha_3 = \alpha_4$*,* $\alpha_1 = 4 \cdot \alpha_3$*,* $\alpha_2 = 2.5 \cdot \alpha_1$*, and* $\alpha_5 = 25 \cdot \alpha_1$*. Thus, by using large factors, the speedup component is favored, the I/O violations are heavily penalized and ISE exploration is allowed to expand in the horizontal direction after the vertical direction has been already explored. We arrived at the above relations by studying the range of values each component can have and identifying the points in the iterative improvement steps where the weights must create a difference in the gain in order to induce a change in the cut growth pattern."*

It was noticed by the author of this thesis that the performance derived from using these weights from the original paper [7] did not actually result in the best performance of the algorithm. For different graphs, different weights appeared to perform better. In order to determine if a better heuristic weighting was available and if a dynamic approach could yield better results, a parameter-space exploration experiment has been performed. Five benchmarks from the UTDSP and SNURT suites along with the FAAD Advanced Audio Codec have been analysed by the ISEGEN algorithm over a large space of heuristic weightings. The results show that the original cited weightings [7] are not optimal, and there no single weighting vector was found that achieves the greatest acceleration in all of the applications and constraints tested. The latter fact motivates the investigation of a dynamic approach, but that is not explored during the course of this thesis and is left for further work.

5.2.2 Weighting Vector Space Exploration Methodology

The ISEGEN algorithm has been implemented in the *isegen* tool, which allows for parameters such as the heuristic weights and number of I/O ports to be passed via the command line. The original ISEGEN toggling heuristic uses five different heuristics in order to determine the merit of toggling a particular node from software to hardware or vice-versa. Section 2.4.2 contains a detailed break-down of the function of these heuristics, referred to here as *savedcycles* (α_1), *io* (α_2), *convexity* (α_3), *largecut* (α_4), and *fragcut* (α_5). In order to produce a single scalar "merit" value for the search to steer between toggling different nodes, the heuristics are combined as follows: $node\_merit = (\alpha_1 \cdot savedcycles) - (\alpha_2 \cdot io) + (\alpha_3 \cdot convexity) + (\alpha_4 \cdot largecut) + (\alpha_5 \cdot fragcut)$ Initial manual exploration in addition to the anecdotal description provided in the original paper [7] suggest that an exponential rather than linear scale of potential weights will produce a better weighting vector for a given number of samples. It was decided for reasons of tractability that since each run of isegen would take up to ten minutes to complete, only six different values could be considered. Covering six values per vector-element leads to a space containing 7776 points to be evaluated. Values $\in \{0, 1, 2, 4, 8, 16\}$ were chosen, representing an exponential scale of weightings rather than a linear one. In this way, the sub-heuristics are assigned to different orders of magnitude: A single-unit change at one order of magnitude will always outweigh a single-unit change at an order below it. The exponential scale in effect assigns the sub-heuristics to tiers of importance; with zero representing the complete exclusion of a sub-heuristic. In this way, we can determine both a suitable weighting vector, and whether any should be excluded entirely.

Register file input and output port constraints of 8-in, 8-out (8/8) and 4-in, 4-out (4/4) are used to determine whether the best observed weightings are different between different I/O constraints. Other studies of ISE have used more varied constraints here; however, the purpose here is not to prove the efficacy of the ISE algorithm but the efficacy of weights used to parameterise it. We only need to determine whether common vectors can be used effectively between different I/O weightings.

Assuming ten minutes per execution of ISEGEN, the 7776-point space would take fifty-four days to evaluate a single benchmark using a single process on a single CPU. This is at the upper end of tractability for a non-distributed approach.

In order to accelerate the execution of these experiments, the Edinburgh Compute and Data Facilities (ECDF) cluster "Eddie" was used to perform evaluation of points in parallel. At the time of performing this work, the Eddie cluster configuration was that:

- Scientific Linux 4.5 64-bit is the basic operating system available to all nodes.
- Sun Grid Engine manages the available resources (nodes) to ensure a fair allocation between users.

- 246 nodes are available.
- Each node is housed in an IBM x3550 server chassis.
- Each node has 16GB of RAM.
- Each node has 250GB of HD for fast node-local disk I/O.
- A storage-area-network provides an additional 275 Terabytes of storage.
- 128 nodes contain 2 Intel "Woodcrest" Xeon 5160 3.0 GHz dual-core CPUs each.
- 118 nodes containing 2 Intel "Harpertown" Xeon X5450 3.0 GHz quad-core CPUs nodes each.
- There are a total of 1456 cores.
- Total theoretical throughput is 12 Teraflops.

Using a cluster, the evaluation of the multiple applications and large design spaces is greatly accelerated by distribution of the parameter sweep across many processor cores. Each parameter space exploration is trivially mapped to the cluster through the specification of an array job, which takes a single program and passes a different index to each new instance. A file containing the weighting vectors, one per line, is used to parameterise the execution of the *isegen* tool for each design point. This is done via scripting, by mapping the array job index to a line number and hence distinct vector from the space outlined above. The Sun Grid Engine scheduler then manages the execution of the *isegen* tool with regards to the 7776 different static weighting vectors.

Applications chosen for evaluation were:

- SNURT CRC; Cyclic Redundancy Check. Naive implementation, not tuned for automated ISE.
- SNURT FFT1K; Fast Fourier Transform - 1024 points. Naive implementation, not tuned for automated ISE.
- SNURT JFDCTINT; JPEG Integer DCT. Mostly straight-line implementation containing large basic blocks with moderate OLP.
- UTDSP FIR; Finite Impulse Response. Loop unrolled sixteen times to create a large degree of OLP.
- UTDSP LMS FIR; Least Mean Square Finite Impulse Response. FIR loop unrolled sixteen times as per simple FIR above to create a large degree of OLP where possible.

- FAAD AAC; Advanced Audio Codec. Full codec application, left untransformed and with moderate OLP.

These applications were selected in order to represent a range of different algorithms with different control and data flow biases, operator level parallelism, and memory transfer requirements. The sparse selection as compared to other work such as the previous chapter is due to the expense of performing such exploration.

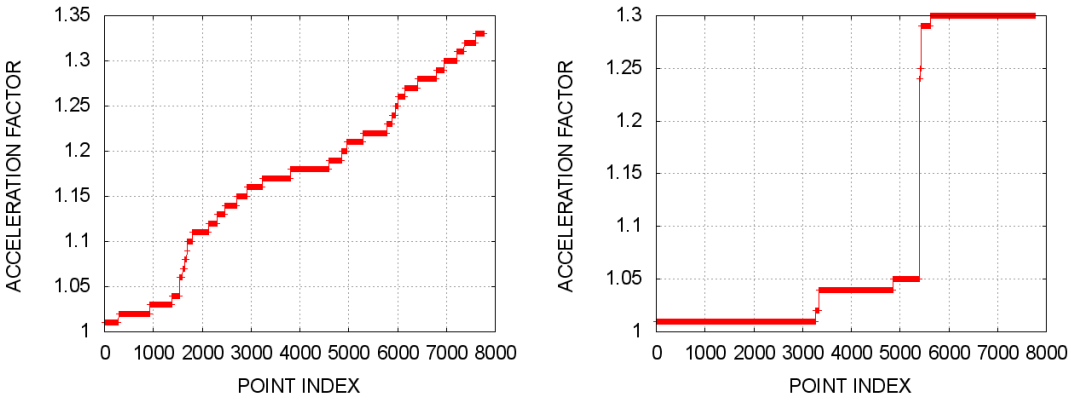
Results are finally ordered ascending by acceleration, forming a monotonic graph for each application and I/O constraint. These graphs are presented in section 5.2.3, alongside further discussion of their implications. In order to determine the best common vector, the set of vectors comprising the top 1% of the speedup obtained for each application are taken and the intersection across the sets calculated. If the intersection contains any elements, then these are given as the best static weighting vector. If no common vector exists, one percent more of the results is taken (i.e. 2%) and the intersection is again examined; this process is repeated until a common weighting vector is located. The results from this process are also discussed in the evaluation below. The number of percent required to obtain a common static weighting vector is referred to as N during the evaluation. The value of N is considered to represent the stability of the heuristic's performance as a whole, with regards to the domain N is measured for. Lower values of N indicate that the heuristic is more efficacious for a wider range of inputs, when using a single static vector.

Resulting from this process, we have monotonically ordered parameter-space graphs for each application and I/O constraint, as well as a series of vectors which form the best individual and best common vectors for the applications and constraints. These results should allow for a discussion of the effects of the heuristic based on the applications in question, and give insight to future development of new heuristics.

5.2.3 Evaluation: Analysis of Parameter Space

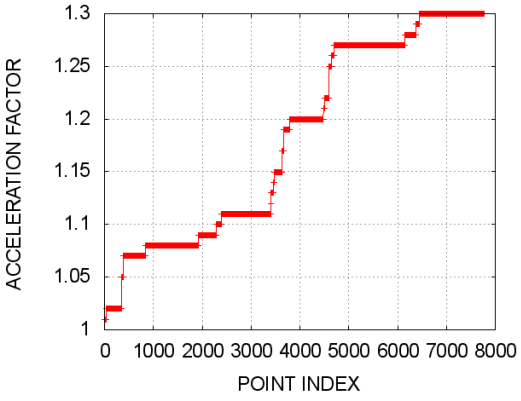
Execution of the parameter space exploration for all the applications and constraints took a total of five days real-time, clocking up around 200 days of single-CPU compute time. A preliminary observation at this point is that cluster computing is very advantageous in performing trivially parallel large-scale sampling efforts. Figures 5.1 and 5.2 contain the monotonically ordered graphs for point index versus application speedup for I/O constraints 4/4 and 8/8 respectively.

All the graphs contain a wide spread of effect from the weighting vector used. The correct selection of a vector is important, covering 20% of the total acceleration achieved in the least affected case (SNURT CRC at 8/8) and 100% in the most effected (UTDSP LMSFIR at 4/4). The specific vectors to use are dependent on the benchmark in question, but similar benchmarks tend to favour similar vectors. Stepping in the graphs plus variety in the underlying designs demonstrates levels of merit-equivalent designs with room for further design concerns to be

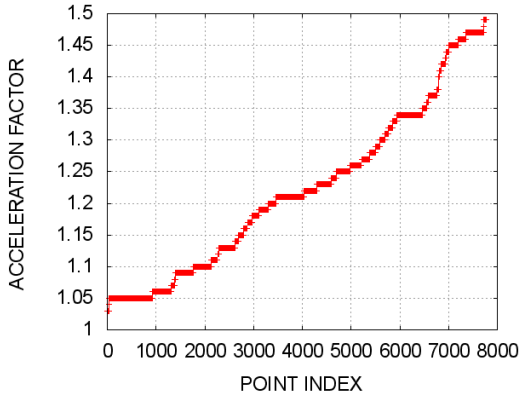


(a) FAAD AAC

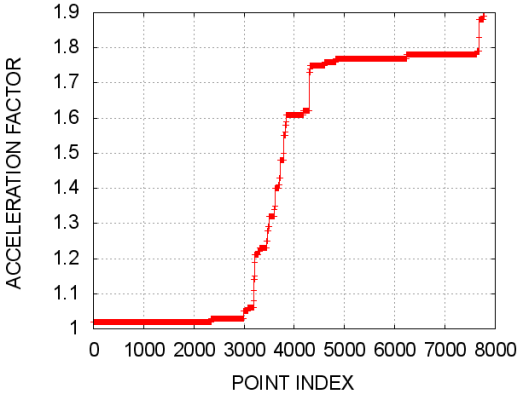
(b) SNURT CRC



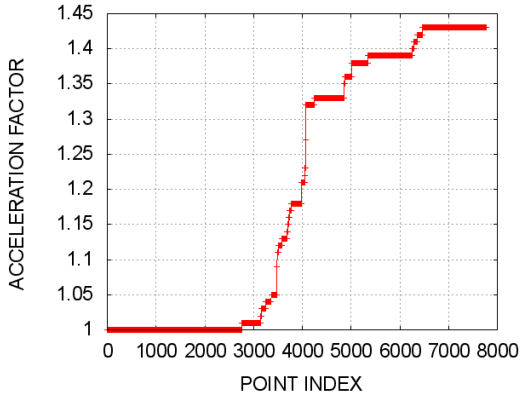
(c) SNURT FFT1K



(d) SNURT JFDCTINT

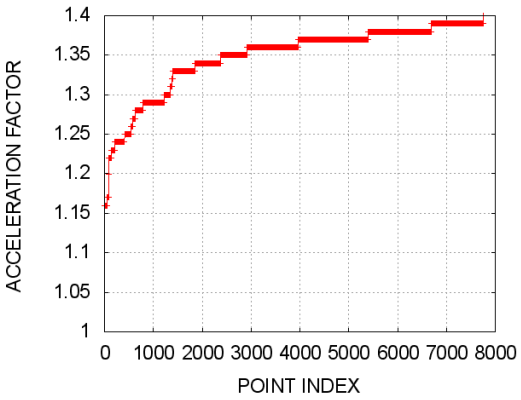


(e) UTDSP FIR

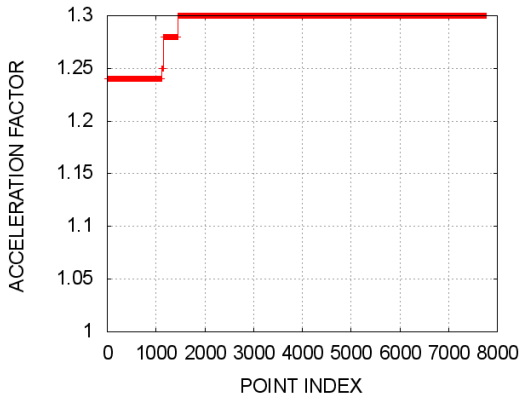


(f) UTDSP LMSFIR

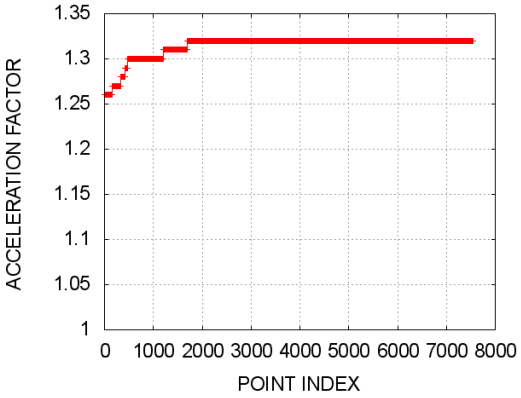
Fig. 5.1: Parameter Sweep Space (I/O: 4/4), Monotonically Ordered by Speedup Factor. The graphs show the distribution of the acceleration resulting from different ISEGEN weighting vectors. The choice of weighting vector affects the quality of the result massively; 100% of potential acceleration by LMSFIR is covered by the choice of weighting vector. These graphs show acceleration is generally lower and more sensitive to the choice of vector, than those for I/O 8/8 (figure 5.2).



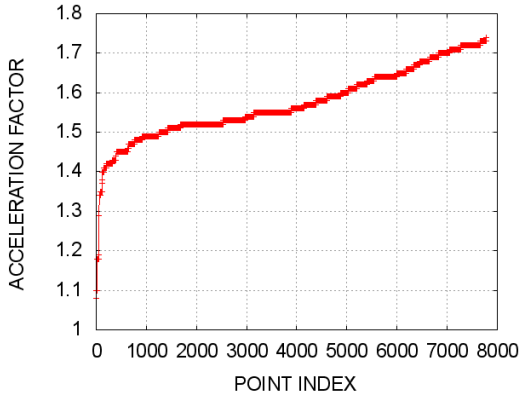
(a) FAAD AAC



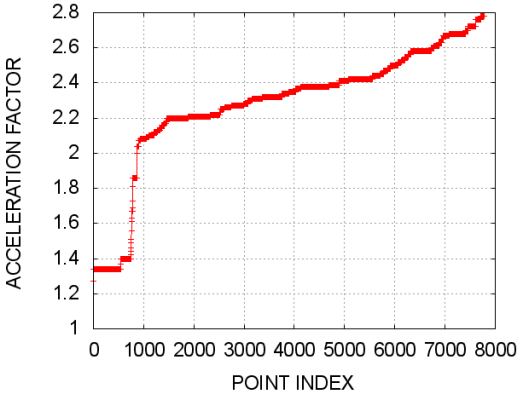
(b) SNURT CRC



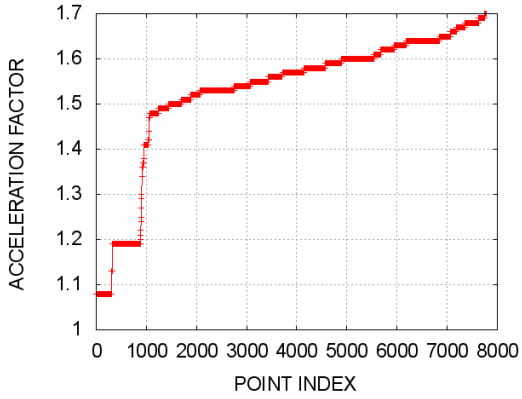
(c) SNURT FFT1K



(d) SNURT JFDCTINT



(e) UTDSP FIR



(f) UTDSP LMSFIR

Fig. 5.2: Parameter Sweep Space (I/O: 8/8), Monotonically Ordered by Speedup Factor. The graphs show the distribution of the acceleration, resulting from different ISEGEN weighting vectors. The acceleration is generally higher, and less sensitive to the weighting vector, than those for I/O 4/4 (figure 5.1).

included. We do not need to explore as much space as we have, as linear equivalents exist in the vector space with regards to their effect on the heuristic. A number of common vectors are covered here, and ultimately we settle upon three sets which are generally the most efficacious seen for the conditions we encounter and the space we have explored: 4/4, 8/8, and combined. The more generalised a scenario you need a static vector for, the less specifically efficacious it will be. This motivates the pursuit of a dynamic approach such as machine learning. These issues are all now discussed in depth.

It follows from the variety of trends observable in the graphs that there is a dependence between properties of the DFG being analysed and the best weighting vector to be used. Since each application tested is comprised of a large number of DFG of varying sizes, where a dynamic heuristic weighting is used it should almost certainly vary between each DFG in a particular application. Similar kernels (UTDSP FIR and LMSFIR) demonstrate very similar weighting preferences, despite the differences between them. The similarity of FIR and LMSFIR results again suggests that there is a correlation between the properties of the DFG in question and the best heuristic weighting vector.

From examination of the results, the "stepping" of the results in most benchmarks represent a number of designs achieving the same merit. The number of steps in each graph corresponds to the number of potential merit levels which ISEGEN has located with modification of the heuristic weighting vector. Of particular note is the size of the lowest performing step, often representing close to zero performance improvement. In the case where the heuristic repeatedly drags the search into invalid sections, the algorithm will rarely settle upon a cut which passes the outer-loop check becoming the best cut. In such cases, the algorithm tends to hit the maximum iteration limit without actually having created any useful cuts. For example, creating one or more serial cuts exploiting minimal OLP. It is interesting to note the size of the lowest step when comparing the results of 4/4 and 8/8 I/O constraints. In the case of lower I/O (4/4), the algorithm has more problems with the misguided heuristics. The toggle selections made in such a case tend to break I/O constraints, before nodes have coalesced enough to be convex. In the larger 8/8 case, more room is available in the I/O constraints for graphs to coalesce to a convex cut through this pseudo-random selection. The choice of a heuristic weighting vector is therefore particularly important when the I/O constraints are considerably lower than the I/O of potential maximal ISE in a given DFG.

The notion espoused earlier in this chapter was that for a given observed merit M there should be a number of design alternatives that obtain M. The stepping of the graphs in combination with multiple distinct designs per step confirms the existence of these merit-equivalent designs. Further heuristic development should be able to widen the design concerns to include merit other than acceleration, and identify more holistically efficacious designs without sacrificing acceleration.

All of the graphs exhibit a degree of stepping as already discussed, but what determines the

threshold between one step and another can be seen by examining the relationships between the weighting vector elements before and after a particular knee. UTDSP FIR contains potentially the most prominent knee, between points 3000 and 4400 in the 4/4 graph; the difference over these points covering 79% (1.05x to 1.75x from a maximum 1.89x) of the total acceleration obtainable over all weighting vectors. The main difference around the knee is the propensity of a zero-weighting for the *savedcycles* component of the heuristic. Of a total 1296 samples having zero *savedcycles* weighting, 1086 (84%) of these resulted in acceleration amounting to less than ten percent of the maximum achieved; the greatest acceleration achieved with a zero-weighted *speedup* heuristic was 76%, obtained for six vectors including the full-zero (all heuristics disabled). The five vectors not being full-zero have disabled all heuristics except for *fragcut*, which has all potential weightings across the five vectors. It would seem that in the case where no other heuristics are active, *fragcut* has no impact for this benchmark. With SNURT JFDCTINT at 4/4, there is not really a knee in the graph. Rather there is a progression, which for all intents and purposes would evade the "by inspection" approach to finding a suitable vector posited by Biswas *et al.*[7]. The lack of large discrete steps in the graph indicate that there is a complex relationship in play, making manual exploration intractable. Looking once more at the *savedcycles* heuristic, we can see that all of the weighting vectors disabling this heuristic fall below 50% of the maximum acceleration available (1.24x out of 1.49x). The remaining weighting vectors have a loose trend of higher *savedcycles* weighting correlating to a higher speedup. This correlation is not absolute, demonstrating the importance of the weighting elements' relative values.

The top speedup for SNURT JFDCTINT I/O constrained to 4/4 (1.49x) is obtained by 61 different weighting vectors. These 61 vectors contain 45 with a *savedcycles* weighting of 16 (the maximum), 14 with a *savedcycles* weighting of 8, and 2 with a *savedcycles* weighting of 4; again this demonstrates the importance of a high weighting of the *savedcycles* heuristic. Due to the linear properties of the heuristic, scalar multiples of a weighting vector are equivalent and will yield the same result. One of the vectors with a *savedcycles* of 4 is repeated in those with a *savedcycles* of 8 (4,1,8,0,1,1 $\Rightarrow$ 8,2,16,0,2,2). Of the vectors with a *savedcycles* of 16, 10 are repeated in those with a *savedcycles* of 16. The number of unique (e.g. excluding equivalent) vectors obtaining the maximum for SNURT JFDCTINT is therefore 50. The linear equivalence has been confirmed here and henceforth will be used to reduce the number of weighting vectors that need be tested. Wherever a vector does not contain the maximum possible value (16 in this case) for one of its elements, scalar multiples of the vector up to and including that containing the maximum possible value can be represented by the lower-value vector. From the original 7776 vectors, the number of non-equivalent vectors is 4652; a reduction of 40% in the vectors which need to be evaluated. In order to determine a more general understanding of the relative weightings, this analysis now turns to locating a common best weighting vector.

Common vectors across all benchmarks are shown in table 5.1; the table details the vector,

along with the performance achieved in each of the benchmarks studied. Across all benchmarks, a range of $N = 14\%$ was required in order to get a common vector. Lower ranges were required across the 4/4 ($N = 10\%$) and 8/8 ($N = 8\%$) cases considered alone. Different, more efficacious static weighting vectors are possible for such narrower domains. This again demonstrates that a dynamic approach would be better suited than a single static vector. Just using two different vectors for 4/4 and 8/8 would raise the efficacy by up to 6%.

Vector	CRC (4/4)	CRC (8/8)	FAAD (4/4)	FAAD (8/8)	FFT1K (4/4)	FFT1K (8/8)	FIR (4/4)	FIR (8/8)	LMSFIR (4/4)	LMSFIR (8/8)	JFDCTINT (4/4)	JFDCTINT (8/8)
8,1,2,4,0	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.720000
16,2,4,8,0	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.720000
8,1,4,4,0	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.720000
16,2,8,8,0	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.720000
16,1,1,8,2	1.300000	1.300000	1.330000	1.380000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.640000
8,2,2,1,1	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.730000
16,4,2,1,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.310000	1.770000	2.580000	1.430000	1.640000	1.450000	1.720000
16,1,0,4,4	1.300000	1.300000	1.310000	1.350000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
8,2,8,0,2	1.300000	1.300000	1.330000	1.390000	1.280000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.720000
8,1,0,4,0	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
16,4,4,1,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.310000	1.770000	2.580000	1.430000	1.640000	1.450000	1.730000
8,1,1,2,1	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
8,1,16,2,2	1.300000	1.300000	1.320000	1.380000	1.300000	1.300000	1.770000	2.580000	1.410000	1.640000	1.470000	1.700000
16,2,0,8,0	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
8,2,0,0,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.730000
16,2,2,8,0	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
8,1,16,4,1	1.300000	1.300000	1.290000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.640000
16,4,0,1,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.720000
8,1,1,4,0	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
4,1,0,0,1	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.730000
16,2,1,4,2	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
16,4,1,1,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.720000
16,1,0,8,2	1.300000	1.300000	1.330000	1.380000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.650000
16,2,16,0,4	1.300000	1.300000	1.330000	1.380000	1.280000	1.320000	1.770000	2.580000	1.410000	1.640000	1.490000	1.720000
16,4,2,2,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.720000
4,1,16,2,0	1.300000	1.300000	1.310000	1.390000	1.280000	1.320000	1.770000	2.580000	1.410000	1.640000	1.470000	1.700000
8,1,8,0,2	1.300000	1.300000	1.330000	1.380000	1.280000	1.320000	1.770000	2.580000	1.410000	1.640000	1.490000	1.720000
8,2,1,1,1	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.720000
8,2,0,1,1	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.720000
16,2,2,4,2	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
8,2,16,1,1	1.300000	1.300000	1.330000	1.390000	1.280000	1.300000	1.770000	2.580000	1.410000	1.640000	1.470000	1.720000
4,1,8,2,0	1.300000	1.300000	1.330000	1.390000	1.280000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
16,4,4,0,4	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.730000
16,4,16,0,4	1.300000	1.300000	1.330000	1.390000	1.280000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.720000

(continued over)

Vector	CRC (4/4)	CRC (8/8)	FAAD (4/4)	FAAD (8/8)	FFT1K (4/4)	FFT1K (8/8)	FIR (4/4)	FIR (8/8)	LMSFIR (4/4)	LMSFIR (8/8)	JFDCTINT (4/4)	JFDCTINT (8/8)
8,2,2,0,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.730000
16,4,0,0,4	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.730000
8,1,0,2,1	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
8,2,1,0,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.730000
8,2,16,0,2	1.300000	1.300000	1.320000	1.390000	1.280000	1.300000	1.770000	2.580000	1.410000	1.640000	1.490000	1.720000
16,4,0,2,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.720000
8,2,16,2,1	1.300000	1.300000	1.330000	1.390000	1.280000	1.300000	1.770000	2.580000	1.410000	1.640000	1.470000	1.700000
8,1,16,0,2	1.300000	1.300000	1.320000	1.380000	1.280000	1.300000	1.770000	2.580000	1.410000	1.640000	1.490000	1.720000
16,4,2,0,4	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.730000
16,4,16,2,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.670000	1.450000	1.720000
16,4,8,2,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.730000
16,2,0,4,2	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
8,2,8,1,1	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.670000	1.450000	1.720000
16,1,2,8,2	1.300000	1.300000	1.330000	1.380000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.640000
4,1,8,0,1	1.300000	1.300000	1.320000	1.390000	1.280000	1.300000	1.770000	2.580000	1.410000	1.640000	1.490000	1.720000
8,2,16,4,0	1.300000	1.300000	1.330000	1.390000	1.280000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
16,4,1,2,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.720000
16,4,16,1,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.670000	1.450000	1.720000
4,1,4,0,1	1.300000	1.300000	1.330000	1.390000	1.280000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.720000
16,4,1,0,4	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.730000
16,2,1,8,0	1.300000	1.300000	1.320000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.700000
4,1,1,0,1	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.470000	1.730000
16,4,8,1,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.730000
16,4,4,2,2	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.730000
8,2,4,1,1	1.300000	1.300000	1.330000	1.390000	1.300000	1.320000	1.770000	2.580000	1.430000	1.640000	1.450000	1.730000

Tab. 5.1: Common vectors at a maximum loss (N) of 14%, the minimum required to obtain common vectors across all benchmarks tested.

Vector	CRC (4/4)	FAAD (4/4)	FFT1K (4/4)	FIR (4/4)	LMSFIR (4/4)	JFDCTINT (4/4)
2,1,16,0,0	1.300000	1.300000	1.280000	1.830000	1.430000	1.460000

Tab. 5.2: Common vector at a maximum loss (N) of 10%, the minimum required to obtain a common vector across all benchmarks with I/O constraint 4/4.

Vector	CRC (8/8)	FAAD (8/8)	FFT1K (8/8)	FIR (8/8)	LMSFIR (8/8)	JFDCTINT (8/8)
4,16,16,1,1	1.300000	1.390000	1.320000	2.700000	1.650000	1.690000
4,16,8,1,1	1.300000	1.390000	1.320000	2.680000	1.650000	1.690000
4,16,1,1,1	1.300000	1.380000	1.320000	2.680000	1.650000	1.690000
4,16,0,1,1	1.300000	1.380000	1.320000	2.680000	1.650000	1.690000
2,2,16,1,0	1.300000	1.370000	1.320000	2.720000	1.650000	1.690000
4,16,4,1,1	1.300000	1.380000	1.320000	2.680000	1.650000	1.690000
4,16,2,1,1	1.300000	1.370000	1.320000	2.680000	1.650000	1.690000

Tab. 5.3: Common vector at a maximum loss (N) of 8%, the minimum required to obtain a common vector across all benchmarks with I/O constraint 8/8.

For each benchmark, there are a number of weighting vectors which obtain the maximum acceleration observed in this experiment. It should of course be noted that there may be weightings other than those tested which obtain even better results, since the vectors tested here are not exhaustive; exhaustive evaluation would require testing an infinite number of vectors. This work has already proven that there is no single optimum vector, and that the vector quoted as optimal in [7] (4,10,1,1,100) is less efficacious than some of the vectors demonstrated herein.

5.2.4 Conclusions

The exploration performed herein has evaluated a large number of ISEGEN weighting vectors, and has come to the following conclusions:

- A poor choice of vector can dramatically reduce the quality of the result. In the most stark example seen here (LMSFIR), 100% of the potential acceleration was not exploited in around 1/3 of the vectors tested under 4/4 I/O. Most other benchmarks in the 4/4 suffer from a similar problem, with many losing nearly all acceleration in the face of a bad vector.

- The cited EPFL weighting vector [7] (4,10,1,1,100) was not the best under the configuration used herein, but may have been for theirs.

- A common static weighting vector is possible without massively compromising the individual result, although this may be due to the largely DSP-like nature of the benchmarks examined with exception of CRC.

- Even more efficacious vectors may exist; the exploration here was non-exhaustive, as the number of potential weightings is infinite. In the range of vectors between (0,0,0,0,0) and (16,16,16,16,16), there are 1419857 vectors. We have only sampled 7776 of those, or 0.54%.

- Dynamic vector configuration via machine learning could close the gap between the common vector and top individual vector performance, but is likely to require being applied on a per-DFG basis in order to be effective.

- Per-DFG is likely to increase the efficacy, possibly above the top individual single-benchmark merit seen here. The fact that there is a significant difference between the static common-vector and the static per-benchmark best vector indicates that there would be a further improvement from making the vector specific to each DFG.

We now have a methodology for locating a static best common-vector which will be used further when developing new heuristics; the vectors derived in this section will be used as the baseline (control) for the combinational heuristic from hereon.

5.3 Search Early Termination

The ISEGEN algorithm is comprised of a three nested loops, with each performing search for a new ISE candidate at different levels. The innermost loop is governed by the heuristic discussed in section 5.2 and selects the nodes that should be toggled between hardware and software partitions. The outer loop keeps track of the best legitimate candidate obtained so far. The worst case runtime of ISEGEN is $O(|V|^3)$ as discussed in section 2.4.2, hence the execution time of the algorithm for larger graphs is rather excessive. Twenty hours of compute time is not unusual for the algorithm running on a typical personal computer in 2010 for a graph containing 500 nodes and 500 edges. Judicious early termination of the search yields savings in execution time, but care must be taken that the search would not have reached a significantly higher merit design point if it had not been terminated. This section creates and evaluates a method of early termination for the ISEGEN algorithm, whilst demonstrating that the early termination does not adversely effect the quality of the result. *ISEGEN early-termination is and shown to improve the runtime of the algorithm by up to 7.26x, and 5.82x on average.*

5.3.1 Faster ISE Analysis Through Shortcuts

For ISE analysis to yield the best possible results, as large a scope as possible must be provided to the search process. It is a widely accepted idea that the larger an ISE is, the better its performance will ultimately be. This assumption is not necessarily true, and in the presence of wire delays and routing congestion this assumption can easily be overwhelmed. It is, however, a certainty that if a larger scope is provided to ISE analysis, the results produced will be of the same acceleration performance or better assuming the algorithm is optimal. This can be trivially proven by the fact that the larger scope contains the smaller scope, and hence any ISE that could be identified in the smaller scope could also be identified in the larger. Results derived from a wider scope will not suffer from unguided ad-hoc partitioning or limitations on code transformations that would otherwise be required. *The more global a scope that an AISE algorithm makes feasible, the less greedy the analysis will be.*

At the most coarse grained level, the ISEGEN implementation utilised here already avoids analysing DFG which have been profiled as having a zero execution count. This avoids unnecessary processing of DFG which are a part of the application source code but that are redundant so far as the actual application's execution is concerned. This high-level change contributed between a 2-5x acceleration in the execution of the ISEGEN tool runtime, but is not evaluated here. Largely dependent on the redundancy inherent in each application, the change is guaranteed not to have any effect on the model execution time either with or without ISE. It is not so much a form of early termination as of input sanity checking; the latter stage of ISE selection would discard any zero-executed ISEs anyway so producing them is not a useful exercise. The following modifications to introduce early termination reduce the runtime for a single DFG,

rather than just reducing the number of DFG that are processed in total.

One existing termination check already introduced between the original ISEGEN paper [30] and the later publication [7], is that the main search loop should be terminated if the result does not change between iterations. The bounded constant number of iterations (set to six in the original paper, and five in the latest) would seem to have been largely introduced to allow for complexity analysis of the algorithm. Without the bounding, the worst-case run-time of the algorithm is infinite. In practice the result very rarely changes past the third iteration of the outer loop, and so the early termination check on the outer loop [7] saves around two fifths of the algorithm execution on average. This is an example of using the properties of the algorithm to limit its execution time: The outer loop resets the state to the last valid cut at the beginning of each iteration. If the valid cut does not change in an iteration, ISEGEN will just repeat the same fruitless search for as many outer-loop iterations as are run from that point.

Examining the ISEGEN algorithm of section 2.4.2, the three major input-dependent contributors to the runtime are the three nested loops which constitute the main body of the algorithm.

The innermost loop is not a good candidate for early termination as this is the heuristic evaluation loop of the search algorithm. To cut the innermost loop short could very severely debilitate the efficacy of the algorithm in an uncontrollable and highly input-dependent fashion.

The next loop out, the continuation of which is predicated on having unmarked nodes remaining in the DFG, has potential for early termination due to required criteria for synthesisable and schedulable ISEs. It has been noted in this thesis that this loop has a tendency to "wander off" into wholly invalid areas of the search space after a number of iterations exploring valid points. In cases examined prior to the experiments of this section, the search rarely returned to a valid point of the space after having moved into an invalid area. Invalid points result from one of two constraints being violated:

- Convexity Constraint; wherein an ISE must not contain any holes, that is an ISE where the output of any node passes through an uncovered portion of the DFG and back into the ISE. This will not be possible to schedule on the target RISC architecture.
- I/O Constraint; wherein an ISE must have fewer than a set limit of inputs and outputs.

For every full execution of the innermost loop, it is possible to check if any of the nodes do not violate the constraints above. If no nodes are valid, then perhaps an early termination of the middle loop is appropriate. In order to avoid a greedy approach, certain properties of the problem and algorithm can be used. The convexity constraint may at times be violated in the search for a convex solution, as in some cases it is the only way for search to expand past a certain local minima. Of the rare cases observed where search did return from an invalid solution to a valid one, convexity was usually the constraint which was originally broken.

Convexity is therefore not a valid constraint to trigger early termination, as it is sometimes necessary to violate it in order to improve a solution.

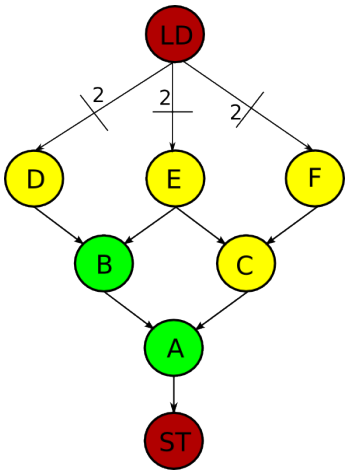


Fig. 5.3: Example of pathological topology required for early termination to change the result versus standard exploration. Given an I/O constraint of 3/3 and assuming nodes are added to the hardware partition in alphabetical order, early termination would stop at {A,B} (I/O: 3/1) whereas the original algorithm would progress to {A,B,C,D,E,F} (I/O: 1/1)

The output constraint has similar issues, in that the search may expand horizontally violating constraints, and then be made valid through the addition of further operators reducing the number of outputs. The input port constraint however has the property that when toggling from software to hardware, the number of inputs will usually either remain the same or increase so long as operator nodes all consume the same or more inputs than they produce outputs. This is the case in all of the binary and unary operations under analysis in the *isegen* tool implementation produced here. That is, the relationship between toggling software nodes to hardware and the number of inputs is *often* monotonic. This means that where only software nodes remain unmarked in an iteration of the middle loop, if all of the nodes will increase the number of inputs past the constraint there is no point continuing the middle loop, as all the following cuts will violate the input constraint. Since cuts are only propagated from the middle loop to the outer loop when they meet constraints, this further search will never actually impact the algorithm's progress other than to increase its execution time. Where any hardware nodes remain there is the possibility that these may be applied in sequence to bring the input count back within constraints, and so such a shortcut may not be taken if hardware nodes remain. It should be noted that this is only a heuristic, as graphs exist which could cause this methodology to terminate search when a better solution could have been found. These graphs are rare however, as they need to encompass several different properties in order to present the condition wherein the ISEGEN algorithm will early-terminate with ill effect. Figure 5.3 demonstrates such a graph and details the conditions required to cause the early termination presented here to miss out on an acceleration opportunity.

A different approach can be taken with hardware nodes; where these exist in the set of unmarked nodes, the sum of their inputs when treated alone provides an absolute upper limit on the number of inputs which could be removed by toggling these to software. The number of inputs in the current cut, minus the sum of the inputs of the *unmarked* hardware nodes,

yields a heuristic minimum number of inputs that might result from their toggling. Calculating the minimum number of inputs in combination with moves from software to hardware that may occur, is much harder to achieve. The heuristic minimum allows for the early termination check to be made more effective by allowing termination to occur when the number of inputs rises above a level where it cannot possibly be reduced to a number within the input port constraint.

At the beginning of each iteration of the innermost loop, we first set a flag to indicate that the loop should early-terminate. When the unmarked nodes are iterated over, if a software node is found which when toggled will maintain the input constraint, the early-terminate flag is set to false. The sum of all hardware nodes individual input edges is also calculated during this process; if the number of inputs in the current cut minus the sum of the hardware node inputs is less than or equal to the input constraint, the early-terminate flag is set to false. If neither the software nor hardware nodes yield the potential for satisfying the input constraint, the iteration is terminated.

The body of the algorithm is therefore modified to include the new termination checks, and becomes as in algorithm 4. The early termination check which is intended to reduce the running time of the ISEGEN algorithm itself incurs a runtime cost. Evaluation must determine that this cost is not overly significant, or at least less so than the search steps it removes. The checks have been deliberately kept simple in an attempt to maintain a benefit from their use in as wide a range of cases as possible.

5.3.2 Validation and Evaluation of Early Termination Approach

The *isegen* tool must first be modified to reflect the changes hi-lighted in algorithm 4. For the purposes of this validation experiment, the ISEGEN implementation is run over the entire SNURT benchmark suite, and the entire UTDSP kernels benchmark suite. Loops in the UTDSP kernels are manually unrolled where this is appropriate, e.g. where loop bodies are independent. This unrolling provides a range of sizes of DFG within the benchmarks, from very small to very large; this in turn can be used to derive a trend between the size of graph and the saving in ISEGEN execution time from the early termination introduced here. In addition, the FAAD "Free Advanced Audio Decoder" application is used to represent a real-world application as might be processed by the algorithm. The machine used to run the experiment is an unloaded Intel Core2 Duo (E6300) clocked at 1.86GHz, with 2GB of RAM, running Scientific Linux on kernel 2.6.18. The timing measurements have been taken using the `time` command; the *isegen* tool loads the input and output filenames, heuristic weights, and constraints from the command line and exits after processing, non-interactively. The I/O involved in file manipulation has been profiled and found to be insignificant compared to the ISEGEN algorithm itself, which constitutes over 99% of the runtime even in small tests. The input and output port constraints are set to 8 each respectively, as this represents a commonly accepted trade-off.

In order to determine the relative efficacy of the early termination, the following modes are

Algorithm 4 The main function of the ISEGEN algorithm, with middle-loop early termination added to reduce the runtime of the algorithm by removing redundant search. The additions are marked in **bold**. When early termination triggers, the control flow leaves the middle loop at line 22 and rejoins the outer loop at line 32.

```
ISEGEN(C, DFG)
00: SetInitialConditions()
01: last_best_C ⇐ C
02: for(i=0, i < NUM_ITERATIONS, i++)
03:     working_C ⇐ last_best_C
04:     best_C ⇐ last_best_C
05:     hw_inputs_max ⇐ 0
06:     early_terminate ⇐ true
07:     while(∃ unmarked nodes ∈ DFG)
08:         foreach(unmarked node n ∈ DFG)
09:             Calculate M_toggle(n, working_C)
10:             if(n.in_cut)
11:                 hw_inputs_max ⇐ hw_inputs_max + n.num_inputs
12:             else
13:                 if(n.input_delta + working_C.num_inputs <= MAX_INPUT_PORTS)
14:                     early_terminate ⇐ false
15:                 endif
16:             endif
17:         endfor
18:         if(working_C.num_inputs - hw_inputs_max <= MAX_INPUT_PORTS)
19:             early_terminate ⇐ false
20:         endif
21:         if(early_terminate)
22:             break
23:         endif
24:         best_node ⇐ Node with Maximum M_toggle
25:         toggle(best_node, working_C)
26:         mark(best_node)
27:         CalcImpactOfToggle(best_node, working_C)
28:         if(working_C satisfies constraints AND M(working_C) >= M(best_C))
29:             best_C ⇐ working_C
30:         endif
31:     endwhile
32:     if(M(best_C) > M(last_best_C))
33:         last_best_C ⇐ best_C
34:         unmark_all(DFG)
35:     else
36:         i ⇐ NUM_ITERATIONS
37:     endif
38: endfor
39: C ⇐ last_best_C
```

timed for each benchmark:

- Without the new middle-loop early termination (**baseline**).
- With the new middle-loop early termination (**early termination**).

Every timing measurement is repeated three times, in order to compensate for any jitter that might be encountered. The resulting three measurements are combined to form a mean average for each combination of benchmark and termination mode, which is then used in the following evaluation.

A bash script is used to initiate the tests automatically over the settings above, and only one instance of the *isegen* tool is run at a time. The combination of 2 different termination modes, 3 repetitions, and 31 benchmarks leads to a total of 186 timing measurements. These are left to run on the single machine used in the experiment over the course of several days.

In order to ensure that the output of the algorithm is not affected by the early termination changes, the output of each of the termination modes for each benchmark is cross-checked for equivalence. If the assumptions made in the description of subsection 5.3.1 are true, the result of running ISEGEN should not change due to any of the early termination employed here. The topology and conditions detailed in figure 5.3 which could lead to the early termination making the result less efficacious will be detected by this check.

5.3.3 Evaluation of Validatory Results

Following the 186 runs of the *isegen* tool as outlined in the above experimental methodology, the results of all the combinations of termination strategies for each benchmark were compared for equivalence. In no case was the result given by the *isegen* tool different depending on the termination mode used. The early termination methodology improves the run-time of the tool by a significant degree, as is now discussed in more detail.

Benchmark	Baseline Time (s)	Early Termination Time (s)	Tool Speedup Factor	Benchmark Speedup Factor	#DFG Active	Max(#nodes)	Max(#edges)
SNURT_insertsort	0.02	0.02	1	1.15	10	13	12
SNURT_fibcall	0.02	0.02	1	1.17	6	13	12
SNURT_matmul	0.08	0.08	1	1.32	25	17	18
SNURT_select	0.08	0.08	1	1.05	51	14	14
SNURT_bs	0.09	0.09	1	1	87	41	40
UTDSP_edge_detect	0.12	0.12	1	1.42	29	32	29
SNURT_qurt	0.12	0.13	0.92	1.12	34	28	22
UTDSP_histogram	0.12	0.12	1	1.2	19	23	23
SNURT_sqrt	0.18	0.18	1	1.11	48	22	20
SNURT_qsort	0.2	0.2	1	1.12	77	16	18
SNURT_ludcmp	0.22	0.22	1	1.15	36	18	19
SNURT_minver	0.23	0.23	1	1.13	67	18	16
SNURT_crc	0.24	0.24	1	1.05	44	24	24
UTDSP_fft_1024	0.5	0.47	1.06	1.82	20	35	44
SNURT_fir	0.52	0.52	1	1.18	44	28	36
UTDSP_iir_4_64	0.66	0.65	1.02	1.57	19	34	47
SNURT_lms	0.72	0.72	1	1.13	58	26	34
SNURT_fft1	0.78	0.72	1.08	1.24	48	38	47
SNURT_fft1k	0.84	0.81	1.04	1.28	45	35	51
UTDSP_spectral	0.92	0.88	1.05	1.6	41	37	55
UTDSP_adpcm	4.36	3.19	1.37	1.37	88	64	85
SNURT_adpcm_test	5.28	4.99	1.06	1.14	197	84	137
UTDSP_mult_10_10	6.17	4.07	1.52	1.75	19	58	104
UTDSP_lpc	9.47	10.64	0.89	1.39	70	72	117
SNURT_jfdctint	252.95	125.48	2.02	1.71	8	101	152
FAAD_fixed	710.43	408.21	1.74	1.6	1190	154	257
UTDSP_fir_256_64	1036.69	350.81	2.96	2.8	14	185	260
UTDSP_compress	1085.45	345.88	3.14	1.3	53	148	236
UTDSP_lmsfir_32_64	65948.79	9078.6	7.26	1.6	15	469	694
UTDSP_latnrm_32_64	66744.16	12998.68	5.13	1.83	15	426	740

Tab. 5.4: Results of Early Termination Evaluation, ordered by time taken at baseline. Tool speedup factor is the factor by which the runtime of the ISEGEN tool was improved using the early termination heuristic. Benchmark speedup factor is the factor by which the benchmark was accelerated by ISEs identified. This table demonstrates the relationship between the time taken at baseline, and the number of nodes (or edges). The runtime of the basic algorithm is $O(|V|^3)$ where V is the set of vertices (nodes). The algorithm with early termination also has a polynomial run-time, but with an apparently lower exponent than the original. This is made evident by the polynomially increasing relative difference, between the baseline and early-terminating algorithms, as node count increases.

All of the savings in execution time can therefore be considered as zero-cost, other than the additional work required to implement the extra tests in the *isegen* tool. This refinement of the ISEGEN algorithm confers only positive effects on the runtime of the analysis by removing only redundant computation from the search.

Table 5.4 gives all of the results obtained, including information on the dimensions of the underlying DFG in the benchmarks processed.

Despite the additional cost incurred in calculating the termination conditions, in most cases the time taken by the early-terminating runs of the *isegen* tool were shorter than the time taken by the baseline version. The *Tool Accel* column from table 5.4 gives the factor that the *isegen* tool was accelerated by. The most notable result from the benchmarks in figure 5.6 is the UTDSP LPC benchmark, wherein the early termination mode leads to a higher *isegen* tool runtime than the baseline. This is due to early termination taking some time to calculate due to the non-trivial number of nodes and edges in the largest graph, and the fact that the early termination does not remove very much search from this benchmark. The only other example of this effect is SNURT QURT seen in figure 5.7, which falls to the same pathological condition. This effect only occurs in smaller benchmarks, with the largest benchmarks having gleaned a tool runtime benefit in all cases. The more nodes that a DFG has, the more likely it is to trigger a condition where toggling of any remaining nodes cannot lead to a valid solution. In both cases, where tool runtime has been increased, it is not by a large amount: just over a second for UTDSP LPC and less than a second for SNURT QURT. This in comparison to the savings made in other benchmarks is rather insignificant when considered in terms of the cost of an ASIP engineer's time.

The best example in terms of both absolute and normalised (percent of total) saving for the early termination is the SNURT LMSFIR benchmark. This benchmark is reduced from 1099 minutes at the baseline; over eighteen hours, to 151 minutes when using the new early termination method. This is a reduction of 7.26x in runtime, reducing the runtime of the *isegen* tool enough that an engineer can explore considerably more design points in the time available to them for any similarly complex application. The UTDSP LMSFIR isn't the longest application to process in the baseline setting of all those tested, but is very close; only thirteen minutes separate the baseline runtime of UTDSP LMSFIR and UTDSP LATNRM. The largest DFG in UTDSP LMSFIR is 469 nodes and 694 edges including non-coverable operations, as detailed in table 5.4 figure 5.5 illustrates the size and complexity of this DFG.

By far the greatest impact on the ISEGEN algorithm runtime is the size of an individual DFG for processing, since this is subject to $O(|V|^3)$ runtime as discussed in section 2.4.2. The number of DFG in a benchmark only contributes to the overall runtime linearly, because the ISEGEN algorithm itself runs with scope of only a single basic block. Benchmarks with greater numbers of DFG may therefore take much less time to process than benchmarks with fewer. This is due to the complexity of an application being innately partitioned into basic

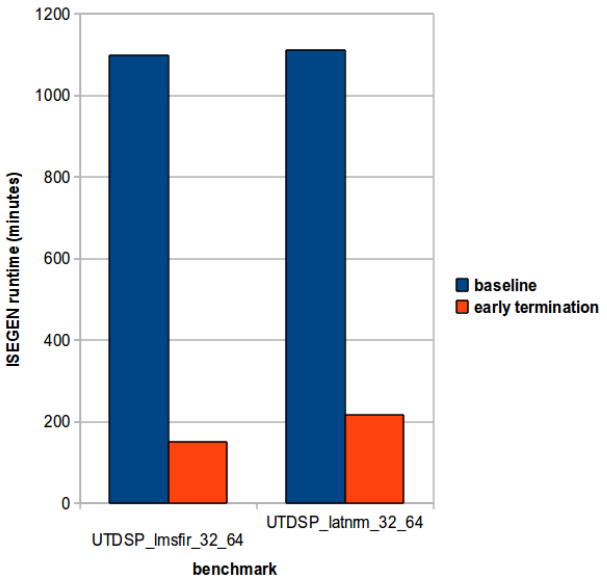


Fig. 5.4: Time taken per termination strategy, for benchmarks taking more than an hour to process. These benchmarks obtain the greatest benefit from early termination; the maximum benefit shown here being 7.26x. This could allow an engineer to explore around the same factor more design points. Average saving is 921.8 minutes, with a standard deviation of 36.83 minutes.

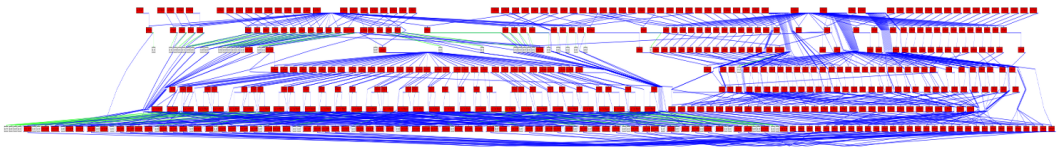


Fig. 5.5: DFG for UTDSP LMSFIR which is the major contributor to the large runtime of the ISEGEN algorithm when run on this benchmark. There are 497 nodes and 694 edges, contributing directly to the runtime of the ISEGEN algorithm. This figure is not intended to illustrate the fine-grain detail of the DFG, but rather demonstrate the size and complexity of the problem faced by ISEGEN for a realistic kernel. This complexity is the result of loop-unrolling the original LMSFIR benchmark to expose OLP. The early terminating algorithm makes feasible the exploration of DFGs with higher OLP, a necessary feature where higher ISE performance is desired.

blocks when processed by the ISEGEN algorithm. These results demonstrate that the early termination method is more effective in reducing the runtime of larger DFG. Looking at the trace of search in baseline and early termination modes it is apparent that the early termination mode removes a vast swathe of fruitless search. The baseline search massively exceeds the input (and output) constraints before falling back to the same final cut as when using early termination.

Taking the mean average factor of acceleration weighted by the time taken at baseline, the early termination accelerates the runtime of the *isegen* tool by a factor of 5.82x. Splitting the results as per figures 5.7 (short), 5.6 (medium), and 5.4 (long); taking the same weighted average as for all benchmarks, the early termination achieves 1.02x, 2.48x, and 6.01x respectively. This approach therefore has greater application in benchmarks taking longer to process, which in itself is a useful feature as it directly competes with the $O(|V|^3)$ runtime of the original algorithm.

The longest benchmark to process is the UTDSP LATNRM benchmark, taking 1112 minutes (over 18 hours) in the baseline and 216 minutes (under 3.5 hours) using early termination. The largest DFG in this benchmark is 426 nodes and 740 edges, which contains more edges than the slightly shorter UTDSP LMSFIR benchmark, but fewer nodes. The tool acceleration of 5.13x is likely to have been smaller than the UTDSP LMSFIR benchmark's 7.26x because the early termination is more effective in the face of a high number of nodes, and should not be effected by the number of edges. This said, there is not a concrete correlation between the number of edges and the tool acceleration imparted, as the results for UTDSP FIR and UTDSP COMPRESS from table 5.4 demonstrate. Early termination contributes a slightly better advantage to the UTDSP COMPRESS benchmark despite that benchmark having more DFG, and a lower maximum number of nodes. The lack of a direct correlation between number of nodes and edges versus ISEGEN runtime is down to the topological differences in the underlying DFG. Different topologies of nodes and edges will lead to different search behaviours. In some cases solutions will be found in the first outer iteration of the ISEGEN algorithm, in others it will take several iterations for this to be realised. Where several iterations are taken, more compact combinations of hardware nodes lead to less effect from the early termination due to the hardware-node element of the check. Since this component of the early termination assumes a worst-case "flattened" organisation of the nodes in order to avoid analysis of topology, it will let the graph grow further into invalid sections before termination is triggered. Graphs which intrinsically lead the search towards flatter (non-serial) ISEs will therefore obtain the greatest benefit from this early termination strategy.

The early termination mode has very little use in benchmarks taking less than a second to complete, as in figure 5.7. In many of these, the time taken using early termination is the same as the baseline. In the benchmarks taking the same time, all ISEs identified are identified in the first outer iteration of the ISEGEN algorithm. At no point in these cases does the search

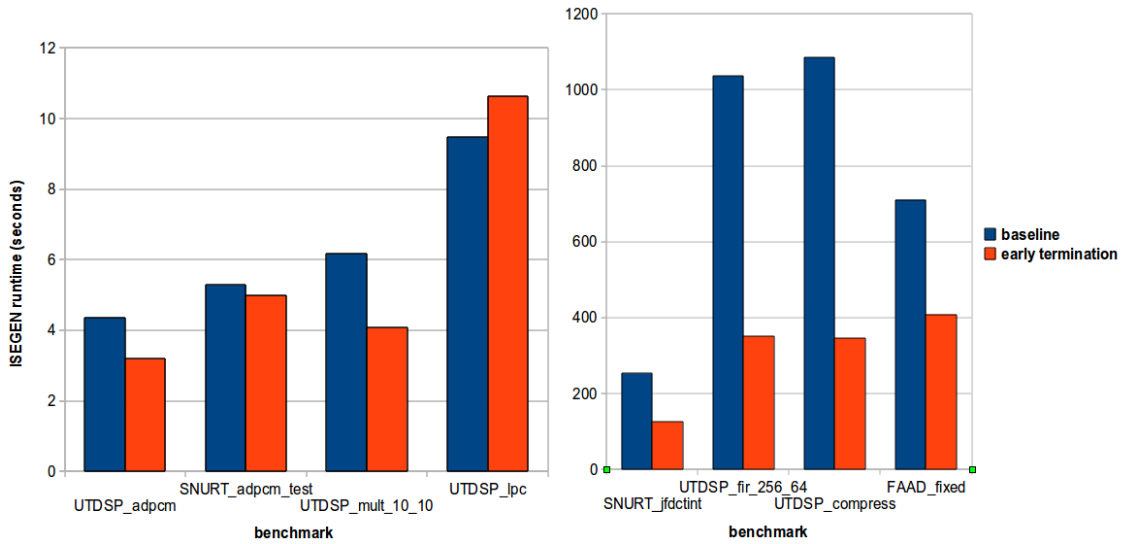


Fig. 5.6: Time taken per termination strategy, for benchmarks taking between one second and an hour to process. Savings made are less than those for figure 5.4, and more than those of figure 5.7. This graph shows an application that is negatively impacted by the early-termination: UTDSP LPC. This benchmark unfortunately gains more overhead from the early termination checks than it saves. Such overhead is only very slight. Average saving for results on the left is 0.6 seconds with a standard deviation of 1.39 seconds. Average saving for results on the right is 464 seconds with a standard deviation of 297 seconds.

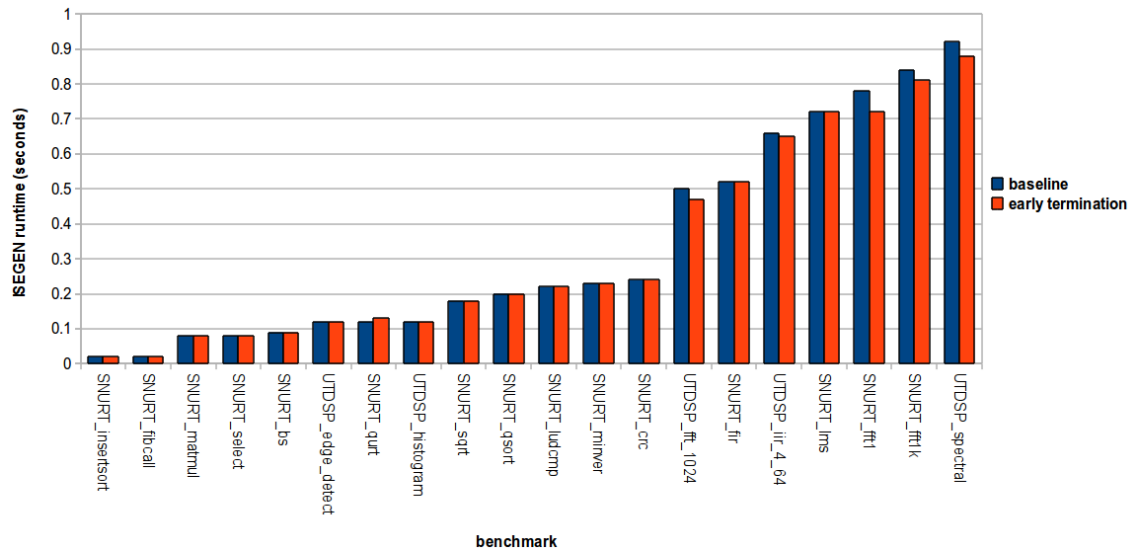


Fig. 5.7: Time taken per termination strategy, for benchmarks taking less than a second to process. These benchmarks are the least improved by early termination, because they contain the smallest DFGs. Like the larger UTDSP LPC benchmark of figure 5.6, SNURT QURT is made a little slower to analyse by early termination. The observation that most of these very short benchmarks are not worsened is vindicating to the early termination method. This graph demonstrates that the conditions for such ill performance are more idiosyncratic than just having small DFGs. Average saving is 0.01 seconds, with a standard deviation of 0.02 seconds.

encounter a condition where it is violating the input port constraint. The DFG are all so small that the calculation of the early termination condition does not add a measurable overhead to the time taken to run the *isegen* tool.

By far the largest benchmark tested here in terms of either number of DFG or lines of code, is the FAAD AAC decoder. It represents a realistic application, as it might be processed (without any source transformation) in order to quickly increase its performance through application specific processing. The largest executed DFG in the FAAD application is 154 nodes and 257 edges. Larger DFG exist in the source, but they all exist in dead code sections so far as the version configured for this work is concerned (fixed point). Of the 4324 DFG contained in the FAAD source code only 1194 of these are actually considered "live" by the *isegen* tool, since they have been profiled with an execution frequency greater than zero. The FAAD source code contains many potential configurations, and in this case the configuration used has only covered around a quarter of the source code available. This is of course a fully functional version of the FAAD decoder, and is configured in such a way as to mimic a version of the decoder that could be executed on the EnCore [10] processor. The early termination mode gives a 1.74x improvement (43% reduction) over the baseline ISEGEN runtime. This amounts to a reduction from 11:50 (m:s) to just 6:48, saving over 5 minutes. Since this version of FAAD has had no source transformations applied and is presented to the AISE tool-chain as-is, there is potential for loop-level transformations such as unrolling to be applied in order to increase the ISE acceleration from the achieved 1.67x. The early termination introduced makes the exploration more tractable; as graphs get larger through OLP-exposing transformations the reduction in tool runtime versus the baseline gets more pronounced, which encourages the exploration of these more fruitful spaces.

Whilst it would still be possible to generate a set of DFG which would strain the algorithm to produce a runtime complexity $\in O(|V|^3)$, it is demonstrated in these results that for realistic applications and benchmarks the runtime of the ISEGEN algorithm can be dramatically reduced from the worst case using early termination.

5.3.4 Conclusions

Having now thoroughly evaluated the effects of the new early termination methodology, the following conclusions are apparent:

- Early termination dramatically reduces the runtime of the ISEGEN algorithm; up to 7.26x in the best case, 5.82x on average.
- Early termination is generally more effective in both absolute and relative terms with larger node-counts:
- The size of an individual DFG has a higher-order contribution (cubic) to the runtime than the number of DFG (linear). Early termination applies to the former, and this is the reason for the growing relative and absolute efficacy with growing DFG node count.
- The mechanism by which early termination can fail to produce the same result as the baseline algorithm has been identified, however:
- At no point during evaluation did the early terminating and original algorithm differ in their result during this analysis.

Henceforth, in this thesis, the ISEGEN algorithm is only used with early termination enabled. The ability to explore several times more design points is worth more than the slight difference in output that the algorithm has to the baseline in rare circumstances, which were not even observed here. The approach outlined in this section has addressed well the concern of engineer time, as by making the algorithm faster the engineer's productivity can be increased by the same factor.

5.4 Pipeline Aware Identification

In this section we look at the issue of making the identification algorithm aware of the benefit in overlapping the execution of independent ISEs. *An extension to the ISEGEN heuristic to account for pipelining is proposed and evaluated, increasing acceleration by up to an additional 1.5x.*

5.4.1 When Serial is also Parallel

Pipeline or temporal parallelism is the overlapping of independent operations in a pipeline, after the processor hardware has determined that the serial instruction stream contains two or more operations which may be overlapped in time in order to reduce their overall execution time. Temporal parallelism accounts for a large degree of the performance available in both embedded and GPP cores. Different mechanisms exist for performing the necessary dependency checking, some of the earliest mechanisms being Score-boarding [142] (developed in 1964 for the CDC6600) and Tomasulo's Algorithm [143] (developed in 1967 for the IBM Model 91's FPU). Both of these mechanisms in modified forms have found use in modern-day processor cores, and so their relevance continues to this day. The common features of processors which make use of these techniques are:

- Multiple and/or multi-stage (pipelined) functional units.
- Functional unit result latency greater than a single cycle.

These features in themselves are common to nearly all processor cores, hence the reason why so many processors feature mechanisms for dynamically exploiting temporal parallelism.

In the context of ISE, there is a design-choice to be made with regards to the amount of pipelining applied to the microarchitecture implementing the ISEs. Purely combinational ISEs are implemented as multi-cycle functional units containing no synchronous registers. The same arithmetic function may also be implemented with a number of registers to allow for an issue latency or "data initiation interval" (DII) less than the critical path of the original ISE. The lowered DII allows for both temporal parallelism and resource-sharing of the AFU representing the ISE. Adding registers in this fashion is a trade-off between the area added by flip-flops to the design, the static and dynamic power required by those flip-flops, and the increase in acceleration which may result from pipelining.

There are a number of different ways temporal parallelism may be exploited by a processor, depending on microarchitecture:

- Where functional units (both baseline and extension) are implemented in non-overlapping microarchitecture, both may be active at the same time.

- Where functional units have a DII less than their critical path due to pipelining, multiple instructions may be "in flight" within the unit.

- Where functional units share microarchitecture spatially, if these units are pipelined the units may again have multiple instructions "in flight", as the resources should be arranged in such a fashion that successive instructions do not generate a structural hazard.

All of these scenarios require that there are not other hazards (data, control) preventing them from occurring. The latter point above covers the scenario represented by CFAs, where multiple ISEs are implemented in a single pipelined functional unit. ISEs sharing a single CFA may execute (in-order) with a single-cycle DII. Longer instructions may hold up shorter ones, and so scheduling is important in such a situation.

Scheduling itself is dependent on the operations available in the instruction set, including extensions. Other work in this field cites [144] the lack of repeating isomorphic ISEs as the reason for not performing pipelining on AFU; this is only true when ISE do not share resources and are data-dependent.

It is important not to confuse the different approaches to exploiting pipelining in ISE analysis, as the term "pipelining" has also been used to denote the use of multiple-cycle input and output to a combinational ISE ([18] [23]), otherwise known as "Distrubuted I/O" [129]. The work of [129] claims to remove the I/O constraint, but this is misleading as this simply converts the discrete single-cycle I/O constraints to a more continuous bandwidth constraint. The I/O constraint is indeed removed in the identification algorithm of [129]. Maximal ISEs are identified, regardless of their potential fit with the actual bandwidth available. This is better represented in the work of [145] where the bandwidth constraint of the register file is used to shape the search process, using a modified version of the Atasu *et al.*ILP-based AISE algorithm [20].

As we have already shown with the earlier section 4.3, splitting up a larger ISE into smaller partitions does not necessarily lead to a massive decrease in performance. The earlier work of [18] and [23] are both using pipelining to make ISEs *as large as possible*, with the general concept that "bigger is better". In the work presented in this section, we are simply looking to make better use of a smaller area through inter-ISE temporal resource sharing. Because we are not relaxing I/O constraints through pipelining here, but rather using pipelining to overlap consecutive ISEs in time, the ISEs in question do not get any bigger in general.

Modification of the ISEGEN algorithm is performed in this section, to include pipeline scheduling within its calculation of the merit function heuristic component *savedcycles*. The original algorithm including the merit heuristic functions can be seen in section 2.4.2. The *savedcycles* component is originally a direct embodiment of the model from section 2.3.1. It is a part of both the inner and outer merit calculations in the ISEGEN algorithm, and is used in isolation to determine the worth of a partition with respect to an induced cut. The

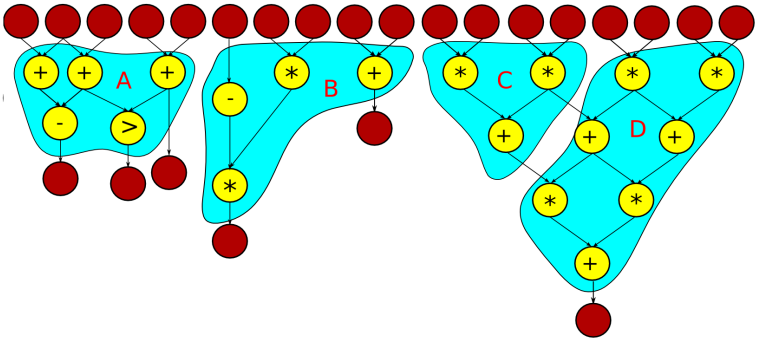


Fig. 5.8: Example DFG to demonstrate heuristics, containing four ISEs and no software nodes; ISE Hardware Latencies: A=2, B=4, C=3, D=8. Software latency: 35 cycles, Hardware (combinational) latency: 17 cycles. Tables 5.5 and 5.6 illustrate the scheduling of this DFG with regards to the ISE identified therein.

savedcycles heuristic is extended to include a pipeline model, and a scheduler including a modifiable scheduling heuristic. As each new ISE is explored, the entire basic block is scheduled each time the *savedcycles* heuristic is evaluated, in order to direct the search towards sets of ISEs which schedule better when combined. The process is somewhat greedy, as there is no backtracking once an ISE has been identified. The new heuristic should regardless provide ISEs which are better than their purely combinatorial counterparts for scheduling on a pipelined implementation.

The approach presented here can be seen as an alternative to distributed I/O, relying on further partitioning of ISEs and the pipelining and forwarding of the processor to increase the throughput of extensions. The increased partitioning allows for resource-sharing algorithms to trivially overlap hardware inter-ISE stage by stage, since the same pipeline stage will not be active at the same time for any two different ISEs in a single-issue architecture. This is one of the principal assumptions of the CFA construction algorithm covered in section 4.2.

5.4.2 Pipeline Model and Scheduling Heuristic

The pipeline model employed is based heavily on the operation of the EnCore pipeline, and the extension pipeline developed for the Castle revision (see section 7.2) of the EnCore microarchitecture. The model is comprised of the following rules:

- Only one operation (baseline or ISE) may be issued each cycle.
- Only one operation (baseline or ISE) may commit each cycle.
- Operations may not overtake one-another; no out-of-order execution is permitted.
- Baseline operations are not pipelined, and only one ALU of each type is available.
- ISE implementations are fully pipelined, and hence may be issued one per cycle.

Stage	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Fetch	B	C	A	D											
Decode		B	C	A	D										
Execute 0			B	C	A	D									
Execute 1				B	C	A	D								
Execute 2					B	C		D							
Execute 3						B			D						
Execute 4										D					
Execute 5											D				
Execute 6												D			
Execute 7													D		
Memory							B	C	A					D	
Writeback								B	C	A					D

Tab. 5.5: Pipeline Schedule for DFG using ASAP-LF heuristic (see figure 5.8). 15 Cycles total; 11 cycles in Execute Stages; 6 cycles saved over combinational due to allowing independent ISEs to overlap in their execution.

- ISEs may arbitrarily overlap in the pipeline where dependencies allow.
- Baseline and ISEs may not overlap.
- Baseline instructions may not overlap one another.
- Forwarding between dependent operations is immediate between the commit of the first instruction and the issue of the second.

Two separate heuristics are used to perform the pipeline scheduling:

- ASAP-SF: Operations are issued as soon as possible (ASAP) (e.g. when their inputs are available), and in the case where multiple operations are ready to be issued these are issued with the shortest latency first (SF).
- ASAP-LF: Operations are issued ASAP, and where multiple operations are ready these are issued longest latency first (LF).

An example schedule of ISEs and baseline instructions for a given DFG (See figure 5.8) as per the ASAP-SF heuristic is given in Table 5.6. The same DFG with respect to the ASAP-LF heuristic is given in Table 5.5.

Note that the model indicated above replaces entirely the model presented in section 2.3.1; both baseline and extended performance are processed with respect to this pipeline model and scheduler.

5.4.3 Determining the Efficacy of the Pipelining Heuristic

To show that the pipelining heuristic is actually useful, it is necessary to demonstrate that it is somehow superior to the existing combinational model used in the ISEGEN algorithm. It is possible that the existing combinational heuristic can be calibrated with a suitable heuristic

Stage	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Fetch	A	C	B	D											
Decode		A	C	B	D										
Execute 0			A	C	B	D									
Execute 1				A	C	B	D								
Execute 2						C	B	D							
Execute 3								B	D						
Execute 4										D					
Execute 5											D				
Execute 6												D			
Execute 7													D		
Memory					A		C		B					D	
Writeback						A		C		B					D

Tab. 5.6: Pipeline Schedule for DFG using ASAP-SF heuristic (see figure 5.8). 15 Cycles total; 11 cycles in Execute Stages; 6 cycles saved over combinational due to allowing independent ISEs to overlap in their execution.

weighting vector as covered in section 5.2 to derive good templates for pipelining. The software and hardware latencies used in this section have been modified from those in 5.2 to better reflect those of the Calton EnCore [10] implementation, a low-cost processor core. This therefore will bring the results obtained here closer to those which may be expected in the microarchitecture developed within the project that this work is contained in. This, and the requirement that the pipelining performance be measured also, requires a re-running of the parameter space exploration as covered in section 5.2. Three different runs are required in order to evaluate the new pipelining heuristic (and scheduling heuristic subcomponent):

- Combinational heuristic, including both ASAP-SF and ASAP-LF acceleration results for the ISEs generated in addition to the combinational model acceleration.
- Pipelining heuristic using ASAP-SF scheduling heuristic, including all three results (ASAP-SF, ASAP-LF, combinational) for acceleration.
- Pipelining heuristic using ASAP-LF scheduling heuristic, including all three results (ASAP-SF, ASAP-LF, combinational) for acceleration.

As per the methodology of section 5.2 results are sorted in order of acceleration, and the top $N\%$ is taken, increasing N until at least one common vector is found across all benchmarks. To determine whether or not the new pipelining heuristic is actually effective at guiding the search towards good pipelined ISEs, the value of N for the combinational heuristic but using the pipeline acceleration performance to order results is taken. The same measurement of N is taken for both the ASAP-SF and ASAP-LF scheduling heuristics to measure pipelined performance, but using the combinational model to actually guide the search. N represents the loss in acceleration required in order to produce an acceleration result within $N\%$ of the maximum observed across all benchmarks.

The values for N obtained when using the combinational heuristic with respect to both pipeline scheduling modes' acceleration represent the value to beat when using a pipeline heuristic in the actual search. The same process is therefore repeated with each of the pipeline heuristic modes, getting the N values for the combinational, ASAP-SF, and ASAP-LF accelerations when using each of the pipeline heuristic modes. If the value for N is lower when using the pipeline heuristic corresponding to the acceleration metric N is produced for, then the pipelining heuristic in question is more stable with regards to a static weighting vector.

N represents the stability of the heuristic with regards to the weighting vector. Higher N represents less commonality between the vectors obtaining the better results in the different benchmarks. N is not the complete picture with regards to the benefits of using this pipelining heuristic. Also of interest is the pipelined acceleration provided by the ISEs identified: both the best result seen regardless of N, and the result obtained with regards to N. These acceleration metrics must be taken for both ASAP-LF and ASAP-SF pipeline schedule models, and the combinational model for the search guided by the combinational model heuristic. For the searches guided pipelining heuristics, only the combinational model and relevant pipeline schedule model acceleration performance is taken. It is assumed that opposing pipeline models' heuristic and acceleration performance are not important in the latter analysis of the heuristic efficacy.

The same suite of benchmarks are used as in section 5.2, both to aid in comparison and because the infrastructure to run these benchmarks is already available due to this earlier work. Several observations can be made with this further data which are not directly relevant to the evaluation of the pipelining heuristics. Most interesting is the potential effect of changing the latencies of operations on the heuristic weighting vector; whether or not the best vectors seen remain the same, and whether the same value for N is required in order to locate a common vector for the combinational model heuristic.

Due to the observations made in the earlier section 5.2 regarding the linear equivalence of weighting vectors in the space analysed therein, those vectors which are equivalent have been reduced to remove all linearly equivalent duplicate vectors. The space in section 5.2 is 7776 points. However, the number of samples taken for each combination of benchmark and heuristic herein is only 4652 points. This has reduced the amount of computing required by about $1/3$.

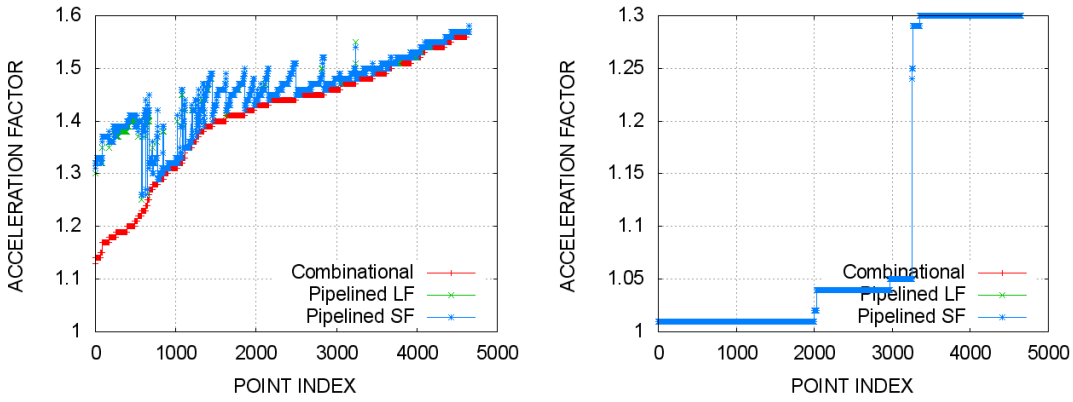
5.4.4 Pipeline Heuristic Results and Evaluation

The relationship between the heuristic used to guide the search, the weighting vector used, and the resulting acceleration in a particular model is a complex one. This relationship is the main focus of this evaluation. There are thirty-six graphs in this section, and each reflects a benchmark's heuristic space when the results are monotonically ordered by the performance model result obtained (combinational, pipelined longest first and pipelined shortest first). We

discover here that the pipelined heuristic requires new weighting vectors, that it is better suited to providing pipeline-aware designs than the original heuristic, and that pipelined designs significantly reduce the I/O required to reach a particular level of merit. These points are now discussed in depth, relating their discovery to the evidence obtained during this experiment.

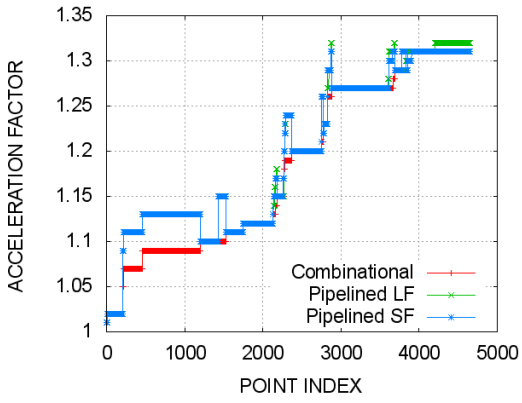
Figures 5.9 and 5.10 reflect the performance of the ISEGEN algorithm guided by the combinational heuristic, including the performance of the resulting ISEs when scheduled using the two pipeline models (LF and SF). As would be expected of the models used, the pipeline performance is always greater than the combinational performance. The combinational performance is the same as the pipelined performance where no overlap can be exploited. This equality is true of all heuristics and vectors when applied to the SNURT CRC benchmark at 8/8, since the I/O allows for all parallelism to be absorbed intra-ISE, giving no opportunity for the inter-ISE parallelism which would be exploited by pipelining. FAAD and SNURT FFT1K at I/O 8/8 are similarly effected, but in each case there are occasional points wherein the pipelined performance exceeds that of the combinational due to some degree of available inter-ISE parallelism. The pipeline heuristic only distinguishes itself from the original combinational heuristic when there is at least one ISE already identified in a DFG, and there are operations parallel to the already identified ISE(s). The new heuristic will then impart a merit boost to the speedup component of the ISEGEN heuristic. This boost will be equal to the additional cycles saved through scheduling the new operation overlapping with the original ISE(s). For the benchmarks and I/O settings noted as having little advantage from pipeline scheduling of the ISEs derived using the combinational heuristic, no significant advantage is yielded through using the new pipeline-aware heuristic. This can be observed for SNURT CRC in figure 5.12 and 5.14 for the larger 8/8 I/O, where there is absolutely no difference between the different measures of merit. Figures 5.11 and 5.13 for I/O of 4/4 demonstrate that the pipelined merit outweighs the combinational merit in some places. Ultimately the pipelined merit only reaches the same level (1.3x) that the combinational heuristic achieved in figure 5.9. Both FAAD and SNURT FFT1K at I/O 8/8 have the same issue, in that there is little to no inter-ISE parallelism to exploit if the intra-ISE parallelism is fully exploited; the top 5% of weighting vectors in both of these benchmarks in all speedup heuristics present points which are the same merit, pipelining or not. These benchmarks have not been transformed with any loop unrolling, and represent a set of smaller DFG as might be present in an application.

FAAD and FFT1K present a different picture at I/O 4/4, where the positive effects of pipelining begin to show. At lower I/O, the potential for inter-ISE parallelism is increased because less of the width of data-parallel DFG can be incorporated into a single ISE. Figure 5.9 demonstrates that with the combinational speedup heuristic and using pipeline scheduling on the resulting ISEs, performance is generally a little better when pipelining even at the top 5% of weighting vectors. At the high end, performance is improved by less than one percent in FAAD (1.57x - 1.58x), and by a similarly marginal amount in SNURT FFT1K (1.31x - 1.32x),

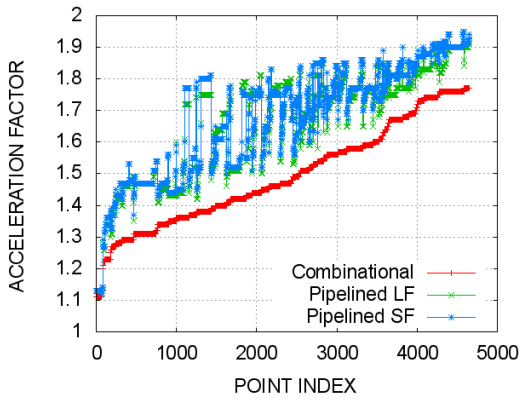


(a) FAAD AAC

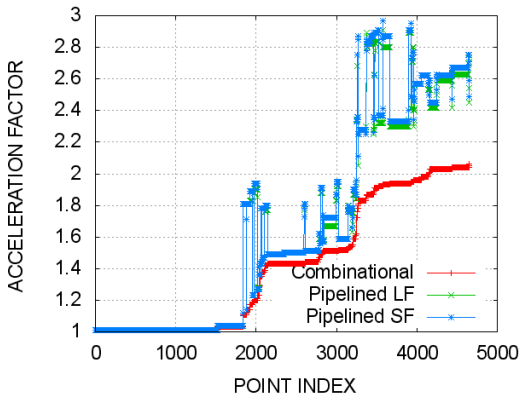
(b) SNURT CRC



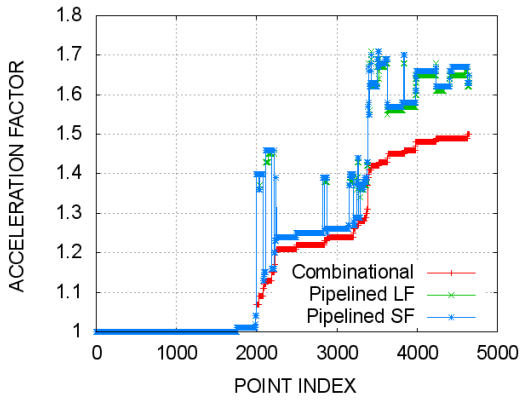
(c) SNURT FFT1K



(d) SNURT JFDCTINT

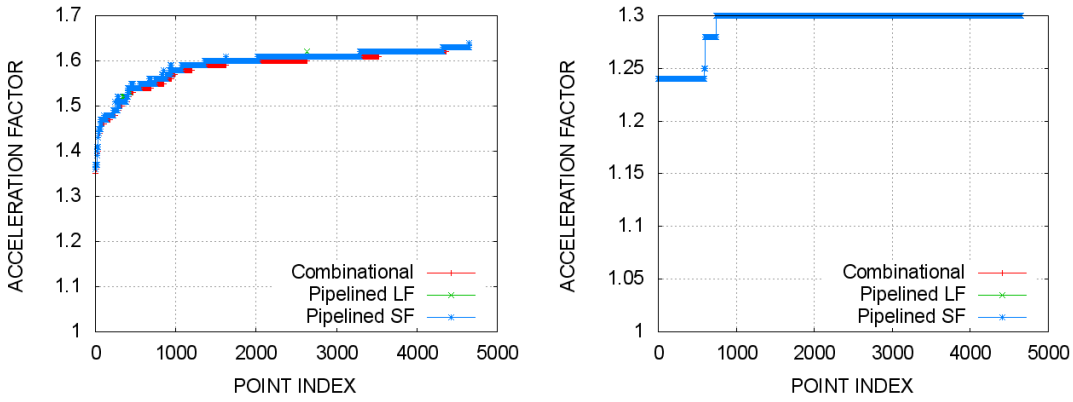


(e) UTDSP FIR



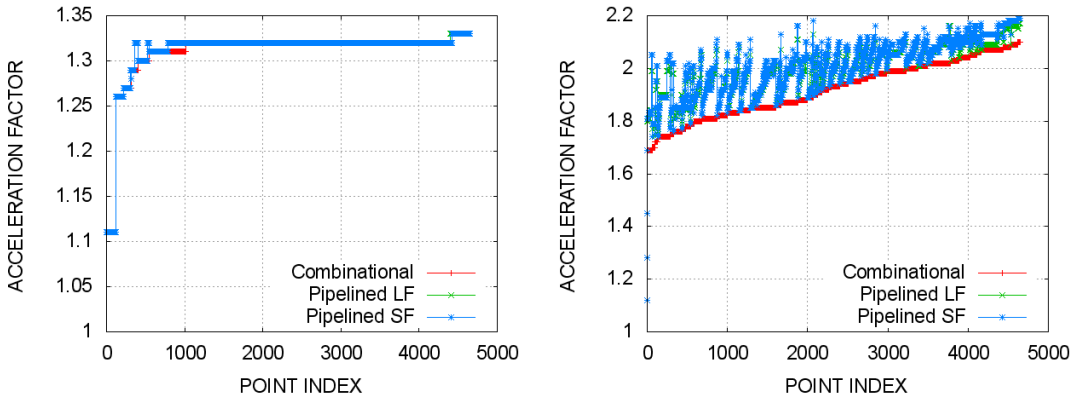
(f) UTDSP LMSFIR

Fig. 5.9: Parameter Sweep Space (I/O: 4/4) With Original Heuristic, Monotonically Ordered by Speedup Factor on Combinational Performance Samples. These graphs demonstrate the relationship between increasing combinational performance, and the performance of the same design point when scheduled with pipelining. For every "step" of acceleration there are a number of different pipelined performances. This implies that a more complex heuristic is required to properly exploit pipelining. The potential of the approach is considerable where OLP is also, adding up to 1.5x additional acceleration in the case of FIR. Less data-parallel benchmarks such as CRC see no benefit.



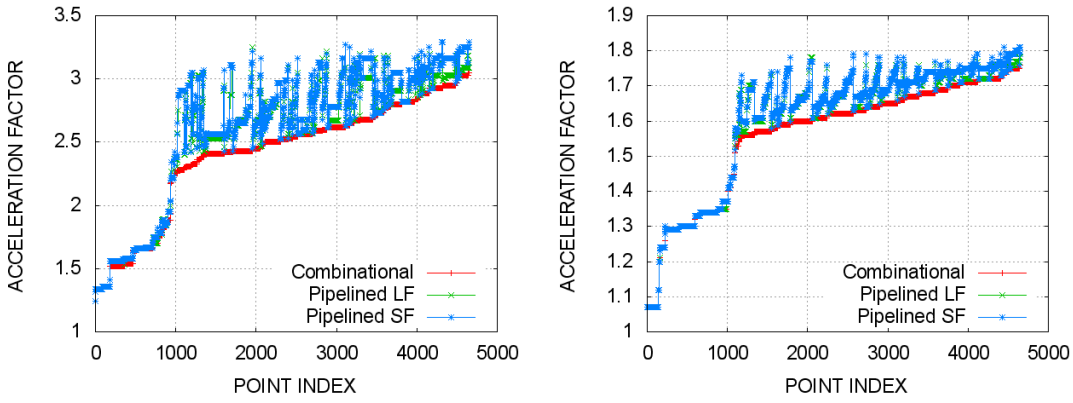
(a) FAAD AAC

(b) SNURT CRC



(c) SNURT FFT1K

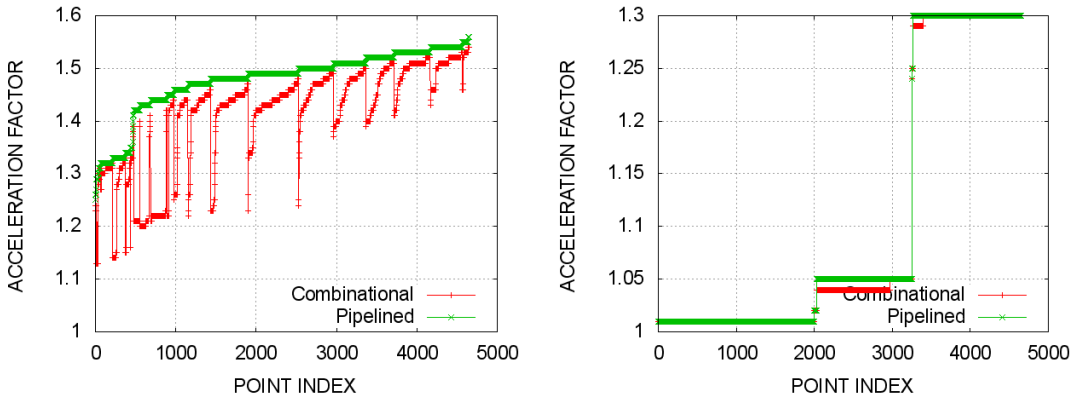
(d) SNURT JFDCTINT



(e) UTDSP FIR

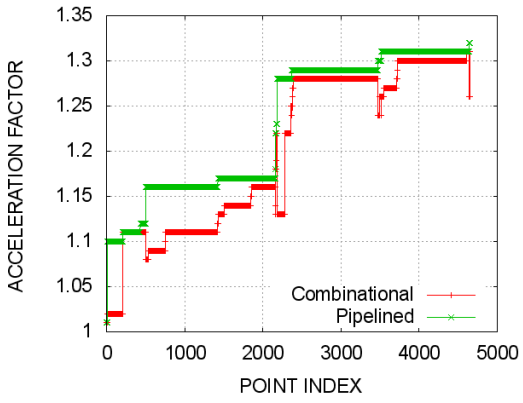
(f) UTDSP LMSFIR

Fig. 5.10: Parameter Sweep Space (I/O: 8/8) With Original Heuristic, Monotonically Ordered by Speedup Factor on Combinational Performance Samples. These graphs demonstrate the same effect as in figure 5.9 but with a higher I/O constraint of 8/8. In particular we can see that the difference between the combinational and pipelined performance has been considerably reduced because more OLP is absorbed intra-ISE rather than inter-ISE. A benefit still exists with using pipelining at this level, as the peaks for pipelined performance are still higher than those for combinational.

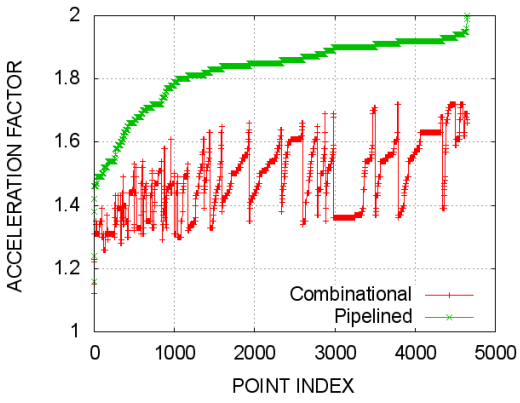


(a) FAAD AAC

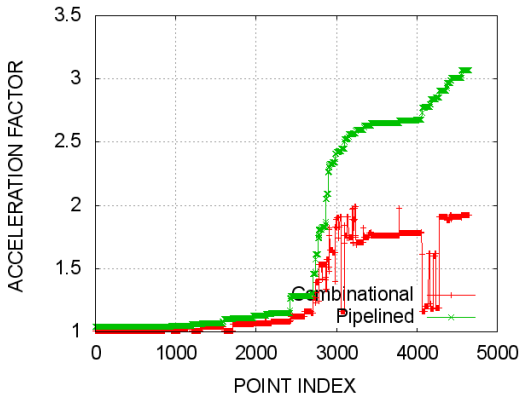
(b) SNURT CRC



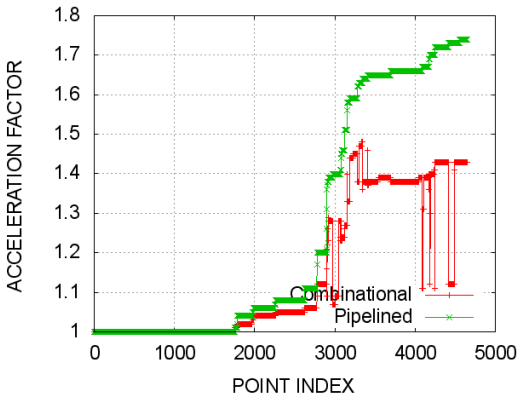
(c) SNURT FFT1K



(d) SNURT JFDCTINT



(e) UTDSP FIR



(f) UTDSP LMSFIR

Fig. 5.11: Parameter Sweep Space (I/O: 4/4) With ASAP-SF Pipeline Heuristic, Monotonically Ordered by Speedup Factor on ASAP-SF Pipeline Performance Samples. These graphs demonstrate once again the non-linear relationship between the combinational and pipelined schedule, caused by the presence or lack of dependencies inter-ISE. Where dependencies exist between two ISEs, the difference between combinational and pipelined performance is negated. This effect summed over all the ISEs in each benchmark account for the non-linearity. The pipelined heuristic has a slightly higher ceiling of acceleration for pipelined design points than the original heuristic from figure 5.9.

definitely not a robust enough improvement to motivate the technique for these smaller DFG. Figures 5.11 and 5.13 demonstrate that when using the pipeline heuristic to actually guide the search roughly the same results are obtained. For these smaller DFG the use of any pipeline scheduling for guiding search or for just exploiting the result of search is fairly useless. The use of the pipelining heuristic does not damage the search at least.

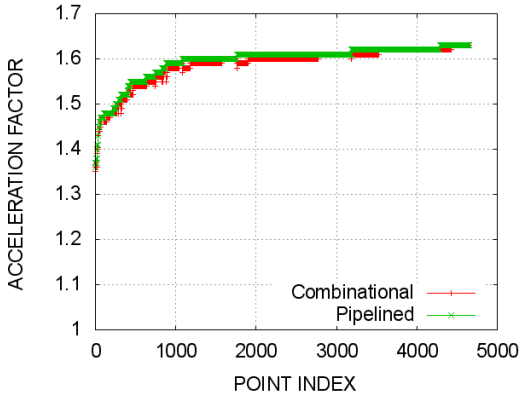
The benchmarks containing a larger degree of operator level parallelism present a far more interesting picture with regards to the difference between the pipeline-aware and original speedup heuristic. SNURT JFDCTINT, and UTDSP FIR and LMSFIR all contain wide DSP-like kernels, performing a large amount of data-parallel operation. For this reason, in both I/O settings, the gap between the pipelined and combinational ISE is significant. The inter-ISE parallelism becomes more readily exploitable at higher I/O. Most interesting regarding these results is the tendency of the combinational and pipelined ISE performance to diverge more when using the pipelined heuristic for speedup than when using the combinational. The combinational performance at the high end of the pipeline-heuristic weighting vectors is consistently lower than it is at the high end of the combinational heuristic weighting vectors, and the pipeline scheduled performance is higher. This effect is not limited to the high end of the weighting vectors either. The SNURT JFDCTINT benchmark presents the most subtle diverging of the two series. It is still significant enough to demonstrate that the pipelined speedup heuristic does impart an advantage in selecting better ISEs for pipelined implementation.

The greatest performance improvement for the combinational heuristic guided SNURT JFDCTINT 4/4 ISEs are 1.77x for the combinational schedule, 1.92x for the ASAP-LF pipelined schedule, and 1.95x for the ASAP-SF pipelined schedule, as in figure 5.9. When using the ASAP-LF heuristic, the maximum combinational schedule performance drops to 1.71x but the ASAP-LF pipelined schedule performance rises to 1.93x, as in figure 5.13. More significantly, when using the ASAP-SF heuristic, the maximum combinational schedule performance drops to 1.72x but the ASAP-SF performance rises to 2.00x, as in figure 5.11. This observation confirms the idea that optimising directly for combinational performance does not lead to pipeline-friendly ISEs, and that the performance of pipelined ISEs if utilised in a combinational context will be bad. The two heuristics' objectives do not correlate well enough to assume that a well-performing solution with respect to one will produce good results for the other. This in itself motivates the use of a speedup merit heuristic specific to pipelining. Further, the ASAP-SF heuristic is superior to the ASAP-LF heuristic in the pipeline model used herein for the SNURT JFDCTINT benchmark. Differentiation between the two heuristics in acceleration is less pronounced but still significant for SNURT JFDCTINT when considered for the I/O constraints of 8/8, where the potential for pipelined (inter-ISE) parallelism to be exploited is reduced. Under the combinational heuristic, the best performance is 2.10x for the combinational schedule, 2.17x for the ASAP-LF schedule and 2.19x for the ASAP-SF schedule (figure 5.10). Under the ASAP-LF heuristic, the best performance is 2.09x for the combinational schedule and 2.19x

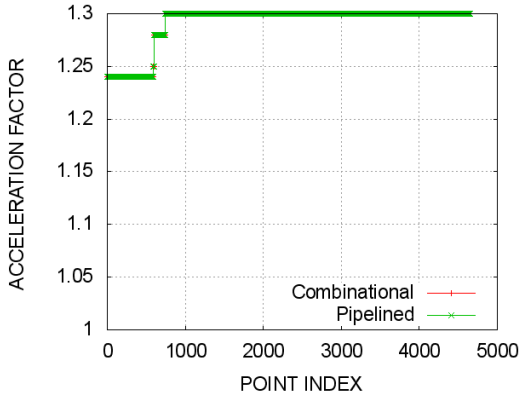
for the ASAP-LF schedule (figure 5.14). Under the ASAP-SF heuristic the best performance is 2.10x for the combinational schedule and 2.22x for the ASAP-SF schedule (figure 5.14). Again, the use of the appropriate pipeline schedule heuristics in search contributes to a slight worsening of the ISEs for combinational schedule, and a slight improvement in the ISEs for pipeline scheduling. The scheduling of ISEs shortest-first (ASAP-SF) again performs better than the longest-first alternative under this larger I/O constraint.

Greater advantages at higher I/O constraints can be seen in the UTDSP FIR and LMSFIR benchmarks, which share similar structure in their computational kernels. Both benchmarks have a considerable degree of OLP, and contain large graphs which hold significant potential for exploiting both intra- and inter-ISE parallelism. In the best case for the original combinational heuristic, the FIR benchmark at an I/O of 8/8 obtains 3.05x acceleration for the combinational schedule, 3.25x for the ASAP-LF schedule, and 3.29x for the ASAP-SF schedule. The same benchmark and I/O under guidance of the ASAP-LF heuristic obtains a maximum of 2.99x acceleration for the combinational schedule and 3.34x for the ASAP-LF schedule. Under guidance of the ASAP-SF heuristic, the combinational schedule reaches 2.97x and 3.34x for the ASAP-SF schedule. The LMSFIR benchmark with the combinational heuristic reaches 1.76x acceleration for the combinational schedule, and 1.81x for both the ASAP-LF and ASAP-SF schedules. Using the ASAP-LF heuristic acceleration is 1.72x for a combinational schedule, and 1.82x for the ASAP-LF schedule. With the ASAP-SF heuristic acceleration is 1.72x for the combinational schedule, and 1.81x for the ASAP-SF schedule. Once again, the use of pipeline-aware heuristics either maintains or improves the performance of the ISEs under ASAP-LF and ASAP-SF scheduling. The combinational schedule does have more trouble exploiting the ISEs in question, but the negative impact is generally by around the same amount that the pipelined result improves by. This technique is not intended to be used for combinational ISE scheduling anyway; rather it is to be used when extending for architectures having a pipelined implementation of ISEs, or when deciding whether the inclusion of pipeline registers is worthwhile when designing an ASIP.

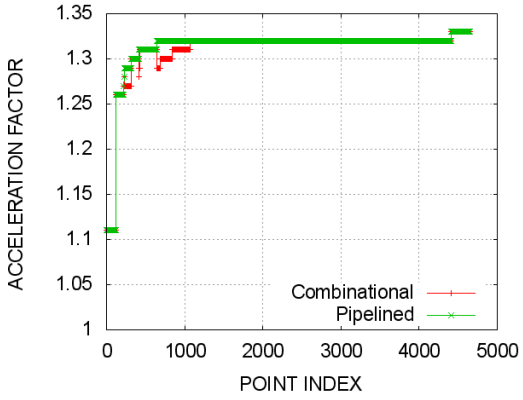
Nearly all of the benchmarks show the greater efficacy of pipeline-aware scheduling and search heuristic at lower I/O levels, and the UTDSP FIR/LMSFIR benchmarks show the greatest benefit of all. At an I/O constraint of 4/4 guided with the combinational heuristic the FIR benchmark obtains a maximum acceleration of 2.06x for the combinational schedule, 2.91x for the ASAP-LF schedule, and 2.97x for the ASAP-SF schedule. Under the ASAP-LF heuristic the combinational schedule reaches 1.98x acceleration, and the ASAP-LF schedule reaches 3.03x. The ASAP-SF heuristic obtains 1.99x acceleration under the combinational schedule, and 3.07x acceleration under the ASAP-SF schedule. Similarly, the LMSFIR benchmark under the combinational heuristic at an I/O of 4/4 achieves 1.50x acceleration in a combinational schedule, and 1.71x acceleration in both ASAP-LF and ASAP-SF schedules. Using the ASAP-LF heuristic LMSFIR reaches 1.47x under combinational scheduling, and 1.70x using ASAP-



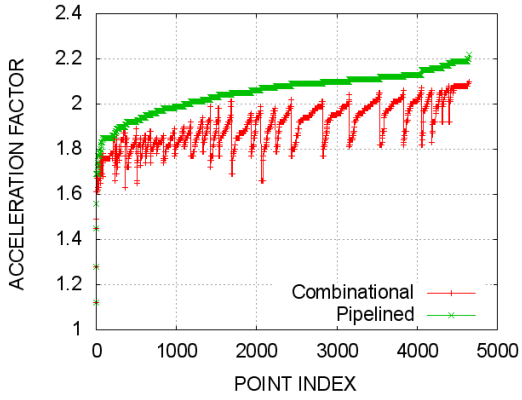
(a) FAAD AAC



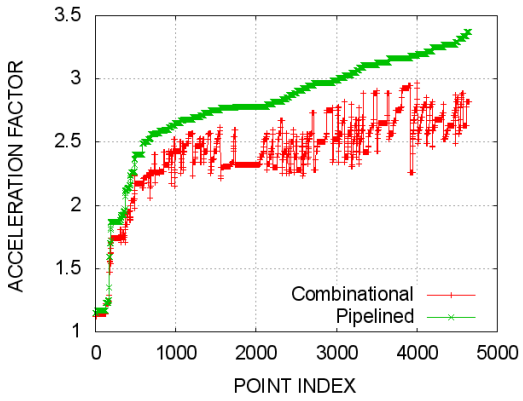
(b) SNURT CRC



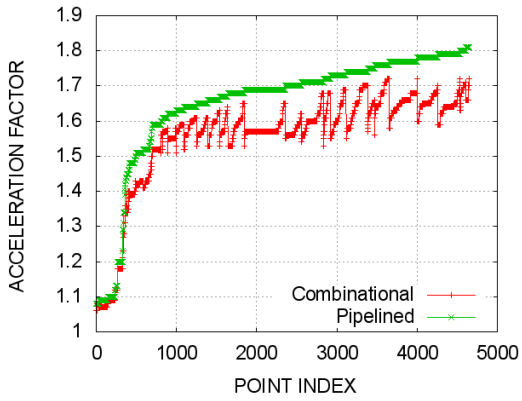
(c) SNURT FFT1K



(d) SNURT JFDCTINT



(e) UTDSP FIR



(f) UTDSP LMSFIR

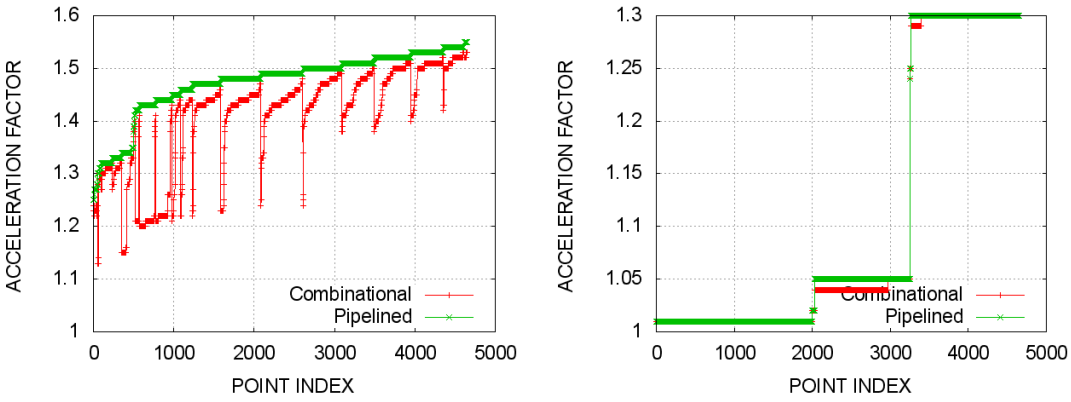
Fig. 5.12: Parameter Sweep Space (I/O: 8/8) With ASAP-SF Pipeline Heuristic, Monotonically Ordered by Speedup Factor on ASAP-SF Pipeline Performance Samples. As per the previous comparison between figures 5.9 and 5.10, these graphs should be compared to those of figure 5.12. The comparison again demonstrates reduced efficacy of pipeline scheduling where a higher I/O absorbs OLP intra-ISE rather than pipelining absorbing it inter-ISE.

LF scheduling. With the ASAP-SF heuristic the maximum acceleration under combinational scheduling is 1.48x, and under ASAP-SF scheduling is 1.74x. Once again, the pipeline-aware heuristic reaches the best performance using ASAP-SF.

The relationship between the combinational and pipelined performance is not well correlated because of the two components which add to the ultimate acceleration obtained in each scheduling mode. With purely combinational scheduling, only the intra-ISE parallelism contributes to the acceleration imparted by a set of ISEs within a basic block. With pipelining, both intra- and inter-ISE parallelism contribute, wherein a dependency between two ISEs immediately removes any potential for inter-ISE parallelism. The non-linear relationship between combinational and pipeline scheduled ISEs visible in the graphs presented throughout this section are a result of this second order effect. Strongly accelerating ISEs under a combinational schedule tend towards wider operation-parallel individual ISEs, and generally larger ISEs because the inclusion of a dependency between two ISEs does not damage acceleration. Strongly performing ISEs under a pipelined schedule tend towards less width, and a generally smaller number of nodes covered than combinational ISEs. This leads to the tendency to have more, and smaller ISEs identified under the pipeline heuristic than the combinational. These exploit available OLP through a mixture of intra- and inter-ISE parallelism, whereas combinational ISE can only exploit OLP intra-ISE. Where I/O constraints are lower, the combinational ISEs have a harder time exploiting the OLP, which accounts for the deterioration of the quality of combinational ISEs under stringent I/O constraints. Pipelined ISEs on the other hand allow for the available OLP to be exploited inter-ISE by overlapping ISEs in time. Exploiting OLP inter-ISE has overhead in comparison to intra-ISE, especially in single-issue architectures since the issue and commit of instructions must take structural hazards into account.

Examining the maxima in the weighting vector spaces is useful as it demonstrates the maximum efficacy of the ISEGEN algorithm when used with different heuristics. A more pragmatic measure of the efficacy of using the new heuristics to exploit inter-ISE parallelism is observation of the maximum acceleration obtained when using the same weighting vector across all benchmarks. A dynamic approach to setting the weighting vector could achieve the maximums already presented, assuming a correlation between the weighting vector and features of the DFG analysed could be found and exploited. Since this is not the purpose of this work, the common-vector approach must be taken instead.

As outlined in the methodology, the weighting vectors which obtain a result within the top $N\%$ of the acceleration observed throughout the vector weighting space for all benchmarks are the common weighting vectors. The minimum value of N required to obtain common weighting vectors is a measure of how stable the heuristic is across different programs; generally speaking, lower N is better as it means the heuristic is more *general*, and be better for a wider array of benchmarks. That the new pipeline heuristics and scheduling lead to a higher merit for the resulting instructions has been established. Determining the stability, or efficacy of the



(a) FAAD AAC

(b) SNURT CRC

(c) SNURT FFT1K

(d) SNURT JFDCTINT

(e) UTDSP FIR

(f) UTDSP LMSFIR

Fig. 5.13: Parameter Sweep Space (I/O: 4/4) With ASAP-LF Pipeline Heuristic, Monotonically Ordered by Speedup Factor on ASAP-LF Pipeline Performance Samples. Comparison of these graphs with those of figure 5.11 shows that the difference in performance between LF and SF is small in terms of absolute performance. LF and SF scheduling do present different trends, demonstrating a difference between the ISE design points being produced. As discussed in the text, the LF scheduling presented here is less effective than SF. The difference in absolute performance is slight, but the SF heuristic has better stability.

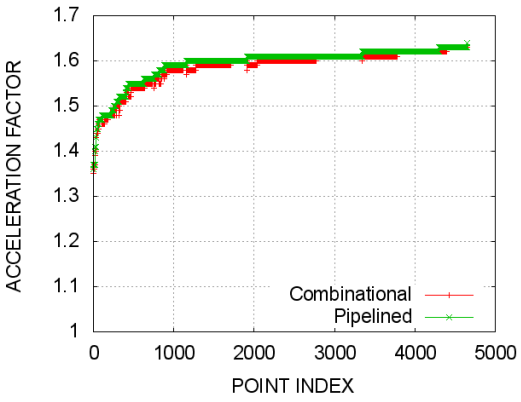
new heuristics across multiple benchmarks, requires calculation of N.

Interestingly, with the slightly different node latencies of this section versus those in section 5.2 the combinational heuristic common weighting vectors change, indicating that the weighting vectors are dependent on the relative software and hardware latencies of operations used in analysis. In prior results there were a large quantity of common weighting vectors and the minimum N required in order to obtain a result for combinational heuristic and schedule was $N = 14\%$. In these results, there is only a single common weighting vector and the minimum $N = 10\%$. This is not directly relevant to testing the efficacy of the pipeline heuristics, but it does imply that the original heuristic's stability is dependent on node latencies. This confirms conclusions made in section 4.2, that there cannot be a single optimal static weighting vector. Different DFG structures have been demonstrated in section 4.2 to impact the optimal heuristic vector, and now different node latencies have also. This further motivates the future investigation of a dynamic approach to the weighting vector.

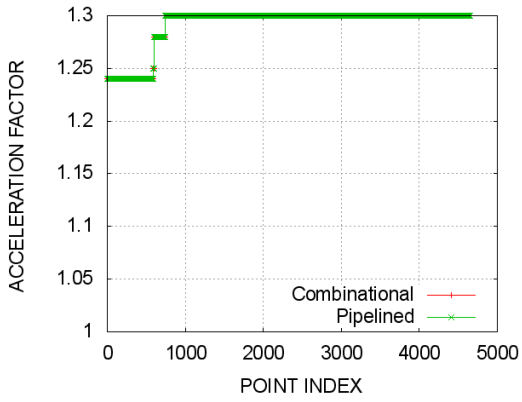
For the combinational heuristic with regards to the ASAP-LF schedule, common vectors were obtained at $N = 11\%$; with the ASAP-SF schedule, common vectors were obtained at $N = 9\%$. These are roughly the same as the N for the heuristic with regards to the original combinational heuristic, indicating that by the selection of weighting vectors the original combinational heuristic can be used to identify instructions which are amenable to pipeline implementation. The vectors identified for the combinational and pipeline schedules are different, meaning that the objective is distinct in purpose. The most significant difference between the common vectors identified is that in all cases of the common vectors identified for the pipeline schedule, the "large cut" heuristic is weighted as zero, meaning that it is effectively removed from analysis. This is in keeping with the earlier observation that the more efficacious pipeline scheduled ISEs tend to be smaller, and more plentiful.

For the ASAP-LF heuristic with regards to the combinational schedule, common vectors were obtained at $N = 19\%$, indicating that as would be expected the heuristic which favours pipelined solutions does not lead to good acceleration when the resulting ISEs are used in a combinational schedule. The ASAP-LF heuristic used with the ASAP-LF schedule, however, leads to $N = 13\%$, which is initially discouraging. Even with a maximum loss of 13% from the maximum obtained in each vector space, the resulting acceleration with regards to ASAP-LF scheduling is still better than the combinational heuristic. For example, FIR obtains 3.01x ASAP-LF acceleration using the common vectors of the combinational heuristic at $N = 11\%$, whereas it obtains 3.13x using the common vectors of the ASAP-LF heuristic at $N = 13\%$. Even though the common vector loses a little more of the maximum obtainable with the ASAP-LF heuristic rather than the combinational heuristic, the results are still consistently better.

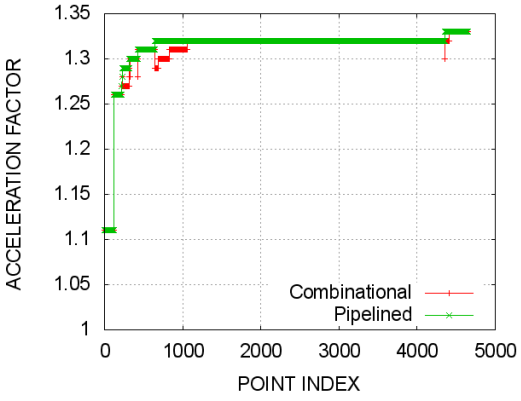
The ASAP-SF heuristic has common vectors for the combinational schedule at $N = 15\%$, which again is considerably higher than the original combinational heuristic; the use of pipeline aware heuristics where combinational implementation is to be used is therefore a consider-



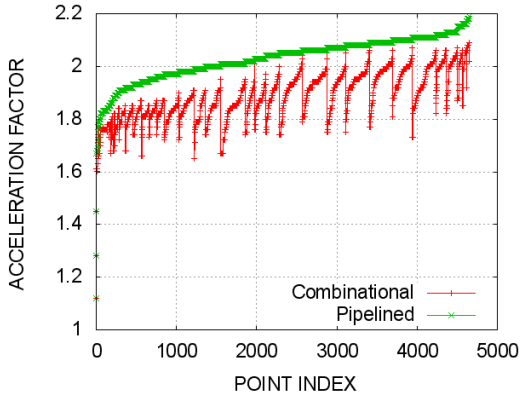
(a) FAAD AAC



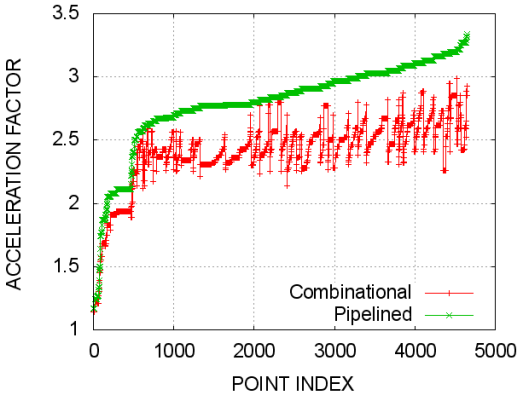
(b) SNURT CRC



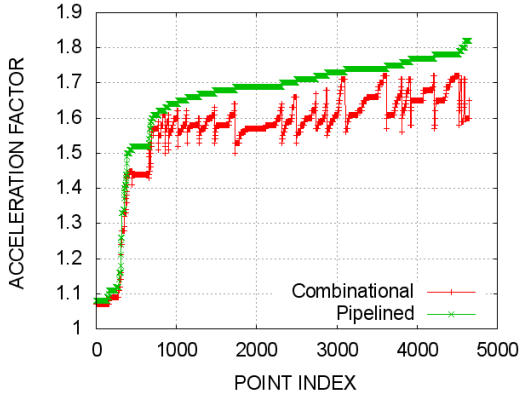
(c) SNURT FFT1K



(d) SNURT JFDCTINT



(e) UTDSP FIR



(f) UTDSP LMSFIR

Fig. 5.14: Parameter Sweep Space (I/O: 8/8) With ASAP-LF Pipeline Heuristic, Monotonically Ordered by Speedup Factor on ASAP-LF Pipeline Performance Samples. This should be compared with the graphs of figure 5.12, to demonstrate the difference between SF and LF scheduling. The difference is lower at this higher I/O than that between figures 5.11 and 5.12 due to the inter-ISE parallelism being less significant. At higher I/O OLP is first absorbed through parallelism intra-ISE, leaving less OLP for this pipelined method to exploit inter-ISE.

ably bad idea, as not only are the maximum accelerations obtained considerably less, common weighting vectors make the situation even worse due to a larger value of N. The ASAP-SF heuristic when used with the ASAP-SF schedule presents a very encouraging value for $N = 6\%$, Not only does the ASAP-SF heuristic and schedule achieve the greatest maximum acceleration, but it also has common vectors for weighting the ISEGEN merit function that obtain within 6% of those maximums. For example, FIR achieves a 3.01x acceleration using the combinational heuristic common vectors at $N = 9\%$ and ASAP-SF schedule, whereas the ASAP-SF heuristic achieves 3.25x using the common vectors at $N = 6\%$.

Looking at the results for this technique we can see that the pipelined technique does not have as much of a lead compared to the original combinational technique at the higher I/O. With higher I/O comes higher exploitation of the OLP available in a given applicatio DFG intra-ISE. That is, a single ISE is better able to exploit the OLP available as it is able to be wider. Whether or not pipelining will impart benefit at even higher I/O will depend entirely on how much OLP is available to be absorbed by ISE: If enough OLP exists that a multitude of high-I/O ISEs may be allowed to overlap in their execution, then pipelining will be beneficial. It is not sufficient to say that this technique would or would not be effective at higher I/O, but rather than for a given I/O there will be a related level of OLP required for pipelining to be of use. As a trivial limit, the OLP must require at least two ISEs (with respect to the I/O constraint) to exploit fully in order for this pipelined identification to be of increased efficacy versus the original technique.

5.4.5 Conclusions

After evaluating this new heuristic for ISE identification in the face of a pipelined microarchitecture, the following conclusions can be made:

- The new pipelining heuristic is more effective than a properly tuned version of the original combinational heuristic in deriving independent and hence instruction-parallel ISEs for overlapping within a single DFG. This is because:
  - The original (combinational) heuristic does not achieve as large a maximum performance under the pipelined schedule regardless of vector; although this is slight (less than 10% relative in all cases) it is consistent in that no benchmark has the opposite effect.
  - The "stability" of the pipelining heuristic when used in ASAP-SF mode, i.e. the maximum amount in percent one must lose from the total performance of any single benchmark in order to obtain a common static weighting vector is only 6%, which is the best seen so far. The ASAP-LF heuristic has a rather higher value of 13% which contends its suitability, but it still produces better results than the original heuristic.
- Inter-ISE parallelism is significant, contrary to assertions made in other efforts. At 4/4 the benefit conferred through using pipelined ISE is up to an additional 1.5x (raising acceleration from 2x to 3x for the OLP-rich FIR benchmark).
- Work which has only attempted to derive benefit from overlapping ISEs with higher (8/8 and above) I/O constraints will see little benefit from this approach unless sufficient additional OLP is exposed:
- There is a finite amount of OLP available in a given DFG analysed for ISE identification, and this "slack" may be taken up by either a lower I/O such as 4/4 with overlapping (pipelined) ISE microarchitecture, or a higher I/O without.
- There is no strong correlation between pipelined and combinational acceleration merit for a given weighting vector. The relationship is highly nonlinear due to structural issues not visible in the graphs produced. Sets of ISEs with greater numbers of inter-ISE dependencies will have lower pipelined acceleration; the combinational model of scheduling is not so affected.
- This pipeline-aware heuristic is also an alternative (in terms of exploiting lower I/O better) to the previously suggested multi-cycle I/O exploitation of [23]. The approach presented in this section should be cheaper in terms of algorithm run-time because as the

authors of [23] state, their approach requires multiple executions of the ISEGEN algorithm; generally sixteen or more. This is because their work did not actually change the ISEGEN algorithm itself, but rather the context in which it is used.

- The ASAP-SF schedule largely beat the ASAP-LF schedule in terms of maximum performance, and in addition to the greater stability of that heuristic under a static common weighting vector it is safe to say that for our configuration this is the more efficacious scheduling heuristic.

- Using the pipeline-aware heuristic when the end result is not to be temporally parallel (i.e. is the basic combinational microarchitecture) is detrimental to the result: This is because the pipelining heuristic favours inter-ISE parallelism, whereas the standard combinational heuristic favours intra-ISE parallelism. The objectives of identifying ISEs for pipelined and non-pipelined microarchitectures are different.

Pipelining is not used in the remainder of this thesis, as another objective (energy efficiency in the combinatorial model) is first examined and found to be of more interest in terms of the potential benefit to the result. This section has proven that there is considerable merit to this approach, and it warrants further attention.

5.5 Energy Aware Identification

Earlier in this thesis (section 4.2), it was determined that there is a near-linear relationship between the area occupied by an instruction and the energy it consumed in operation. The relationship was expanded to a heuristic, which it was claimed could be used in order to produce more energy-efficient ISE implementations, when realised via the CFA microarchitecture. In this section, it is demonstrated that the relationship between area and power can indeed be exploited. ISEs result which are more capable of shrinking the integral of runtime versus power, leading to an overall decrease in energy consumption. *This energy-aware heuristic reduces the energy used by a CFA implementation of a set of ISEs by an average of 1.6x, up to 3.6x.*

5.5.1 Better Value ISE: Making ISEGEN Optimise for Energy

In the earlier section 4.2, a near-linear relationship was determined between the silicon area consumed by a CFA and the power it would consume during execution. Of particular interest is table 4.1, which demonstrates that the specific values for a range of benchmarks, with regards to the number of milliwatts consumed per mm^2 of die area. The variability present in the table is due to the difference in power consumption between floating point and integer operators, wherein the benchmarks utilising only integer operations get at most 25% better power performance per mm^2.

The following relationship was determined to be a suitable heuristic for introduction into the DSE process used in order to design application-specific CFAs, from section 4.2.2:
C_{ise_sw}: Cycles taken to execute this ISE in software.
C_{ise_hw}: Cycles taken to execute this ISE in hardware.
P_{sw}: Combined dynamic and static power consumption for the baseline processor during execution.
P_{sw_cg}: Power consumption for the baseline core whilst CFAs are active (reduced compared to P_{sw} due to clock-gating).
P_{hw}: Power consumption of CFAs during ISE execution.
P_{hw_cg} : Power consumption for CFAs when not executing ISEs.

$$C_{ise\_sw}/C_{ise\_hw} \geq (P_{hw} + P_{sw\_cg})/P_{sw}$$

Any ISE which satisfies this inequality should contribute a benefit to energy efficiency, with respect to the model described in section 4.2.2. In order to extend ISEGEN with this heuristic, it is used to replace the existing $M(C)$ *speedup* merit heuristic which is used in both the weighted and unweighted merit calculations of ISEGEN, with a new heuristic:

$$energy\_saved = (C_{ise\_sw}/C_{ise\_hw}) - ((P_{hw} + P_{sw\_cg})/P_{sw})$$

That is: the difference between the ratio of software to execution time, minus the ratio of the power consumed during ISE execution to that consumed during baseline instruction execution. A larger number indicates a greater energy saving, through a reduction in the integral of the power versus time function of the architecture.

In order to make the *isegen* tool able to estimate P_{hw}, a number of derivations must be made from any DFG cut (ISE).

First, the CFA representing the ISE must be modeled including the configuration memories, permutation and routing layers, and pipelined operators as appropriate. A CFA would normally be constructed to cover multiple ISEs. For this heuristic, the model is instantiated to cover only a single ISE. Afterwards identification, selected ISEs are combined into a smaller number of CFAs by the *uarchgen* tool. The power estimates will have wandered somewhat after *uarchgen*, due to resource-sharing removing the 1:1 mapping of ISEs to CFAs.

Second, the area of the constructed CFA must be estimated; a process already contained within the *uarchgen* tool which has been provided with the areas of the individual components of the CFA, parameterisable with dimensions where this is appropriate (e.g. the permutation and routing networks, and the MCU configuration memories). The areas of the units used in a particular CFA design are added together to produce an estimate of area, which has been found to be an accurate (to within 15% for designs evaluated) estimate of the area consumed after the RTL is converted to gates. This area is the gate area, so is the die area consumed if the utilisation of the finished design were 100%. For the purposes of this work, the area is simply used to predict the power used by the CFA when active. As discussed, a correlation between the area consumed by a CFA and its dynamic power has been established earlier in this work in section 4.2. Table 4.1 gives the ratio of area to dynamic power for a range of benchmarks tested in the same 130nm process for which the component areas have been provided to the CFA construction process. The parts of *uarchgen* which are concerned with construction of CFAs and calculating area have been copied into the *isegen* tool. The augmentation makes *isegen* able to construct a CFA object model for and estimate the area of any valid DFG cut when implemented as a CFA. The average value of the correlation from area to power is taken from the results in table 4.1, and multiplied by the area in order to derive the estimate for power used as P_{hw} in the new heuristic. The other values in the heuristic are either available from existing functions within the *isegen* tool (e.g. C_{ise_sw} and C_{ise_hw}), or are provided as static values to the analysis (e.g. P_{sw} and P_{sw_cg}).

5.5.2 Determining the Efficacy of the CFA Energy Optimisation Heuristic

In order to determine whether or not the new heuristic is effective, comparisons must be made between the cost and the benefit provided by the original heuristic and this new energy optimisation heuristic. This does mean that once again the weighting vector sweep must be performed

using the new heuristic so that the best weighting vector amongst those tested can be located. For this purpose, the ECDF "Eddie" cluster is used again as per the methodology of sections 5.2 and 5.4. All 4652 linearly distinct points in the weighting space are evaluated using the new energy-aware heuristic in place of the original. Both 4/4 and 8/8 I/O constraints are explored, and the benchmarks used are identical to earlier experiments utilising "Eddie", as these represent a good cross-section of functionality. Since the software and hardware latencies used in this experiment are identical to those used in the earlier pipeline heuristic experiment (section 5.4) the weighting space results for the combinational heuristic are re-used here so that redundant work is not performed on the cluster. The same tools are used for this stage as in previous experiments; *isegen* is the only tool used on the cluster.

At this point, the 4652 points each in the weighting spaces for both the original combinational heuristic and the new energy optimisation heuristic have been evaluated, and acceleration derived for each using the model of section 2.3.1. The top ten results by acceleration for each combination of benchmark, heuristic, and I/O constraint (24 combinations) are taken and synthesised to obtain power results. The DesignCompiler, PowerCompiler, and ModelSim tools are used to synthesise the CFA Verilog models and test-benches from the *uarchgen* tool as per the methodology in section 4.2.2. Using the model presented in section 4.2.2, an estimated figure for energy is produced for each of the 240 synthesised sets of CFAs with respect to a single run of their original applications. FPU power is included as before in results for FIR, LMSFIR, and FFT1K as these utilise floating point. Applications vary quite greatly in their absolute performance, so all results are normalised against the baseline (non-extended) performance. Once performance is known for each of the 240 points, the sets of 10 originally produced are combined to form arithmetic averages from which to compare the efficacy of the original combinational and new energy-aware heuristic. Performance figures covered are application acceleration, application ISE coverage, acceleration over ISE coverage, power, and energy.

Determining the stability of the new heuristic requires evaluation of the value of N as discussed in earlier experiments (sections 5.2 and 5.4): The percentage lost from the maximum acceleration observed, which must be accepted in order to acquire a common weighting vector for all benchmarks tested. A lower value of N indicates a more stable heuristic.

5.5.3 Energy Heuristic Results and Evaluation

A concern when creating this heuristic was that the added runtime complexity required in order to estimate the power for a particular ISE could be prohibitive. In reality the runtime of the *isegen* tool is not usually increased. For the most complex benchmark tested so far (e.g. that with the largest number of nodes in a single basic-block DFG), UTDSP FIR, the execution time of the *isegen* tool goes from 8:35 to 6:25 (m:s) when using the energy-aware heuristic instead of the original one. The reason for the decrease in execution time is not immediately obvious,

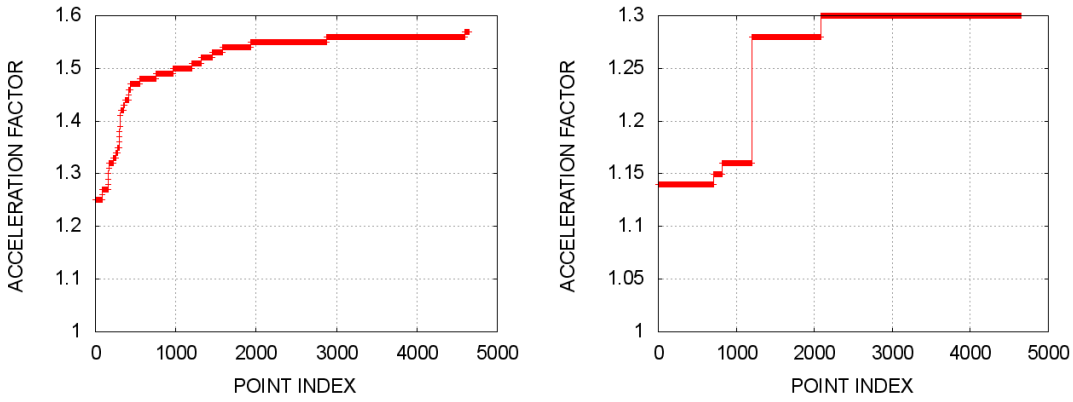
since the heuristic is being made more complex. The ISEGEN algorithm is an approximate search algorithm [21]. The ISEGEN search may therefore be shortened by the new heuristic reducing the number of steps taken to settle on a solution. If the algorithm converges on a non-improving solution faster due to the increased complexity of the heuristic, then the execution time can be reduced in the way seen here.

The results of the weighting vector-space sweep for the energy-aware heuristic are presented in figures 5.15 and 5.16. The shape of these graphs should be compared to figures 5.9 and 5.10 from the previous experiment, which represent the weighting vector spaces for the original heuristic that the energy heuristic is being compared to.

Despite the new heuristic being intended to optimise for energy, in several cases a better acceleration has been produced also. For all benchmarks except UTDSP LMSFIR, the top design points (those produced at the top-performing end of the weighting vector space) have greater acceleration under the new energy-aware heuristic, than with the original heuristic. The energy-aware heuristic encompasses a different aspect of acceleration merit than the original; the actual ratio of software to hardware ISE latency forms the merit, rather than the difference in cycles. This new heuristic therefore favours smaller ISEs than the combinational heuristic. The expansion to the largest possible ISE will in most cases cause the ratio of software to hardware to decrease, in addition to a reduction in the number of output ports used per-ISE.

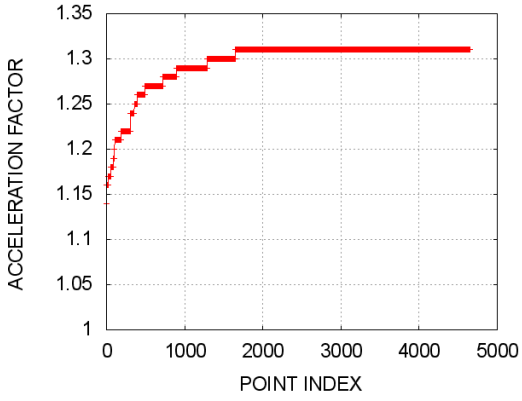
The weighting vector spaces for the original and new are similar in shape, but with some important differences. A large portion of the vector space results is close to (within 10% of) the maximum acceleration obtained in that benchmark. This was not the case with the original heuristic. This is not a trivial result, as it means that the algorithm is much less sensitive to the weighting vector with regards to producing a better result. It is as if the entire vector space had been flattened towards higher acceleration, and this is as a result of using smaller ISEs. One major derivation that can be made from this is that smaller ISEs are more effective *in general*, and this is because smaller ISEs contain less complex structure and are hence less sensitive to partitioning. This flies in the face of assumptions made in work such as [18], that larger ISEs are always better, and supports the commercial Tensilica approach [25] along with other small-ISE approaches. These results imply that a more effective and stable heuristic with respect to evaluating ISE merit, is the ratio of software to hardware latency of an ISE. This is rather than the difference between them, as is the case under the original heuristic based on the model in section 2.3.1. The CFA design though, does not allow for serial operations to contribute to the merit of the result through clock period surplus aggregation. Both hardware and software node latencies are integer, so as to facilitate the construction of CFA hardware as per algorithm 2. All operations in CFAs both start and finish on a clock boundary, where registers are situated to provide pipelining.

Whilst the ISEGEN toggle heuristic is parameterised by a weighting vector, the replacement of the *last_best_C* solution maintained by the algorithm at each iteration is determined by

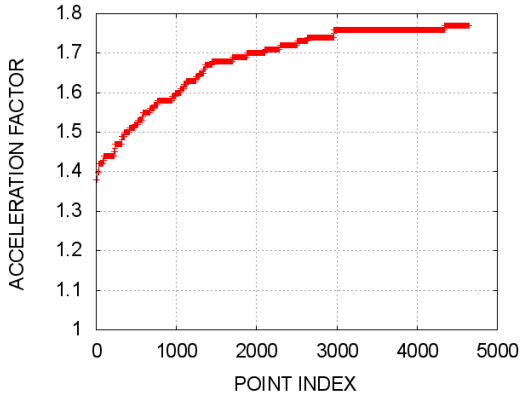


(a) FAAD AAC

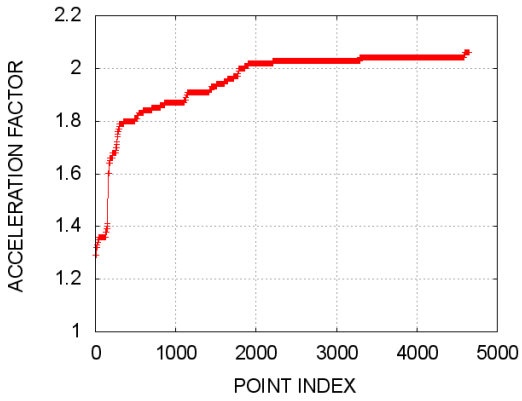
(b) SNURT CRC



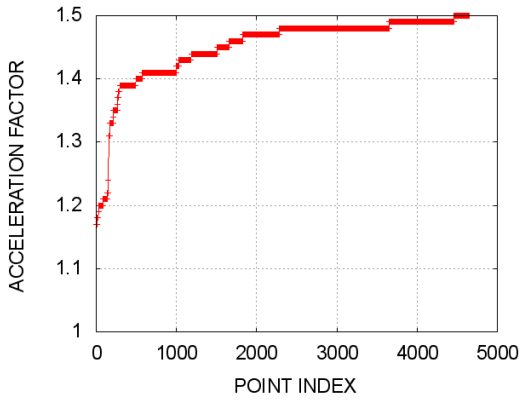
(c) SNURT FFT1K



(d) SNURT JFDCTINT

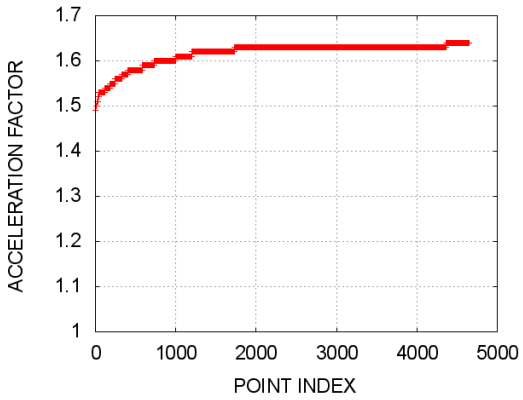


(e) UTDSP FIR

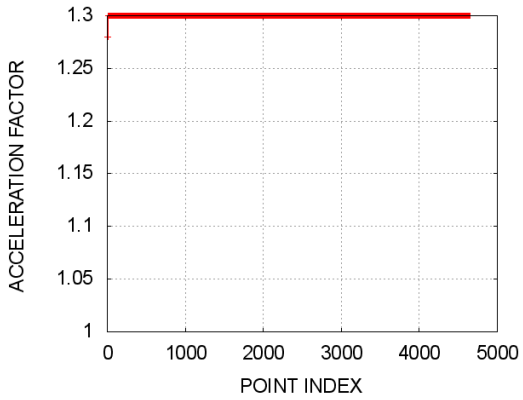


(f) UTDSP LMSFIR

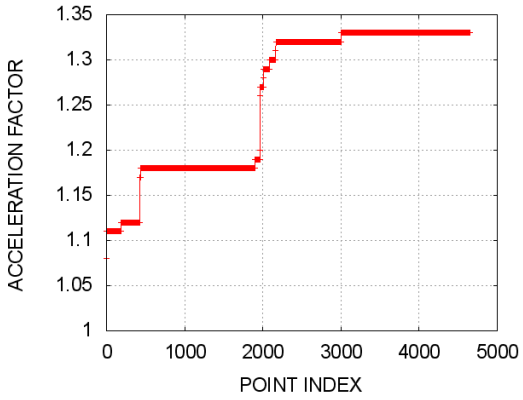
Fig. 5.15: Parameter Sweep Space Under Energy-Optimisation Heuristic (I/O: 4/4), Monotonically Ordered by Speedup Factor. These graphs should be compared to those of figure 5.1. The major difference between this figure and figure 5.1, is the vertical space which each series occupies. The energy-aware heuristic is *less sensitive* to the correct setting of the static weighting vector. In figure 5.1 up to 100% of the acceleration was lost if a bad weighting vector was chosen; in this, at most 64% is lost. Again, the worst case for sensitivity is LMSFIR.



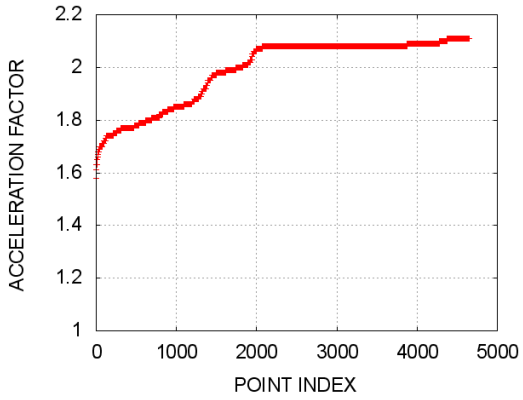
(a) FAAD AAC



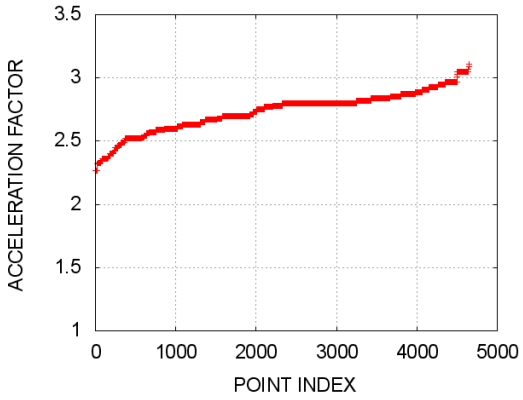
(b) SNURT CRC



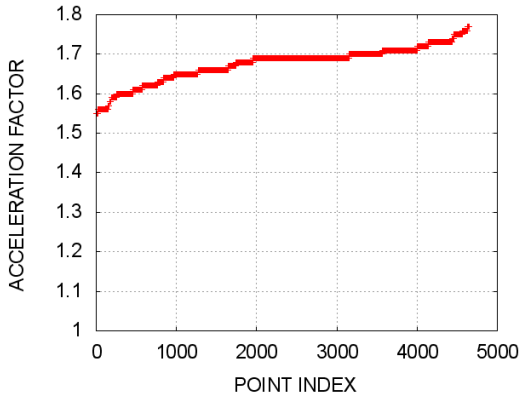
(c) SNURT FFT1K



(d) SNURT JFDCTINT

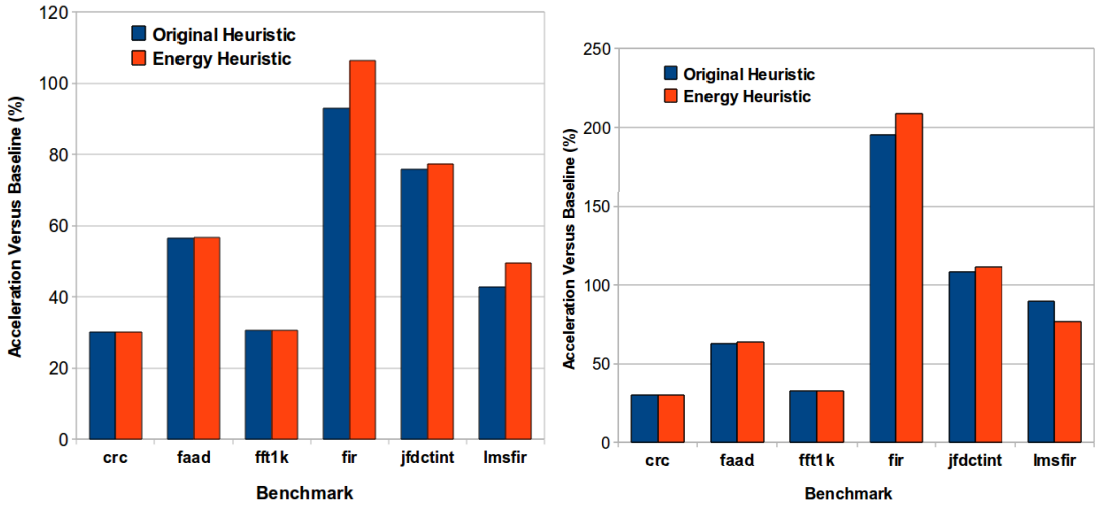


(e) UTDSP FIR



(f) UTDSP LMSFIR

Fig. 5.16: Parameter Sweep Space Under Energy-Optimisation Heuristic (I/O: 8/8), Monotonically Ordered by Speedup Factor. When compared to the original heuristic used in figure 5.2 we again see the flattening of the range, indicating that the energy heuristic is more stable than the original. Comparing these graphs to those of 5.15 we can see that the increased I/O has also increased the stability of all but one benchmark. FFT1K gets less stable here than under 4/4 I/O. Due to a single larger ISE identified early in the process, the potential for variation is reduced due to a larger portion of the vector space leading to the same result. The higher I/O of 8/8 enabled this large ISE to occur. Once again the idea that "bigger is better" in ISE is contradicted. Particularly striking here is the CRC benchmark, which through the energy-aware heuristic is almost completely stable throughout the entire vector space.



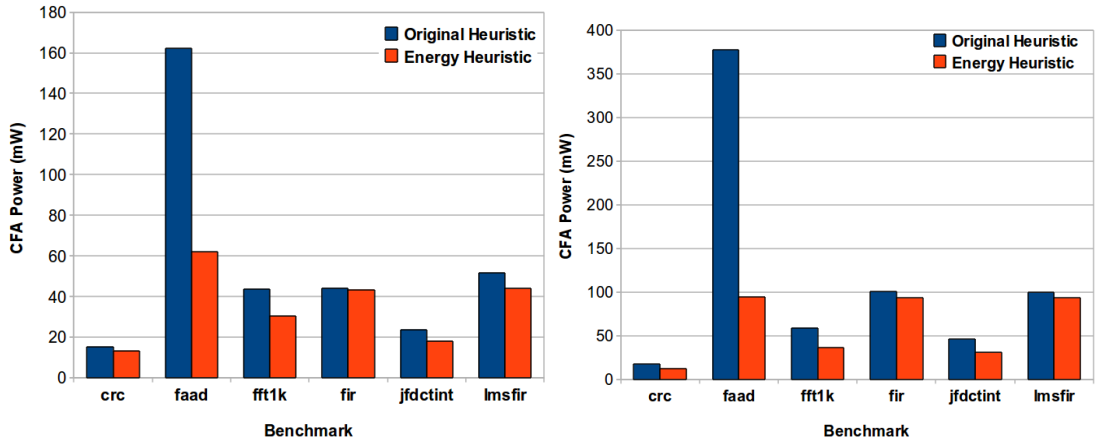
(a) I/O: 4/4 (b) I/O: 8/8

Fig. 5.17: Comparison of Acceleration Normalised to Baseline for Original and Energy Heuristics. All but one benchmark obtain acceleration greater than under the combinational heuristic. This is in addition to the reductions in power and energy shown in figures 5.18 and 5.17, so we are obtaining a slightly better acceleration for a much lower cost.

the $M(C)$ function alone. In this section, the $M(C)$ function was originally that detailed in section 2.4.2 but has now been replaced with the energy-aware heuristic. In particular, the effect of changing $M(C)$ can be seen in the graph for SNURT CRC at 8/8. Almost the entire graph sits at the top acceleration observed. We would expect at least one fifth of the graph to display rather more varied behaviour, because the new heuristic would be zeroed out in the weighting vector. Under the original heuristic, the graph displayed worse acceleration where the *speedup* merit element was nullified. Whilst one benchmark behaving in this manner does not provide any concrete conclusions, it does seem that the $M(C)$ whole-cut merit function performs better using the energy-aware heuristic.

There is no cross-benchmark trend immediately visible in the values of the weighting vectors for the top ten results for each benchmark. The best ten weighting vectors within each benchmark's results are, however, similar. After performing the same analysis as in previous sections, a common vector $\{1,2,16,0,1\}$ was located at $N = 5\%$, the lowest N for a common vector across all benchmarks tested for any heuristic so far. This is to be expected given the observations already made on the shape of the weighting vector space graphs; much of the weighting vector space results in ISEs close to the maximum performance observed.

The energy-aware heuristic effectively works on three axes of concern at once, which are connected to one-another as design concerns in the architecture as was detailed in the earlier section 4.2. Area lies at the top of this chain, in that it is the metric which we can actually control directly by inclusion or exclusion of nodes in an ISE. When ISEs are combined to form a CFA, the average size of the ISEs will generally determine the area consumed by a CFA, and hence the power it consumes. In all cases examined here, the area of the CFA is reduced along

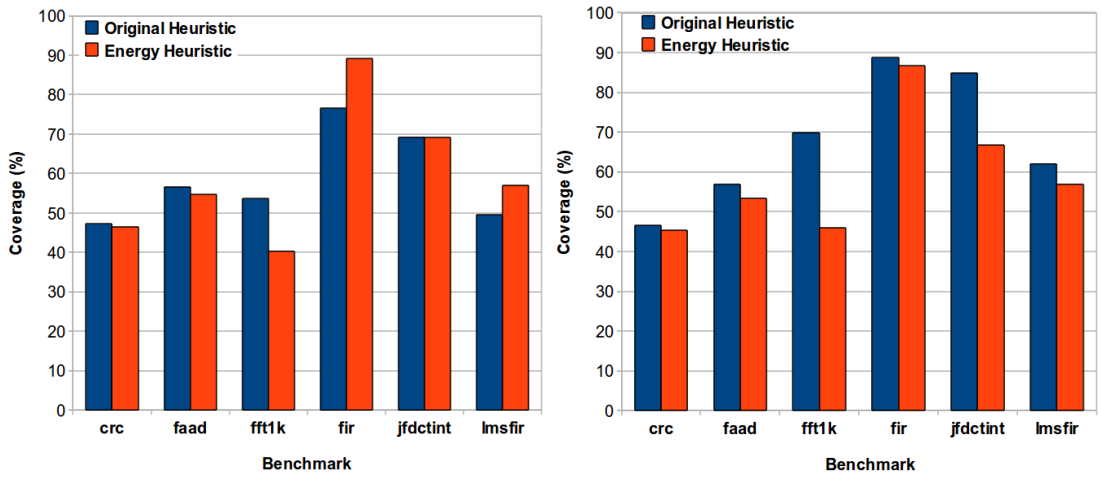


(a) I/O: 4/4 (b) I/O: 8/8

Fig. 5.18: Comparison of CFA Power for Original and Energy Heuristics. The power of the CFA is largely proportional to its area, and the energy heuristic utilises this in reducing the overall energy of the solution. FAAD obtains the greatest reduction because of the diversity of its various kernels. The same effect is observed in the graphs of section 4.3; FAAD obtains a massive reduction in area compared to the single kernels. By identifying smaller ISEs, the energy-aware heuristic reduces their complexity and hence improves resource sharing. The reduction in complexity is tied to the reduction in size, which also reduces total area. The reductions in power are the largest contributor to the energy savings the energy-aware heuristic obtains, but not the only contributor.

with the number of CFAs (multiple units are used when multiplexors gets too wide). In terms of cost, the area used could be as important as the power and execution speed. The holistic improvement in costs afforded by the energy-aware heuristic is definitely a useful property, and an improvement on the original heuristic.

Largely proportional to the area reduction of CFAs, is the power reduction of CFAs; a concept which was used to develop the heuristic used herein. Figure 5.18 demonstrates the power reduction of CFAs across all benchmarks. The greatest benefit is seen for the FAAD application, which for I/O: 8/8 achieves a 4x reduction in power through the use of the energy-aware heuristic. Through identifying smaller and faster (higher acceleration *factor*) ISEs, the area and hence the power is reduced considerably as these ISEs better share resources owing to their lower complexity. Higher I/O makes the energy-aware heuristic relatively better compared to the original. At 4/4 the power is reduced on average by a factor of 1.45x. At 8/8 the power is reduced on average by a factor of 1.77x. This is because at 4/4 the ISE candidates are already around half the size of those at 8/8. The energy-aware heuristic favours smaller, faster ISEs in order to achieve lower power. The smaller 4/4 ISE candidates are closer to the smaller ISEs generated by the energy-aware heuristic. Being closer to begin with, there is less room for improvement. This is not to say that the energy-aware heuristic is unable to utilise higher I/O. This is demonstrated by the improvement in acceleration when moving from 4/4 to 8/8 using the energy-aware heuristic. The energy-aware heuristic is just able to apply larger I/O more



(a) I/O: 4/4 (b) I/O: 8/8

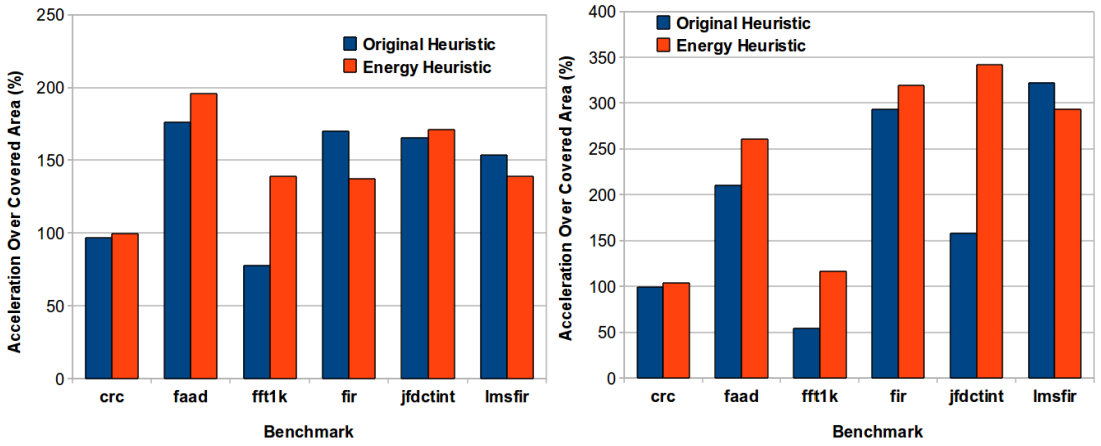
Fig. 5.19: Comparison of ISE coverage (percent of full benchmark) for Original and Energy Heuristics. Between 4/4 and 8/8 a trend emerges; higher I/O leads to lower overall coverage of the code by ISEs, when using the energy-aware heuristic, versus the original. By covering a smaller quantity of code with ISEs having a higher acceleration factor (see figure 5.20) the resulting application acceleration is maintained or improved, at a lower cost. The new heuristic is apparently more successful in this trade-off at the higher I/O of 8/8.

judiciously.

Power is only one factor in reducing the energy of the extended hardware/software co-design, falling squarely under hardware. The other side of the coin is concerned with the actual use of the ISEs defined with respect to the software they extend. The acceleration of a benchmark can be considered as two different factors: the coverage of the application by ISEs, and the acceleration of the covered area. This presentation has a lot in common with the Amdahl Limit defined in section 2.3.4, which considers the maximum amount of the application which *can* be covered by ISEs and an infinite speedup of those coverable areas.

In figure 5.19, the resulting ISE coverage of the application is shown with the old and new heuristics; in figure 5.20 the acceleration obtained over the covered portion is shown. Combining the two numbers appropriately we reach the results presented in figure 5.17. There is no absolutely common trend between all the benchmarks for coverage, but for the most part the energy-aware heuristic results in a smaller portion of coverage. The exceptions here are UTDSP FIR and LMSFIR at I/O: 4/4, which both have a larger coverage under the energy-aware heuristic. The extremely data-parallel nature of these benchmarks which share FIR filter code is the cause of the *increase* in coverage using the energy-aware heuristic. Where an ISE has a single node which is not operation-parallel, as is often the case in complex instructions, the original heuristic will likely include it whereas the energy-aware heuristic will not.

By way of example, consider the expression $x = (a * b) + (c * d)$, which is a 4-input, 1-output expression. The + is not operation-parallel, and would likely be included under the

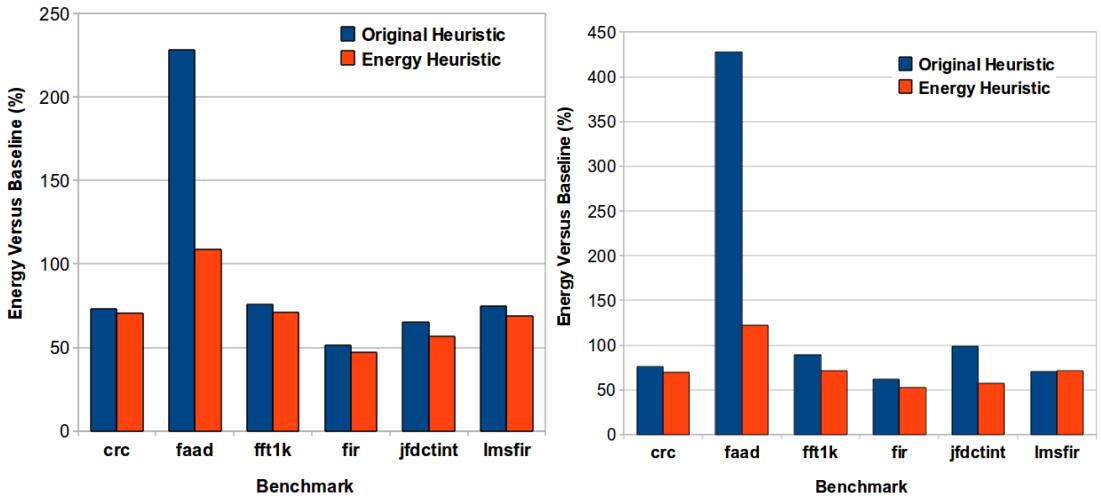


(a) I/O: 4/4 (b) I/O: 8/8

Fig. 5.20: Comparison of Acceleration Over Covered Area for Original and Energy Heuristics. The energy-aware heuristic tends to favour ISE which are individually faster than the ISE that would be identified by the original heuristic. As seen here once again, the energy-aware heuristic is rather more effective versus the original at the higher I/O constraint. FFT1K and JFDCTINT at 8/8 are the most affected in this regard, obtaining ISEs that are 2x higher acceleration over the area covered, than those identified by the original heuristic. Referencing figure 5.19 we see that this is in tandem with a nearly 2x reduction in the area covered. The two effects cancel out in terms of acceleration, but the resulting ISEs are much smaller and hence cheaper under the energy-aware heuristic.

original heuristic but excluded with the energy heuristic (resulting in a 4-input 2-output ISE). This effect leads to the increase in coverage because the new heuristic favours expansion of new ISEs in a breadth first fashion, which allows for a greater coverage of the LMSFIR and FIR filter code at 4/4. Where FIR and LMSFIR have greater coverage, they have lower acceleration. The extremely flat nature of the FIR operation (which has very little depth in its DFG) is what led to this effect. The original heuristic has no depth to explore, and hence performs a similar exploration to the energy heuristic, which unfortunately falls short. It is likely that data-flow flattening source-transformations such as software-pipelining performed prior to ISEGEN analysis under the original heuristic would therefore also result in lower-area, lower-power, lower-energy extensions with a similar or better acceleration. It is unclear, however, how these would compare with the more complex approach of this energy-aware heuristic. With lower coverage resulting from extensions as are favoured by the energy-aware heuristic, higher acceleration over the covered area is likely. This in itself is proof that the heuristic is working as intended, as this is in line with the original observations: smaller extensions with a higher acceleration are better for energy consumption. The same is largely true for application acceleration. Unlike other works such as [18] this result directly disproves the notion that the larger an individual ISE is, the better it is for acceleration.

The intended bottom line improvement in this work, is the actual energy consumption of the combined hardware-software co-design, when using CFAs for ISE implementation. We



(a) I/O: 4/4 (b) I/O: 8/8

Fig. 5.21: Comparison of Energy Normalised to Baseline for Original and Energy Heuristics. The combined effect of ISEs identified under the energy-aware heuristic is a reduction in energy. In more complex cases such as FAAD, the effect is considerable, and a 3.6x reduction in energy is made. This is in combination with a slight increase in acceleration in most cases. The energy-aware heuristic fails to improve LMSFIR energy due to having reduced both coverage (figure 5.19) and acceleration over the covered area (figure 5.20). The resulting decrease in application acceleration is greater than the decrease in power (figure 5.18).

have already demonstrated that in all but one case (UTDSP LMSFIR) the energy-aware heuristic outperforms or equals the acceleration obtained by the original heuristic. In the case of LMSFIR the drop in acceleration leads directly to the inefficacy of the energy-aware heuristic in reducing energy. Area and power are still reduced for LMSFIR, but not by enough to counter the ill effect of the lower acceleration. LMSFIR is 0.90% greater energy using the energy-aware heuristic than the original. This is the only case where the original heuristic outperforms the energy-aware heuristic on energy.

The energy-aware heuristic has been shown to reduce the area and power in all instances tested. Figure 5.21 demonstrates the energy consumption of the designs resulting from the two heuristics and in all cases but LMSFIR the result is an energy saving. Under both heuristics, all benchmarks apart from FAAD are reduced to energy consumption below the baseline. The average energy consumption versus the baseline over all benchmarks is 115% for the original heuristic, and 72.31% for the energy-aware heuristic. The energy-aware heuristic has taken the domain represented by these applications together, and turned the CFA-based ISE effect from a net energy loss to a net energy gain. This is particularly striking as it is also done in tandem with a small increase in acceleration.

This experiment has demonstrated therefore that it is possible to make use of a more sophisticated heuristic to make improvements without having to trade off different axes. In many cases here, it was possible to improve all axes at once: Area, Power, Acceleration, and En-

ergy. This is important in itself, as it demonstrates that the improvement in energy is not just through an improvement in acceleration. The improvement is through a holistic refinement of the structure in the resulting ISEs, across all axes of concern. The quality of the ISEs with regards to their implementation as CFAs has been improved by imbuing the ISEGEN algorithm with a notion of the *costs* of the resulting architecture. Through employing smaller and higher acceleration-factor ISEs the new heuristic reduces energy by an average of 1.6x versus the original heuristic, whilst maintaining the same or better overall acceleration in most cases. Cases where the new heuristic fails to raise the bar on performance, may be better addressed with further component modifications to the $M\_toggle(n,C)$ compound heuristic.

The performance of the new lower-energy ISE candidates would contribute positively to the area-energy-acceleration trade-off of the kind explored in section 4.2. The function of area to energy demonstrated in the earlier section 4.2 would be skewed downwards, as the individual ISE candidates are themselves lower energy. This is verified later in section 6.3. An exploration of the effects of CFA staggering as outlined in section 4.3 would also be interesting in determining if the benefits of using the energy-aware heuristic remain post-staggering. This is also examined in section 6.3.

5.5.4 Conclusions

This section has looked at the idea of producing more energy-efficient ISE through modification of the merit calculation heuristic to include the effects of energy as observed in prior work. In particular, the following conclusions are made:

- The energy heuristic introduced here works particularly well in its intended purpose; that of reducing the energy of a CFA-based ISE design through consideration of the energy effects in the identification algorithm.
- Versus the original heuristic, the average improvement in energy consumption is 1.6x. A maximum of 3.6x improvement was observed for FAAD at 8/8.
- The new heuristic was effective at reducing energy in all of the cases observed, except for LMSFIR.
- Where the new heuristic produces greater energy consumption than the original, it is only by 0.90%. Further work on the compound ISEGEN heuristic may be successful in overcoming such cases.
- The energy-saving effect works through several different engineering factors: area (and hence power), proportion of application covered, and the acceleration over the ISE-covered section. It would probably be more correct to call this a holistic heuristic, rather than an energy heuristic.
- The major effect seems to come from reducing area and hence power, ultimately making the new heuristic favour smaller ISEs with more acceleration per area/power.
- The energy heuristic is the most stable under a static weighting vector seen so far, with $N = 5\%$.
- The demonstration of energy concerns in this section shows that the "bigger is better" mentality taken by some approaches is not completely valid if we do not consider the cost of the acceleration produced.
- Despite applying this energy-aware approach at the localised level of a single DFG/ISE/CFA, it still works in aggregate when multiple CFAs are merged. The idea that multiple smaller CFAs merged together create a smaller set of merged CFAs has now been confirmed.
- An energy-based selection heuristic would allow this to be properly extended to include an area-limit, as the efficacy of the existing selection heuristic based on acceleration alone is likely to be worse in comparison.

- The energy-aware heuristic ought to be useful for non-CFA ISE, assuming a model is available for area and the same correlation between area and power is present in the representative microarchitecture.

The energy heuristic is included in the later section 6.3 to provide a counterpoint to the combinational heuristic when performing work to determine the energy efficiency of floating versus fixed point number formats.

5.6 Summary

This chapter has proposed and evaluated four novel ways for improving the ISEGEN identification algorithm:

- A methodology for obtaining a more effective static weighting vector.
- Early termination, considerably reducing ISEGEN run-time in more complex cases.
- An energy-aware heuristic, able to focus the algorithm on energy reduction.
- A pipeline-aware heuristic, able to turn identification towards ISEs which can be scheduled in an overlapping fashion to provide greater acceleration.

All of these efforts have been successful in increasing the efficacy of the ISEGEN algorithm with regards to the engineering concerns outlined in chapter 1.

The baseline ISEGEN algorithm itself has been tuned to a higher performance than it originally had through location of a new weighting vector, different to the one previously claimed by the original authors of the algorithm [7].

The time taken by the ISEGEN algorithm has been considerably reduced, dramatically in many cases, and without compromising the quality of the result. Whilst there is the potential for differences between the original and early-terminating versions of the algorithm's output, no example of this was observed in the benchmarks examined. Beyond seven times runtime reduction were observed using the early termination heuristics. Both the trend observed and analysis of the original algorithm's runtime show that the saving will become both absolutely and relatively more significant as the size of DFG analysed grow.

The pipelined structure of the CFA (and any other microarchitecture capable of pipelined ISE execution) is exploited through a new pipelining heuristic embedded in the ISEGEN identification algorithm. In addition to demonstrating the benefits of pipeline scheduling on independent ISE and the potential for this to be used as an alternative to other OLP-consuming techniques, the pipelining heuristic was more successful and stable than the original combinational heuristic in producing such designs.

The energy consumption of CFAs as modelled in earlier experiments was introduced as yet another modification to the ISEGEN algorithm's heuristic. The identification process was hence steered towards smaller, higher acceleration-factor, more cost-effective ISEs. Considerable energy reductions are effected through the identification algorithm favouring smaller ISEs with a greater acceleration ratio, rather than a greater number of cycles removed. In most cases the new algorithm managed to reach even greater acceleration than the original "bigger is better" heuristic managed to reach. The consideration of energy and implication for the structure of identified ISEs is offered as evidence that the "bigger is better" school of thought in ISE is naive with regards to any other engineering concern than pure acceleration.

Considerable advances in efficacy have now been reaped in these areas attacked; this thesis now moves on to examining the effect of source code form on the AISE process.

6 FORM OVER FUNCTION: SOURCE TRANSFORMATION

"Form ever follows function"

– Louis Sullivan

This chapter looks at the effects of source code transformations and number formats on AISE performance. Different representations of the same application are processed by AISE to see if any particular trends emerge that can be exploited. *A methodology for combined exploration of source transformation and ISE is presented, and demonstrated to improve the acceleration of the result by an average of 35% versus ISE alone. Floating point is demonstrated to perform worse than fixed point, for all design concerns considered here, regardless of ISEs employed.*

6.1 Introduction

The AISE process operates with the source code of the application in question as a major input. Those familiar with compiler technology are aware of the fact that any given function (for example an algorithm) can be given multiple forms. This is basically saying that there are many equivalent ways of doing the same thing; this is of interest because the form in which a function is represented has considerable sway over a compiler's ability to produce an efficient implementation. This is the basis of source-level transformation.

Tools exist to allow for large swathes of transformation spaces to be defined and explored with a source-to-source preprocessing of the application in question. Whilst originally intended as a preprocessor to enable such transformations in a compiler, we can also use these tools to set up and explore transformation spaces with regards to AISE. The next section in this chapter uses this approach with an older version of the tool-chain than used in prior experiments, in an effort to make observations we can use to further increase the efficacy of ISE. In particular, the process of software transformation is cheap in comparison to the process of ISE. The former affects only software and the latter requires hardware support. This means that it is important to determine whether a relationship exists between the work performed in software transformation and the cost-benefit of ISEs resulting from AISE.

The choice of number format is a decision usually falling on the side of fixed point when embedded processors are considered; the alternative of course is floating point (and generally the IEEE-754 single and double-precision form). It has been said that "floating point is for people who don't know their data", because the use of floating point is considerably more expensive and often less accurate than properly designed fixed point. Being properly designed is of course very important, and therein lies the real trade-off: floating point is supposed to be cheaper in terms of engineer time, and fixed-point is supposed to be cheaper in terms of hardware required. The former point is trivially true assuming you don't mind a little inaccuracy in

your data. The latter point could be disrupted by ISE technology. To determine the trade-offs inherent in the choice between the two number formats when also considering ISE, the final section of this chapter performs DSE using the tool-chain from earlier chapters. Staggering and the energy heuristic are explored in addition to the original heuristics to give a clearer understanding of their behaviour in combination.

6.2 Transform Space Exploration

In this section we look at the combined space of source code transformation and ISE, looking for trends which will enable further work involving this combination. *A methodology for combined exploration of source transformation and ISE is presented, and demonstrated to improve the acceleration of the result by an average of 35% versus ISE alone.*

6.2.1 The Need for Source-to-Source Transformations in ISE

Compiler transformations at any level have the potential to be used to manipulate the syntax and semantics of a program in a way that will not affect its function. A trivial example is constant sub-expression elimination which removes redundant operations by taking constant expressions and reducing them to a constant value. A program that has been transformed should not differ in black-box behaviour, with exception of the time it takes to execute and the resources (memory and processing) it consumes. The exception is the benefit we are actually looking for when we employ transformations; whilst not all transformations are beneficial to optimisation, we use them because we hope that they are. Where hope in this context becomes reality is largely governed by heuristics in the compiler, which make guesses on which transformation to apply in order to obtain a benefit. In the absence of heuristics from prior knowledge of the area we are trying to optimise for, large-scale sampling of the transformation space is required to get an idea of the lay of the land. The combination here of ISE with prior source transformation warrants such an approach. So herein we perform a large-scale sampling of the transformation space and process it with an ISE tool. In this manner we hope to get an idea of how source transformations can be used to optimise the source to make it more amenable to ISE.

A very simple worked example now follows to motivate the combination of the two techniques:

As an example, consider the code excerpt in figure 6.1(a). The two functions *icrc1* and *icrc* are part of the SNURT *CRC* benchmark and implement a cyclic redundancy check for an input string stored in the array *lin[]*. The key features of the code are small *for* loops in both functions, which contain conditional branches and perform a larger number of bit-level manipulation operations. AISE generates new candidate instructions, which result in a 25% performance improvement over the baseline code.

In figure 6.1(b), the main differences due to source-level transformations of the code in figure 6.1(a) are shown. While the code is functionally equivalent, it outperforms the code in figure 6.1(a) by a factor of 1.63x. Loop unrolling has been applied to the *for* loop in the *icrc1* function. This reduces the loop overhead and improves flexibility for instruction scheduling. In the *icrc* function, the effects of source-level transformation are more fundamental. *for* loops in the code have been lowered into *do-while* loops and, most important, *bit-packing* and *hoisting*

```
unsigned short icrc1(unsigned short crc,
                     unsigned char onech)
{
  int i;
  unsigned short ans = (crc^onech << 8);

  for (i=0;i<8;i++) {
    if (ans & 0x8000)
      ans = (ans <<= 1) ^ 4129;
    else
      ans <<= 1;
  }
  return ans;
}



unsigned short icrc(unsigned short crc,
                    unsigned long len,
                    short jinit, int jrev)
{
  /* Some variable declarations */
  /* ... */


  if (!init) {
    init=1;
    for (j=0;j<=255;j++)
    {
      icrctb[j]=icrc1(j << 8,(uchar)0);
      rchr[j]=(uchar)(it[j & 0xF] << 4 |
                      it[j >> 4]);
    }
  }

  if (jinit >= 0)
    cword=((uchar) jinit) |
      (((uchar) jinit) << 8);
  else
    if (jrev < 0)
      cword=rchr[HIBYTE(cword)] |
        rchr[LOBYTE(cword)] << 8;

  for (j=1;j<=len;j++)
  {
    if (jrev < 0)
      tmp1 = rchr[lin[j]]^ HIBYTE(cword);
    else
      tmp1 = lin[j]^ HIBYTE(cword);
    cword = icrctb[tmp1] ^ LOBYTE(cword) << 8;
  }

  if (jrev >= 0)
    tmp2 = cword;
  else
    tmp2 = rchr[HIBYTE(cword)] |
      rchr[LOBYTE(cword)] << 8;

  return (tmp2);
}
```

(a) Original SNURT CRC implementation

```
unsigned short icrc1(unsigned short crc, /*...*/)
{
  /*...*/

  /* Loop unrolling */
  for (i=0;i<8;i+=2) {
    if (ans & 0x8000) ans = (ans <<= 1) ^ 4129;
    else ans <<= 1;
    /*...*/
  }
  return ans;
}

unsigned short icrc(unsigned short crc, /*...*/)
{
  /*...*/

  /* Bit packing */
  suif_tmp = (cword /*...*/) |
    (jinit /*...*/) >> /*...*/ |
    (jrev & /*...*/) >> /*...*/ |
    (len & /*...*/) >> /*...*/;

  cword = 1u & suif_tmp;

  if (!init) {
    init = 1;
    /* Loop lowering */
    j = 0;
    do {
      /* Computation of icrctb & rchr ...*/
    } while (j <= 255);
  }
  /* Bit unpacking */
  if (0 <= ((2u & suif_tmp) >> 1u))
    cword = ((2u & suif_tmp) >> 1u) |
      (((2u & suif_tmp) >> 1u) << 8u);
  else
    if (((4u & suif_tmp) >> 2u) < 0)
      cword = /*...*/

  j = 1;
  if (1ul <= ((8u & suif_tmp) >> 3u)) {
    /* Move loop invariant conditionals */
    if (((4u & suif_tmp) >> 2u) < 0) {
      /* Loop lowering */
      do {
        /* Computation of cword ... */
      } while (j <= ((8u & suif_tmp) >> 3u));
    }
    else {
      /* Similar lowered loop as before ... */
    }
  }

  if (0 <= (4u & suif_tmp) >> 2u) tmp2 = cword;
  else tmp2 = /*...*/;

  return tmp2;
}
```

(b) CRC after transformation

Fig. 6.1: Original *SNURT CRC* implementation (a) and after application of source-level transformations resulting in best combined performance (b).

```
EXPR: mirConvert
  EXPR: mirConvert
   EXPR: mirConvert
    EXPR: mirOr
     EXPR: mirConvert
      EXPR: mirShiftLeft
       EXPR: mirConvert
        INPUT REGISTER - BB #1 ::  EXPR: mirContent
       INPUT REGISTER - BB #2 ::  EXPR: mirIntConst Val:4
     EXPR: mirConvert
      INPUT REGISTER - BB #3 ::  EXPR: mirContent
```

Fig. 6.2: Complex instruction template generated for the transformed CRC code in figure 6.1(b)

of loop invariant conditionals transformations have been applied. *Bit-packing* packs multiple variables into a single variable of type integer and, on its own, usually degrades performance. When combined with ISE generation, however, otherwise expensive bit-level manipulation operations for packing and unpacking can be encoded as complex, but fast instructions and yield an overall performance improvement. In fact, the instruction templates generated for example 6.1(b) are generally more complex than those generated from the baseline code. An example of such an instruction is shown in figure 6.2 and it implements the mentioned packing/unpacking operation. Moving loop invariant conditional outside the loop eliminates redundant comparisons and jumps and further increases performance.

ISE generation based on the transformed code in figure 6.1(b) result in a further 23% improvement (over just transformed code), or a total combined speedup of 2.02x over the baseline. Only a certain part of the performance gain can be directly attributed to code transformations, the rest is due to the enabling effect of the source-level transformations on the ISE generation.

This short example demonstrates how difficult it is to predict the best source-level transformation and instruction set extension for a given application. It also shows that high-level code and low-level architecture optimization cannot be separated, but are tightly coupled. Combined exploration of both the software and hardware design spaces will generate a significantly better solution than isolated optimization approaches could consistently produce. Presented now is an empirical evaluation of this HW/SW design space interaction. It is later shown that a probabilistic search algorithm is able to examine a tiny fraction of the optimization space and still find significant performance improvements. This is in keeping with the earlier observations regarding the static weighting vector exploration, in sections 5.2, 5.4, and 5.5.

6.2.2 Transform Space Exploration Methodology

The primary concern of this experiment was to determine which transformations or combinations thereof infer the greatest acceleration to application-specific software under ISE au-

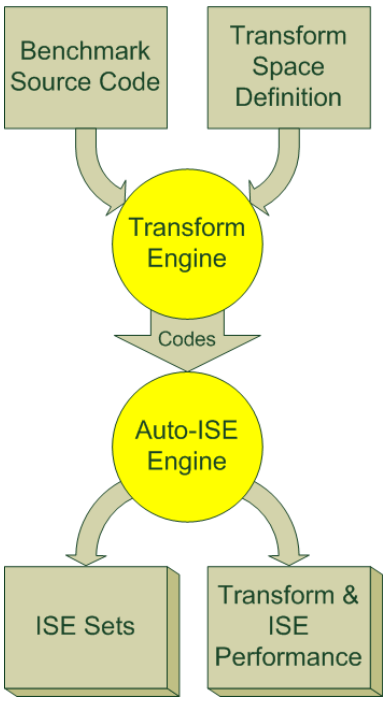


Fig. 6.3: The combined but phased searching of transform and ISE design spaces; our experiment methodology as a flow diagram.

tomation. Secondly, the experiment was to find limits for performance gain and loss from the combined design space defined by transformation and ISE over a baseline design employing neither. With this information, we are well equipped to properly focus the efforts of future research towards the most beneficial transformations for ISE.

To represent the transformation design space in this experiment, we use a source to source transformation tool built upon the SUIF1 [62] compiler framework. Samples are taken with uniform probability, at random points across the entire space of potential transformation. A sample in this sense represents a single point in the transformation space, and results in the ordered set of transformations selected at that sample point to be applied to the code. The tool generates large volumes of transformed source code samples rapidly from a definition of:

- The source code, in C; for this a variety of single-function benchmarks are tested.
- The Transform Space Definition, as the boolean inclusion or exclusion of transforms permitted in the space, plus the maximum number of transformation phases for each sample. The tool supports a wide array of source to source transformations to be used in the exploration
- The number of samples to take from the transformation space, and hence the number of transformed source codes to produce. Here this was set to 10,000, however some sequences of transformation were invalid or produced invalid code and hence were culled

from the results. The number of valid transformation sequences is covered in section 6.2.3.

The benchmarks used in this experiment were taken from SNURT[146] and UTDSP[136] suites. Those taken were as follows:

- **SNURT**; adpcm, crc, fft1, fft1k, fir, jfdctint, lms, ludcmp, matmul, minver, qsort-exam, qurt, select.
- **UTDSP**; edge_detect, fft_1024, fft_256, fir_256, fir_32_1, histogram, iir_4_64, latnrm_32_64, latnrm_8_1, lmsfir_32_64, lmsfir_8_1, mult_10_10, mult_4_4.

We store for each benchmark the entire set of transformed source codes representing that benchmark after sample points in the transformation space are applied to it. The set of transforms applied at each sample is also stored for later correlation in analysis.

The transforms from which each point is selected are as follows; further information may be obtained from the SUIF1 documentation [62]:

- Array Delinearization.
- Bit Packing.
- Break Load Constant Instruction.
- Bounds Comparison Substitution.
- Call By Reference Replacement.
- Chain Array References.
- Common Subexpression Elimination.
- Control Simplification.
- Constant Propagation.
- Constant Folding.
- Copy Propagation.
- Dead Code Elimination.
- Default SUIF Transformations.
- Dismantle Array Instruction.
- Dismantle Div Ceil/Floor Instruction.

- Dismantle Div Mod Instruction.
- Dismantle Empty Tree For.
- Dismantle Int Abs/Max/Min Instruction.
- Dismantle Abs/Min/Max Instruction.
- Dismantle Multiway Branch.
- Dismantle Non Constant For.
- Dismantle Tree Block.
- Dismantle Tree Block Without Symbol Table.
- Dismantle Tree For.
- Dismantle Tree For With Modified Index Variable.
- Dismantle Tree For With Spilled Index Variable.
- Dismantle Tree Loop.
- Eliminate Enumeration Types.
- Eliminate Struct Copies.
- Eliminate Sub Variables.
- Explicit Load Store.
- Extract Upper Array Bounds.
- Find For.
- Fix Address Taken.
- Fix Bad Nodes.
- Fix LDC Types.
- Full Copy, Forward and Const Propagation.
- Global Variable Privatisation.
- Globalise Local Static Variables.
- Guard For.
- If Hoisting.

- Imperfectly Nested Loop Conversion.
- Improve Array Bounds.
- Induction Variable Detection.
- Kill Redundant Line Marks.
- Lift Call Expression.
- Loop Invariant Hoisting.
- Loop Unrolling: 2x, 4x, 5x, and 7x.
- Loop Flattening.
- Mod Ref Annotations.
- Normalisation.
- Pointer Conversion.
- Privatisation.
- Reduction Detection.
- Replace Constant Variables.
- Scalarisation.
- Scalarise Constant Array References.
- Unstructured Control Flow Optimisation.

This set of transformed source codes forms a representative sampling of the entire search space for that benchmark, each sample is then processed by an automated profiling ISE tool based on the Atasu et al. Integer Linear Programming method of derivation [20]. The tool operates in three phases:

- Instrumentation; wherein the ISE tool augments the intermediate representation of the application with counters for profiling. The CoSy-based tool emits the i686 assembly for this profiling executable which is then assembled and run using the standard GNU tool chain.
- Execution; running the instrumented binary records per-basic-block execution frequencies, which are stored in a file for use by the extension phase.

- Extension; The IR is augmented with profiling statistics, which are then used to select the top 4 instructions using the Atasu ILP AISE algorithm [20]. The ISE tool's profiler combined with a latency table for the given target architecture produces runtime and code-size performance metrics for the original transform-space sample. These metrics and the generated instructions are stored alongside the transformed code and transform-point definition.

The ILP AISE algorithm used generates data-flow-graph templates through conversion of basic blocks to a set of constraints in an Integer Linear Program and solution of that program. A tool built into a CoSy compiler uses the lp_solve library to solve such problems, and generate a set of candidate templates for an entire program. Constraints are declared from each basic block to generate a template such that:

- The template is convex (i.e. does not have any holes), so that it may be scheduled.
- Input and output port constraints are met (i.e. the number of register input and output ports are sufficient), so that it may be implemented.

In addition to the constraints, a goal function is also expressed. The goal function for the Atasu algorithm [20] is the same as the ISEGEN algorithm. See section 2.3.1 for details: The per-template difference between software and hardware execution time is the per-execution gain in cycles to an architecture implementing that template. Following the generation of templates from basic blocks, the templates are checked for isomorphism with one another using the NAUTY graph isomorphism library, then ranked using the product of their estimated usage and per-execution gain. The top 4 of these instructions are then recorded alongside their performance estimates for inclusion in results.

For the purposes of this experiment, we configured the latencies to those of an Intel XScale PXA270 processor, a current high-performance embedded microarchitecture in 2007 based upon the ARM 7 instruction set. An input/output port constraint of 8/8 is set, to allow a wide range of potential ISEs and avoid limitations on our results due to the synthetic microarchitectural constraints set in the ISE algorithm. It has been shown in other work [23, 129] and in the earlier section 5.4, that pipelining of ISEs is possible to reduce per-cycle register file I/O to suit actual requirements.

We therefore have for each benchmark, for each of up to 10,000 transformation-space sample points:

- Source code after transformation.
- Instruction Set Extensions defined as data-flow templates.
- A record of performance in cycles (runtime) and instructions (code-size) before and after the transformations are applied to the benchmark.

- A likewise record of the improvement to each of the performance metrics for each of the instructions generated by AISE for the transformed source.
- Aggregation of the results of the top four of these instructions to calculate the overall benefit to the transformed code.

.

So that there is a control point for reference, we ensure that a baseline utilizing no transforms is sampled from the transformation space. Our tool produces as many ISE templates as it can find within the source code. However, we limit the number used in the results to four. In this manner, we allow only the inclusion of the best-performing ISEs, such as we hope to reveal through transformation. Current commercial approaches such as the Tensilica XPRES [25] tend to use large numbers of small instructions to preserve generality; this experiment assumes a very application-specific core is desired.

This entire experiment was run on a quad-core machine running Linux 2.6, over the course of several days in order to allow for the large-scale sampling. The tools are "pipelined" in their operation to speed up results generation, as illustrated in figure 6.3.

With these results recorded, we go on to make observations on the correlation of transformation and ISE performance, in how the spaces combine to form the more relevant performance measure: overall performance.

6.2.3 Evaluation

The graphs of figures 6.4, 6.5, 6.6, and 6.7 require a little explanation, as their presentation is not immediately obvious. The three bars presented for each benchmark give first the greatest observed performance (either acceleration or code size improvement depending on the graph) from transformation alone, followed by the ISE performance without doing any transformation, followed by the greatest observed performance from combining transformation and ISE.

Peak runtime improvements of 2.70x (SNURT ludcmp), 1.46x and 2.85x (both SNURT FFT) are seen, for transformations alone, ISE alone and the combination of the two, respectively. Average runtime improvements across both benchmark suites are 1.35x, 1.09x and 1.47x respectively. It can also be seen that of the 26 benchmarks considered 5 of them see a combined transformation and ISE runtime performance improvement of over 2.0x and only 6 see an improvement of less than 1.15x.

The graphs for code size are not based on the same sample points that runtime improvement figures are, but separate transformation sequences that were found to be effective at reducing code-size. Peak code-size improvements of 1.18x (SNURT minver), 1.46x and 1.95x (both SNURT CRC) are seen, for transformations alone, ISE alone and the combination of the two, respectively. Average code-size improvements across both benchmark suites are 1.05x, 1.08x and 1.15x respectively.

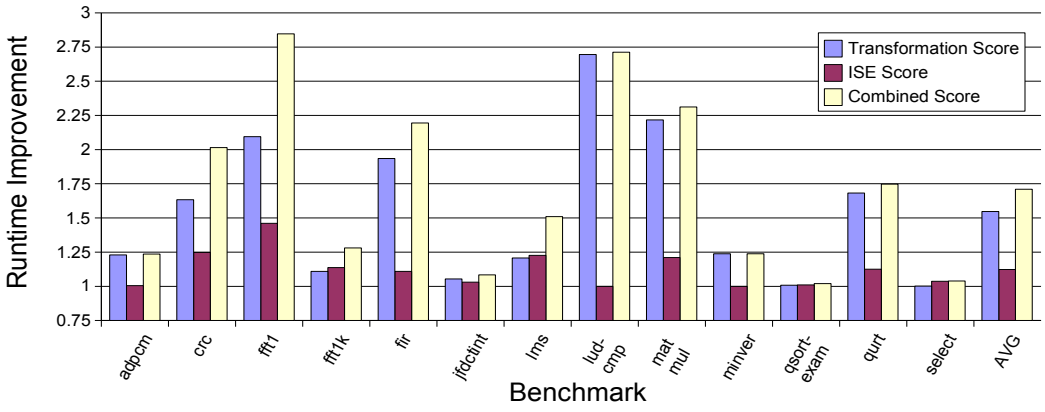


Fig. 6.4: Runtime Improvements achieved on the SNURT benchmarks. Combined score indicates the best point in the transformation space observed, for transforms and ISE combined. The combination is invariably better performing than the techniques in isolation, despite the potential for transformation to worsen results (see figures 6.8 and 6.9 for illustration).

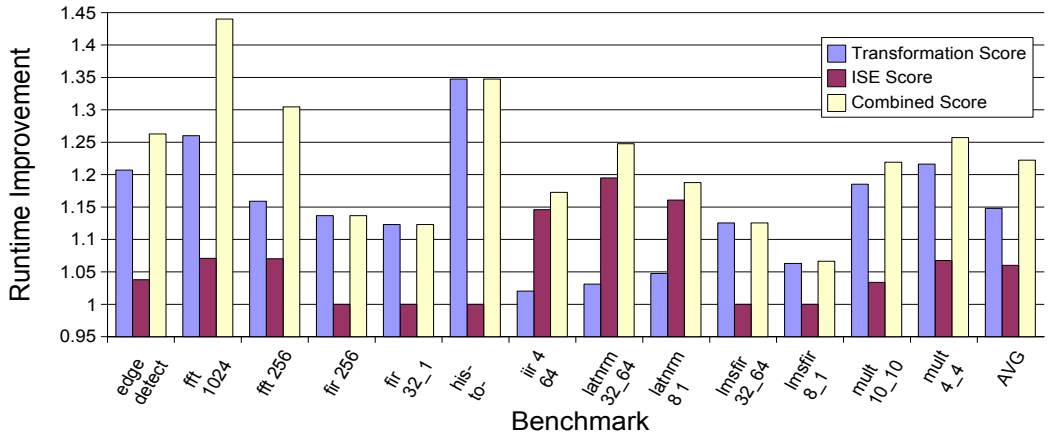


Fig. 6.5: Runtime Improvements achieved on the UTDSP benchmarks. Combined score indicates the best point in the transformation space observed, for transforms and ISE combined. The combination is invariably better performing than the techniques in isolation, despite the potential for transformation to worsen results (see figures 6.10 and 6.11 for illustration).

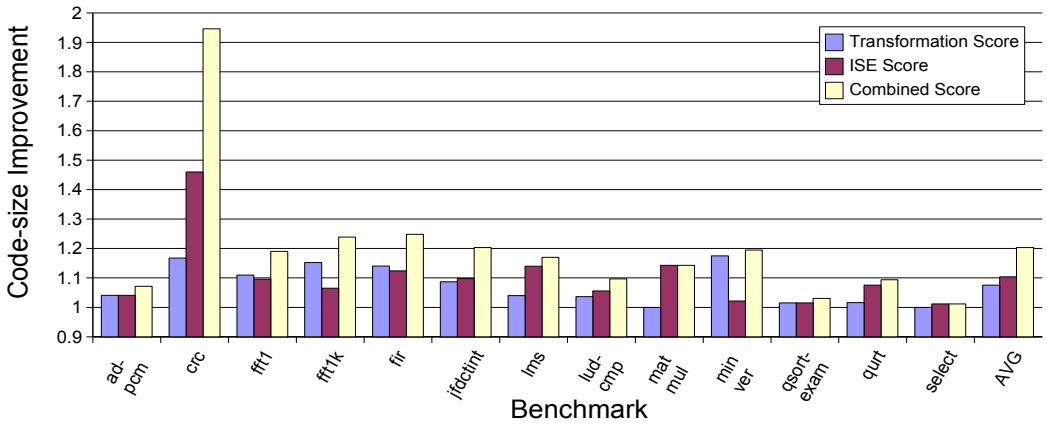


Fig. 6.6: Code-size Improvements achieved on the SNURT benchmarks. The reduction in executable size can be an important factor for cache performance. Once again, combining transform space exploration and ISE provides a better result than either technique in isolation.

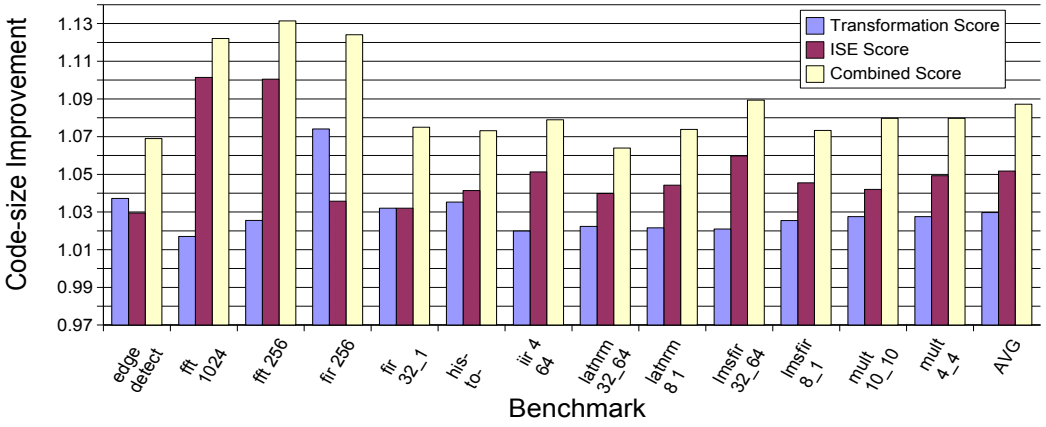


Fig. 6.7: Code-size Improvements achieved on the UTDSP benchmarks. The reduction in executable size can be an important factor for cache performance. Once again, combining transform space exploration and ISE provides a better result than either technique in isolation.

Regarding the number of samples considered in the results, it can be seen in figures 6.8, 6.9, 6.10, 6.11, 6.12, and 6.13, that the number varies between benchmarks. This is because some of transformation sequences either caused compiler integrity assertions to be triggered, or in a few cases generated incorrect code. Failing sequences are discarded, but there are still a great many remaining. Sequences found to produce correct behaviour are used to generate the results here.

In figures 6.4, 6.5, 6.6 and 6.7 we see that the average results for the SNURT benchmarks are noticeably higher than for UTDSP. The principal reason for this is that the SNURT benchmarks are smaller, so the potential selection space is smaller and thus better suited to uniform sampling. Although we only explore a tiny fraction of the overall search space, we still obtain very good results. It seems likely that exploring a larger portion of the search space will yield even better results, especially for larger programs. Larger programs are also likely to benefit from a more directed search technique that can quickly focus on more promising areas of the search space, such as that described in work by Franke *et al.*[47].

This experiment shows that in many cases simply choosing transformations that allow effective use of ISEs will not give good overall performance. Strong examples of this are the UTDSP FIR-256 and FIR-32_1 benchmarks, where the best combined performance seen is given by a set of transformations that did not allow any runtime improvement through ISE at all. Examples where combined performance is significantly better than either transformations or ISE alone are SNURT CRC and the SNURT FFT benchmark.

The graphs in figures 6.8, 6.9, 6.10, and 6.11 show the performance for each individual technique and the combination of the two for every sample point in the search space, for a small selection of benchmarks. The samples are sorted by the performance of combining transformations and ISE. This allows the ratio of transformation to ISE performance to be seen and lays bare the correlation between the two individual techniques. The correlation is seen where either the performance of both individual techniques improve at the same point or where one gets better but the other gets worse. An example of this correlation can be seen on the left side of figure 6.10. We can see sets of transformations which promote better ISE performance. The points perform poorly overall due to the transforms not contributing well to the total acceleration. AISE is apparently able to make up for poorly configured transformation, but that is not necessarily a good thing. Transformation comes at a far lower cost than ISE. Exploiting the maximum available (observed) optimisation in the software domain is far cheaper than blindly applying AISE to poorly performing code.

The coupling of transformation and ISE is important with regards to the slack-absorbing effect of ISE, but more specific dependencies also exist. A direct example of the coupling between transformations and ISE is shown in the motivating example, SNURT CRC, with the bit packing transformation. Sequences that make use of this transform are marked as short vertical bars in figure 6.8. It can be seen that all the best performing sequences make use of

this transformation. At the points where it changes from being turned off to turned on there is dip in ISE performance and a rise of about the same magnitude in transformation performance. Transformation performance improves greatly from negligible improvement up to 1.63x, and ISE performance recovers thereafter. So with appropriate supporting transformations the bit packing transformation allows both code performance on its own and good ISE performance for this benchmark.

Figure 6.9 shows an almost ideal set of results (for SNURT FFT1k), where the best set of transformation sequences when considered alone also allow the greatest gain from ISE. When the best sequences seen overlap in this way, the combined performance is considerably improved. Results for SNURT FFT1K move from peaks of 1.11x and 1.14x with individual techniques, to a peak of 1.28x with combined. Not all sequences give such clean results though, figure 6.8 (from the motivating example, SNURT CRC) shows patterns that are visible in several benchmarks. The best acceleration is given by finding a transformation sequence that both improves the runtime on the code and does not damage the ISE acceleration factor. The best sequence seen actually gives ISE performance slightly below that which was achieved on the baseline code, but overall performance is high due to good runtime improvements from the transforms themselves. Figure 6.11 show the results from a benchmark where almost none of the overall improvement comes from transformations but almost entirely from ISE (UTDSP latnrm-8_1). The graph does still show that whilst the transformations do little to improve acceleration on their own, ISE efficacy is still affected by the underlying code form.

Figures 6.12 and 6.13 show the performance of the best transformation sequence found so far as each point in the sample space is evaluated. These graphs are made monotonic without reordering the points so as to show the actual progress of the stochastic transform search. Figure 6.12 shows an example (SNURT JFDCTINT) having the characteristics which led to evaluating such a large number of samples in the transformation space. The graph contains several steps, with the highest step not being found until after several thousand samples were evaluated. This was not typical of most benchmarks; figure 6.13 is an example (UTDSP FIR) that exhibits the typical behavior. This graph also has steps, but they are much closer together and the maximum is encountered after about five hundred samples. None of the thousands of sequences evaluated after that point performed better. This evidence suggests that as with many stochastic sampling mechanisms, the return on the number of samples explored has an exponential decay. Considering a smaller number of samples than we have applied here should afford good results in many cases, but further performance can be gained through wider sampling.

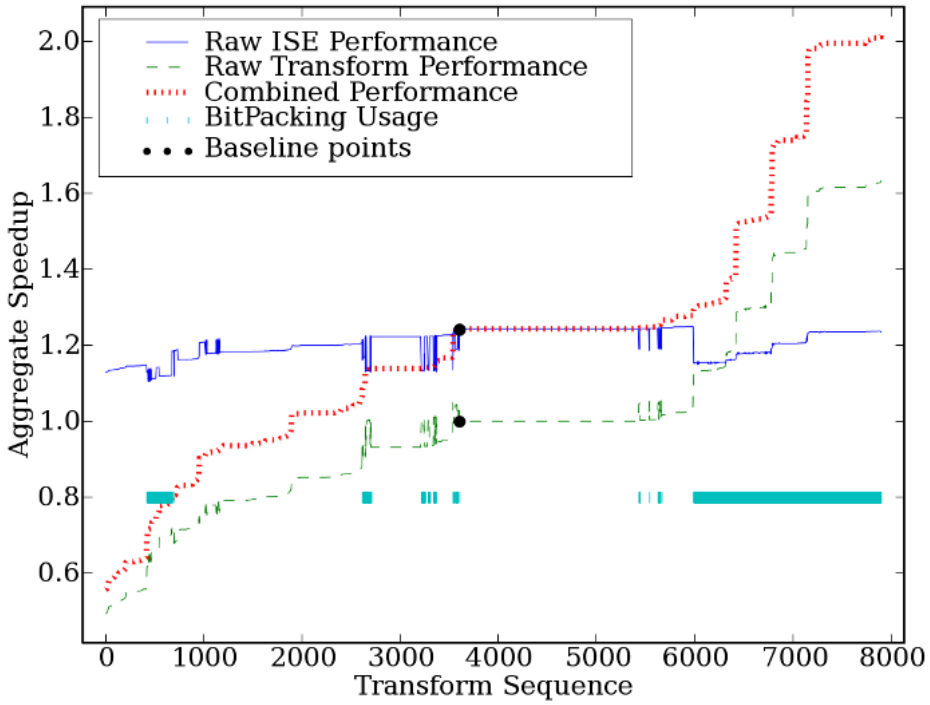


Fig. 6.8: Runtime Improvements Per-Transformation for SNURT CRC (based on 7896 runs). Turquoise tics indicate the points at which bit-packing is enabled. Bit-packing appears to be an enabling transformation in the last section of this graph; both transformation and ISE performance increase in tandem steps with the inclusion of bit-packing. Transformations which both accelerate the underlying code without ISE, and increase the relative acceleration of ISEs, are of particular interest in this study.

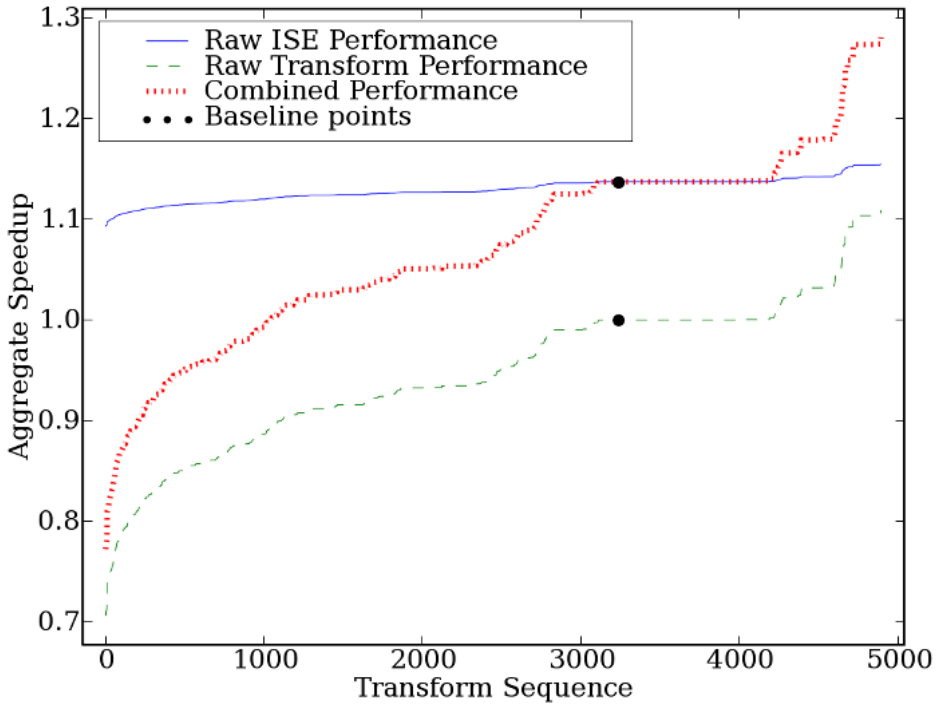


Fig. 6.9: Runtime Improvements Per-Transformation for SNURT FFT1K (based on 4892 runs). Once again there appears to be an enabling transformation, at the highest ten percent of the space. It was not possible to determine a single transformation responsible, so it is likely that a combination of transformations are in play simultaneously.

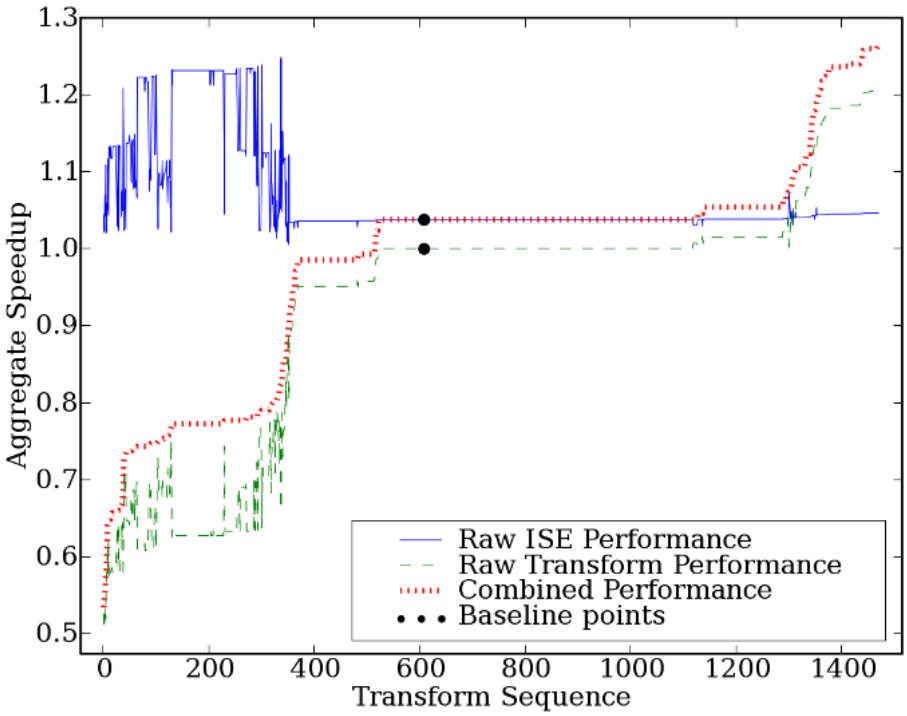


Fig. 6.10: Runtime Improvements Per-Transformation for UTDSP Edge Detect (based on 1470 runs). This graph demonstrates the shared exploitation of potential acceleration between ISE and transformations: where transformations are ineffective, ISE appear to "take up the slack". Whilst not apparent in all results, this does suggest that engineers must be careful not to present a badly transformed application to AISE. The result may well be accelerated by ISE that are apparently performing well relative to the bad code they are related to. ISE however are much more expensive than software transformation; flexibility being the very essence of software.

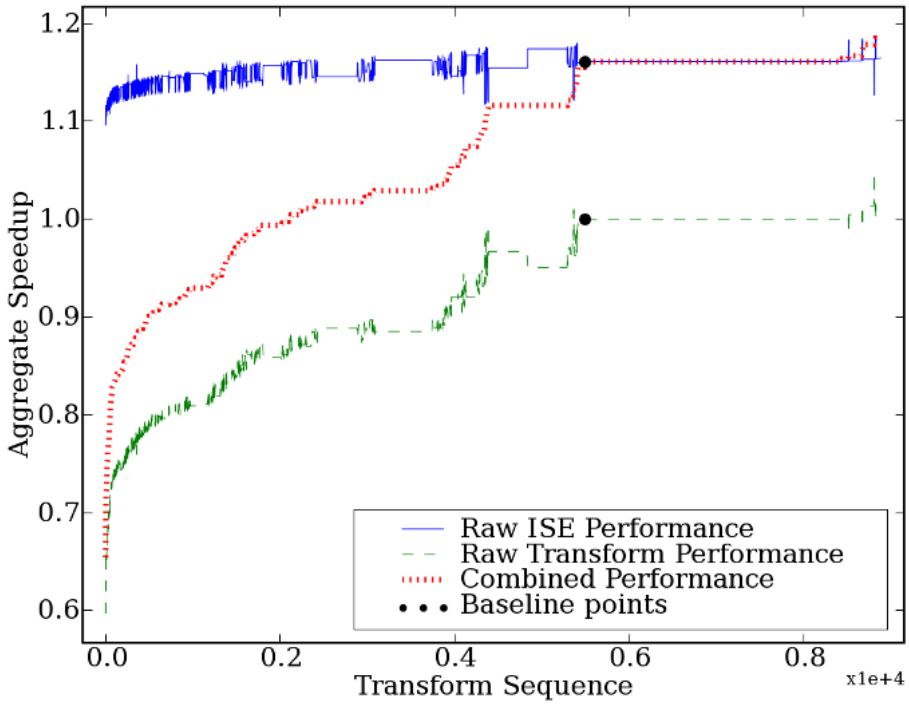


Fig. 6.11: Runtime Improvements Per-Transformation for UTDSP LATNRM 8_1 (based on 8882 runs). Once again ISE is offsetting the negative impact of bad transformations, at a considerable hardware cost. Transformations should be properly explored with regards to their impact on ISE, and on the application acceleration.

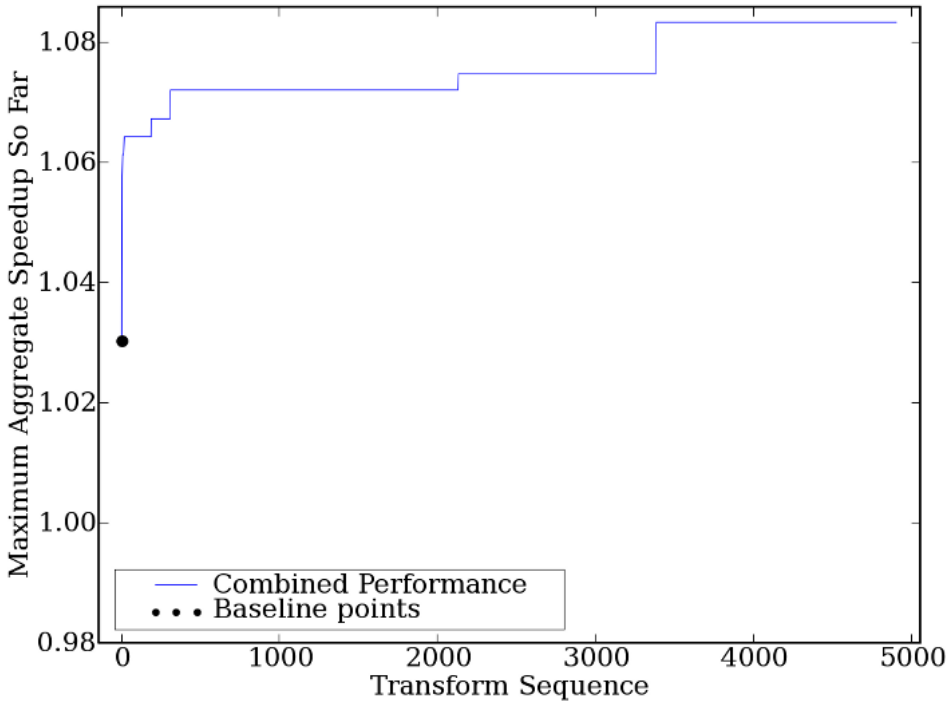


Fig. 6.12: Maximum performance found for each point in the sample space for SNURT JFD-CTINT. Performance within 10% of the maximum is obtained after 10% of the space has been explored, demonstrating that the quality of the best result observed is roughly logarithmically related to the number of points explored, on average.

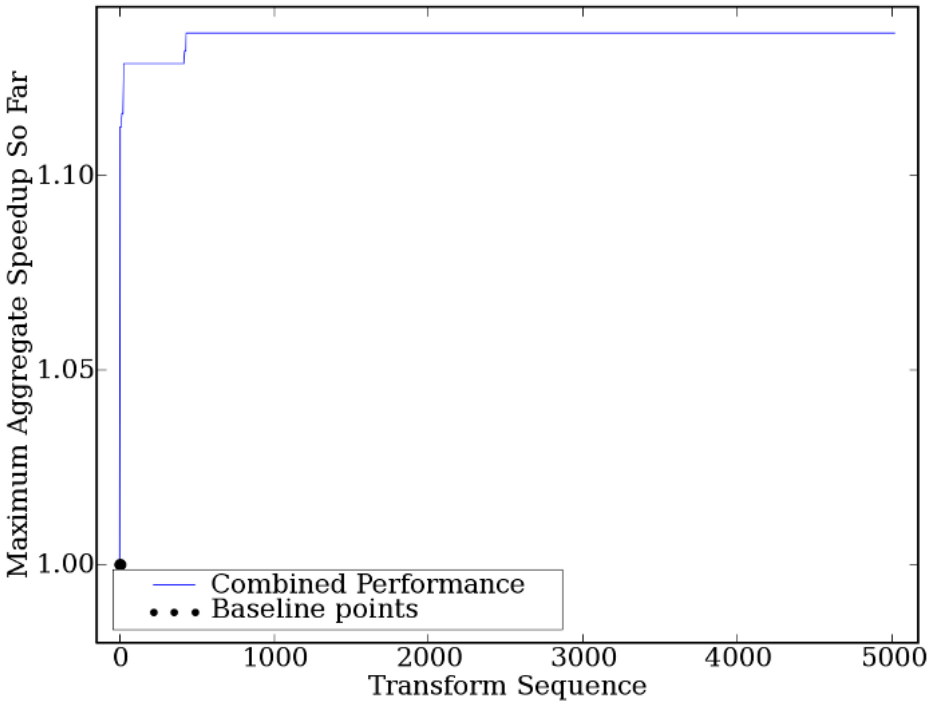


Fig. 6.13: Maximum performance found for each point in the sample space for UTDSP FIR-256. In this case the maximum performance is actually encountered within 10% of the total space explored. Once again this shows the stochastic sampling of many fewer points in the transform space should be similarly effective.

6.2.4 Conclusions

In this section a methodology for improved ISE generation is presented and evaluated. This work combines the exploration of high-level source transformations and low-level ISE identification. Several conclusions can be drawn:

- In this particular tool-chain and with the Atasu algorithm rather than ISEGEN as used elsewhere in this thesis, an average of 1.47x acceleration was achieved with the combination of ISE and source transformation.
- The combination of ISE and source transformation is greater than the sum of its parts in some cases; i.e. the best combined score is better than the product of the ISE and transformation improvement factors.
- There is a particularly strong need to perform good transformations when AISE follows after; AISE will often take up the slack where transformations are inefficient. It is far cheaper to perform source transformations than ISE.
- Source-to-source transformations are not only very effective on their own, but provide much larger scope for performance improvement through ISE generation than any other isolated low-level technique.
- Integration of both source-level transformations and ISE generation in a unified framework can efficiently optimize both hardware and software design spaces for extensible processors.
- Compared to previous work [37], we have covered a much broader array of existing transformations to get a more global picture of the potential for transformation in improving instruction set extension.
- We have empirically demonstrated that there exists a non-trivial dependence between high-level transformations and the generated instruction set extensions justifying the co-exploration of the HW and SW design spaces.

Due to the difference in tool-chain (and indeed AISE algorithm) between this and other work of this thesis, it is not possible to compare results directly. The above conclusions are apparent from the work done in this section alone.

Future work should investigate the integration of machine learning techniques based on program features into the transform-space exploration process.

6.3 Floating versus Fixed Point

This section attempts to address the design trade-offs inherent in selecting a number format for performing arithmetic requiring a decimal point, particularly in combination with ISE implemented using CFAs. A variety of notable points in the trade-off space are modeled and synthesised to obtain their performance, so as to evaluate them. Techniques developed in earlier sections 5.5 and 4.3 are included to determine their relative merit in such a scenario. *Floating point is demonstrated to perform worse than fixed point, for all design concerns and applications studied here, regardless of ISEs employed.*

6.3.1 Introduction

Digital Signal Processing (DSP) applications are able to use either floating or fixed point number formats in order to implement arithmetic including a decimal point. The two formats have an inherent trade-off in terms of the quality and cost of the result produced:

- Fixed point can be tailored for a specific precision over a covered range, depending on the position of the decimal point.
- Floating point has a greater dynamic range than fixed point.
- Floating point is generally easier to develop for than fixed point, the latter of which generally requires macros to be used instead of standard operators available in the language. This is particularly true in the most common embedded language: C. Additional format problems such as overflow, underflow, and rounding errors further complicate the issue.
- Fixed point requires only integer units to process, whereas floating point require the more complex and generally expensive floating point units. Software emulation exists, but is prohibitively slow.

Floating point is almost always as per the IEEE 754 standard [147], specifying either 32- (single precision) or 64-bit (double precision). In floating point, the mantissa and exponent of a decimal number is encoded in separate fields. The exponent defines where the decimal point lies, hence the name of the format. An additional bit specifies whether the number is positive or negative. Fixed point has a constant exponent, leaving more room than floating point for a given bit-width to encode the mantissa. Floating point is generally best used in cases where data is non-uniformly distributed over the entire range covered by the format. In essence, floating point numbers are the "general purpose" decimal number format, and for some considerable time hardware support has generally only been available for the IEEE-754 standard single- and double-precision formats. Fixed point on the other hand can in many cases be tailored to a greater precision for a specific range, and it is this that contributes to the engineering costs associated with writing software utilising fixed point. Much like the specialisation of

ISE, the use of fixed-point numbers must be specifically targeted to the application (and even expression) with which they are concerned. In this manner, it is possible that fixed point is both more precise and more accurate than floating where it has been properly engineered. Floating point is still very popular with scientific and other more abstract applications, where the data is not necessarily well understood or the engineering costs of producing a suitably balanced fixed point number format outweigh the benefits from added precision.

Hardware tends to vary in its support for floating point, due to the expense of including such a unit in a processor design. Any decent GPP intended for desktop computing in 2010 will include floating-point support, usually including multiple FPU in order to exploit data parallelism. Embedded processors on the other hand tend to take a more conservative approach, but examples of high-spec floating-point DSP do exist. An example on sale at the time of writing this thesis is the TigerSHARC ADSP-TS101S: a super-scalar DSP capable of performing six simultaneous floating-point operations in a single cycle [148]. The engineer-time cost of floating versus fixed point is hard to quantify, so this is not approached here. What is possible to quantify is the performance space available through using either floating or fixed point for a particular kernel or application. Towards this end, this section presents a thorough evaluation of run-time, energy, power, and area performance in both a baseline and several extended EnCore-based designs. This gives insight into the combination of ISE and decimal number format, which can be further applied to make informed design decisions in future efforts.

6.3.2 Methodology

It is fairly difficult to present an entirely equivalent example of fixed versus floating point applied to the same application due to the engineering costs of producing both versions in parallel. We have, however, already encountered one application in this work which contains such configurability: FAAD. In addition, the DSPStone [149] benchmark suite contains fixed- and floating-point versions of each of the kernels implemented. For the purposes of this evaluation, the combination of FAAD and the DSPStone benchmarks configured for either 32-bit fixed- or floating-point are used to represent the two number formats for evaluation.

The same tool-chain and flow are used as in previous sections of this thesis where energy is to be derived (sections 4.2 and 5.5). The EnCore is once again employed here as the baseline core for the purposes of measuring relative merit of using ISE.

The benchmarks from the DSPStone suite and FAAD are treated somewhat differently with regards to the design space evaluated, due to FAAD's much larger code-base and the more pronounced trends that are therefore visible. The design points evaluated for each benchmark in the DSPStone suite are all combinations of:

- Fixed and floating point.
- I/O constraint: 4/4 and 8/8.

- Area constraint: 0.2mm^2 and unlimited.
- Energy heuristic (section 5.5) and combinational heuristic (section 4.2.2).
- With Staggering (section 4.3) and without.

These configurations of the synthesis flow (*uarchgen* and *isegen* tools) lead to thirty-two designs per DSPStone benchmark which are further processed by DesignCompiler using a 130nm library, as per previous similar experiments in this thesis. The 0.2mm^2 is not an arbitrary constraint, but rather is the size of the combinational area consumed by the baseline EnCore excluding memories. This was added into the points for evaluation since prior experiments in this thesis have demonstrated that without an area constraint, the design trade-offs can be expensive. With FAAD, a different approach is warranted due to the greater size of the application. Instead of just sampling with the area constraint at 0.2mm^2 and unlimited, design points are taken from 0.2mm^2 up to 3.0mm^2 in 0.1mm^2 increments. Where further area is expendable, sampling runs from 4.0mm^2 up to the maximum in 1.0mm^2 increments. FPU area (0.132mm^2) and power (27.16mW) are included in results for designs which utilise scalar floating point. This is so that the relative cost of using the standard floating point unit can be seen versus the baseline of integer units only employed in the fixed point designs. Results are grouped into data series for display as graphs in evaluation:

- DSPStone benchmarks have four series, each series having constant number format and whether or not staggering is used.
- FAAD has sixteen series, each series having constant every variable except for area constraint.

This grouping allows for distinction to be more easily visualised between the different combinations of specification detailed above.

All ALUs used here are taken from the Synopsys DesignWare library, with hardware latencies normalised to a 250MHz clock (4ns). Software latencies are configured to match the Calton revision of the EnCore [10] processor.

The *uarchgen* selection heuristic, that used to select the set of ISE templates to implement as CFAs, is still of the original form detailed in algorithm 2. The energy aware heuristic has only so far been applied to the *isegen* tool, leading to potentially misguided results at area constraints which are not unlimited. This effect is examined in more detail in the following subsection.

6.3.3 Evaluation

The principal observations required of this experiment are the design performance with regards to the number format applied to DSP kernels when ISE is considered. Floating point is now shown to be the more expensive option in all axes, both with and without ISE.

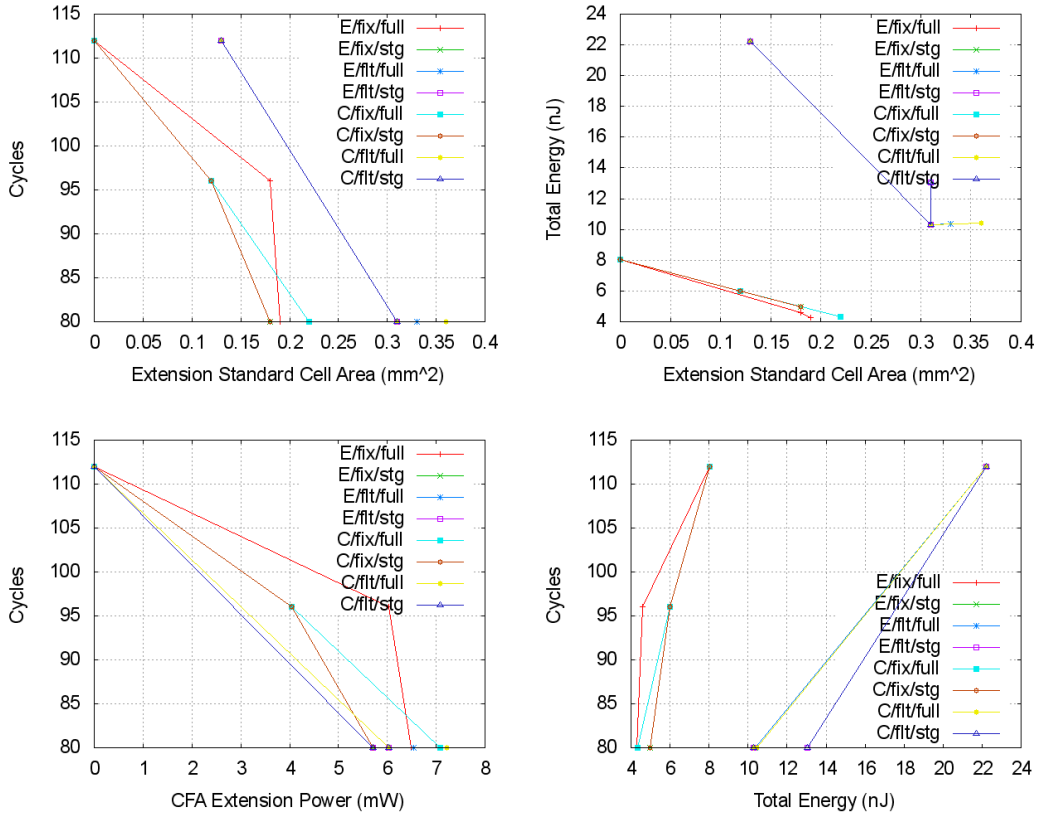


Fig. 6.14: DSPSTONE CONVOLUTION design points demonstrating cost versus performance. This is the simplest benchmark used here, and the trends are similarly simplified. Floating point is shown to be more expensive in all axes, especially energy. This is a trend which is continued throughout all of the benchmarks tested.

DSPStone Kernels

The primary concern in most ISE studies is that of the acceleration factor obtained in utilising ISE with various constraints versus a baseline core. For this work however, absolute cycle counts are considered since these allow for the dual baselines of fixed- and floating- point.

When looking at the graphs, towards the bottom-left is generally better performance, and towards the top-right is worse. We can see from the graphs for area versus application cycles that in all cases, the floating-point designs are slower for a given extension area, versus the baseline core. Whilst the additional area incurred by the scalar floating point unit is included in these areas, the number format itself is slower and this leads to one major observation from the area-cycles graphs: At no point do the fixed- and floating- point series intersect, regardless of heuristic or staggering. For the given configurations, there was not a single case where the floating-point design was faster-executing (in absolute cycle terms) than a fixed-point design of equal or less area cost. This must therefore lead to the conclusion that in terms of area and cycles, floating-point is always inferior in terms of cost for benefit. In itself, this result will not surprise many as this has been demonstrated in a scalar context before; this experiment serves to extend the observation to designs incorporating ISE.

The difference between the baselines on floating- and fixed-point ranges from nothing (in the CONVOLUTION kernel), to a 29% (93 cycles) difference in runtime in the MATRIX1x3 kernel and a 15% (4000 cycles) difference in runtime in the MATRIX1 kernel. The latter two observations are the greatest observed DSPStone baseline relative and absolute differences respectively. Before ISE has even been considered, in some cases the consideration of whether to used fixed- or floating- point should be coloured by the difference in baseline runtime between the two being around the same magnitude as expected from ISE. This further supports the conclusions made in section 6.2.4, that it is far more important to evaluate as wide a section of the HW/SW co-design space, than it is to consider a few features exhaustively. This result will also not come as a surprise, as it is a foregone conclusion of Amdahl's law.

From the results presented here, there are a range of different trends with regards to the effect of ISE on the benchmark depending on the number format used. The most neutral result contains no difference at all between the floating- and fixed-point versions maximum and minimum cycles (112 and 80 cycles respectively). The difference in area implementation if the scalar FPU is considered puts the fixed-point design at an advantage once again even in this seemingly neutral case. In all other benchmarks as previously noted, the baseline execution-time of the floating-point version is worse than the fixed-point. There is also an apparent tendency for the absolute improvement in cycles to be either the same or less for floating-point versus the fixed-point alternative. These two points in combination lead us to realise that fixed-point is invariably the better design option with regards to the execution speed, regardless of the ISE applied. In most cases (FIR2DIM, IIR, LMS, MATRIX1, MATRIX1x3, MATRIX2, N REAL UPDATES), the fixed-point benchmark not only started with a lower execution time

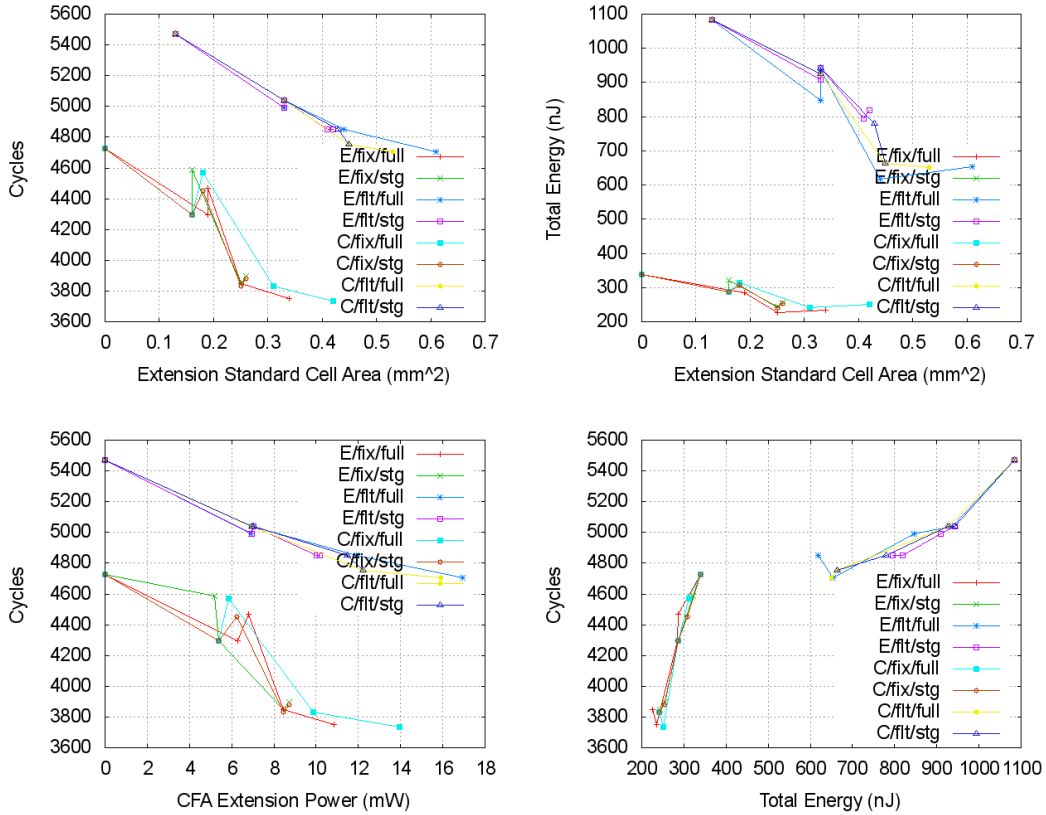


Fig. 6.15: DSPSTONE FIR2DIM design points demonstrating cost versus performance. Confirmation of the results of section 5.5 can be seen in all of these graphs by comparing the **C**ombinational series to the **E**nergy-aware series, without staggering (full). The latter has better acceleration, power, area, and energy.

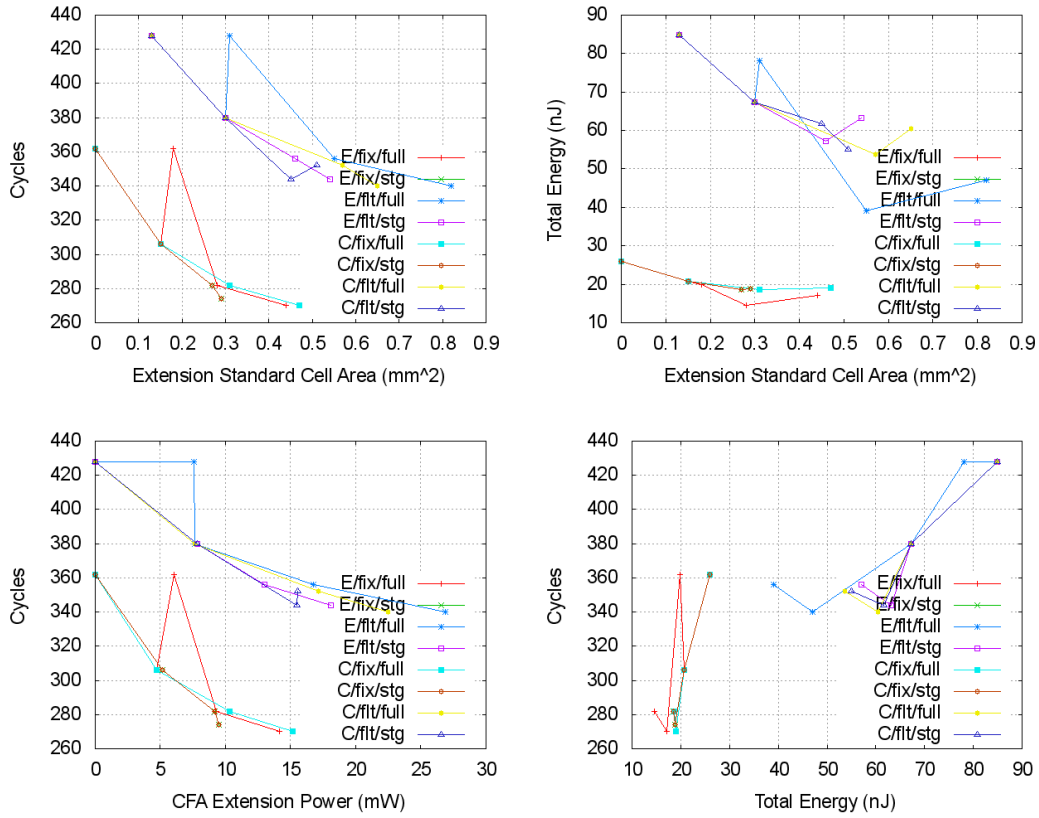


Fig. 6.16: DSPSTONE IIR BIQUAD N SECTIONS design points demonstrating cost versus performance. The jaggedness of the series is because both 4/4 and 8/8 I/O are included in each. In this case, both floating- and fixed-point fail to produce a useful acceleration at 8/8 I/O, with the energy heuristic, and a 0.2mm^2 constraint. The limit is too severe to enable acceleration at this point with this heuristic.

but after unlimited AISE the absolute (and relative) difference in cycles had increased yet further. The only benchmarks evading this effect were CONVOLUTION which is too simple, and N COMPLEX UPDATES in which there is a particularly parallel structure which is better exploited in the simpler IR expression semantics of the floating-point. The result that floating-point is always lower performance for a given extension area would not be the case if the results presented here did not consider the FPU as part of that area. Several benchmarks would find intersecting series (meaning that floating-point could be considered of equal or better value at some point in the trade-off curve) if the area of the FPU was not added to the results. In all of these cases however, the fixed-point version eventually becomes the best performing solution at maximum area utilisation since regardless of the area additions (shifting the floating-point series to the right). The cycle counts are equivalent regardless of this shift, for limit comparison.

The energy consumed for a particular design point depends on the power of any CFAs used, the power of the baseline core, and the time the application takes to execute in those two domains. Due to the higher (more than double) baseline power consumed in the FPU-

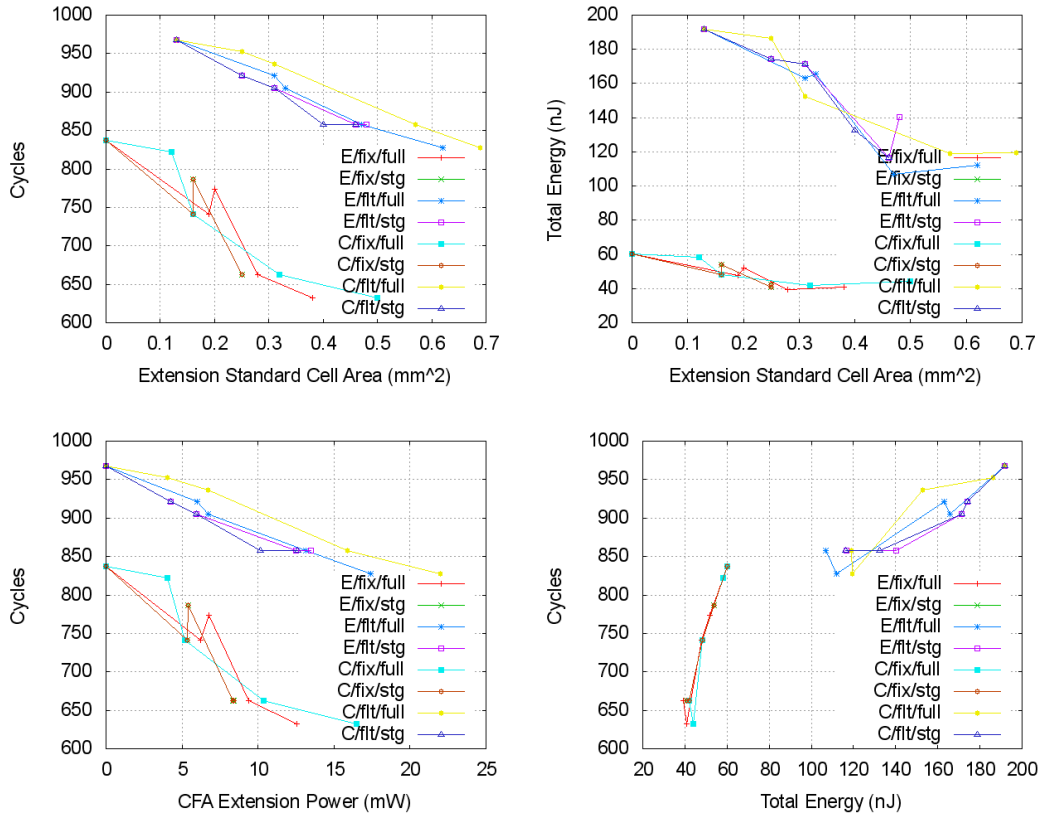


Fig. 6.17: DSPSTONE LMS design points demonstrating cost versus performance. Here in particular we can see the similar effects, of using the original heuristic and staggering, versus the **E**nergy-aware heuristic without staggering. With regards to acceleration and area, staggered points are slightly less accelerating and slightly less large. The energy result still comes out in favour of the energy-aware heuristic, demonstrating that it is more suited to energy optimisation than staggering.

inclusive design and the higher cycle count in most cases, the floating-point designs all tend towards a higher energy for both area and cycle costs versus fixed-point. In terms of the energy-saving made by using CFAs versus the baseline, the floating-point wins out in both relative and absolute terms in most benchmarks. Due to the higher power consumption of the floating-point baseline, removing cycles has a greater impact on the energy consumption than for the already lower-power fixed-point baseline. Because the integer-only (fixed-point) EnCore baseline is already so frugal with regards to power and energy consumption, it obtains less improvement when AISE is employed. The lesson here is that if you are already committed to using floating-point in a design, AISE will achieve a greater energy benefit than if you had been using a fixed-point format.

On average, the baseline of floating-point is 3.22x higher energy consumption than the fixed-point equivalent. The smallest difference is 2.76x in the CONVOLUTION benchmark once again, as may be expected given the lack of difference in runtime at the limits of execution time. The greatest difference is 3.91x in the MATRIX3x1 benchmark, which has the

smallest baseline and accelerated cycle count after CONVOLUTION; this suggests that smaller benchmarks are less predictable in their energy cost. At the opposite limit (i.e. the greatest area utilisation), not all of the benchmarks obtain the greatest energy improvement. Once again this demonstrates that there is a diminishing return at higher areas which can reverse a benefit to a disadvantage if followed too far. In all of the DSPStone results, the greatest area represented is the design which includes all identified ISEs for that benchmark. Benchmarks which have their best energy performance at this unlimited point are for the fixed-point series CONVOLUTION, IIR, LMS, MATRIX1, MATRIX1x3, MATRIX2, N COMPLEX UPDATES. Similarly those having their best energy performance at the area limit in their floating-point series are MATRIX1, MATRIX1x3, and N COMPLEX UPDATES. The N REAL UPDATES benchmark does not achieve this in either fixed- or floating-point series, but the energy results at the area limits are rather close to the best obtained. Importantly, for none of the DSPStone benchmarks examined is there a point at which the energy performance is *worse* after ISEs/CFAs are utilised. This would seem to be at odds with earlier results in section 4.2, but due to the small size and lack of software transformations applied here to the DSPStone benchmarks there are very few ISEs that could be considered extraneous, or that would worsen an energy result. When we progress to evaluation of the FAAD application, the power-hungry effect on energy returns with a vengeance.

We can now move on to examine more specific energy effects of ISE with regards to both number formats. Relative to the baseline, the average energy improvement made at the area limit for fixed-point is 1.52x, and for floating-point is 1.70x. The highest energy improvement obtained is 1.86x (N COMPLEX UPDATES) for fixed-point and 2.16x (CONVOLUTION) for floating-point. As was noted in an earlier evaluation, the difference is mostly down to the energy consumption of the baseline in floating-point being higher. Despite the fact that extension logic then goes on to use floating point units which themselves are higher power than integer ones, the reduction in runtime outweighs this disadvantage insofar as energy is concerned. This is examined in more detail further down the page. Another interesting observation is the difference between floating- and fixed-point energy consumption using the lowest energy CFA design for each format. The average relative difference is then 2.87x which is lower than the baseline 3.22x; the application-specific nature of the CFA inclusive designs has narrowed the gap between the floating- and fixed-point alternatives in terms of energy consumption. The smallest difference observed is again in CONVOLUTION, of 2.07x; this benchmark also had the least energy difference between formats at the baseline, a difference of 2.77x. The maximum difference observed is again MATRIX3x1 at 3.21x, notably less than the previous (baseline) average and less than the previous maximum of 3.91x, also from MATRIX3x1. We can therefore conclude that whilst the CFA-inclusive designs narrow the gap between fixed- and floating-point energy performance, they cannot yet close it entirely. In terms of energy cost, the fixed-point designs remain the most cost-effective regardless of application-specific design,

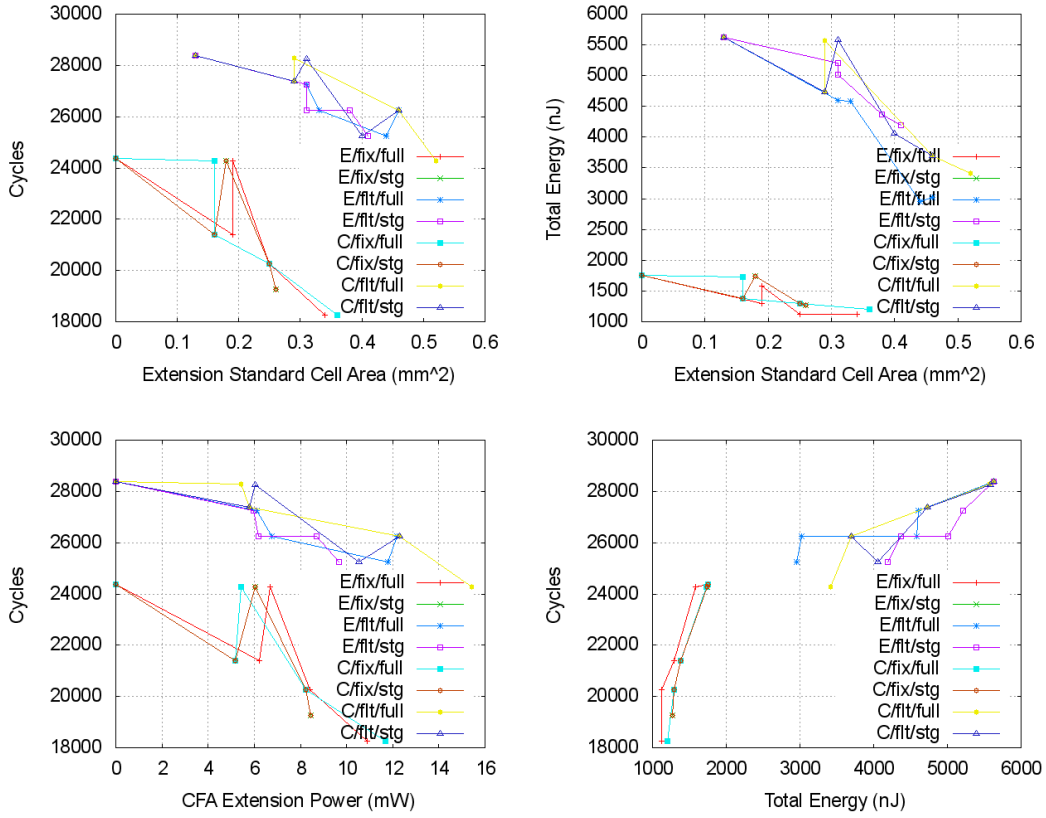


Fig. 6.18: DSPSTONE MATRIX1 design points demonstrating cost versus performance. Here again we see barely effective design points at 0.2mm^2, for both fixed- and floating-point. The conditions are identical to the same observation made in Figure 6.16, except this time they occur for the combinational heuristic both with and without staggering in addition to the energy heuristic without staggering.

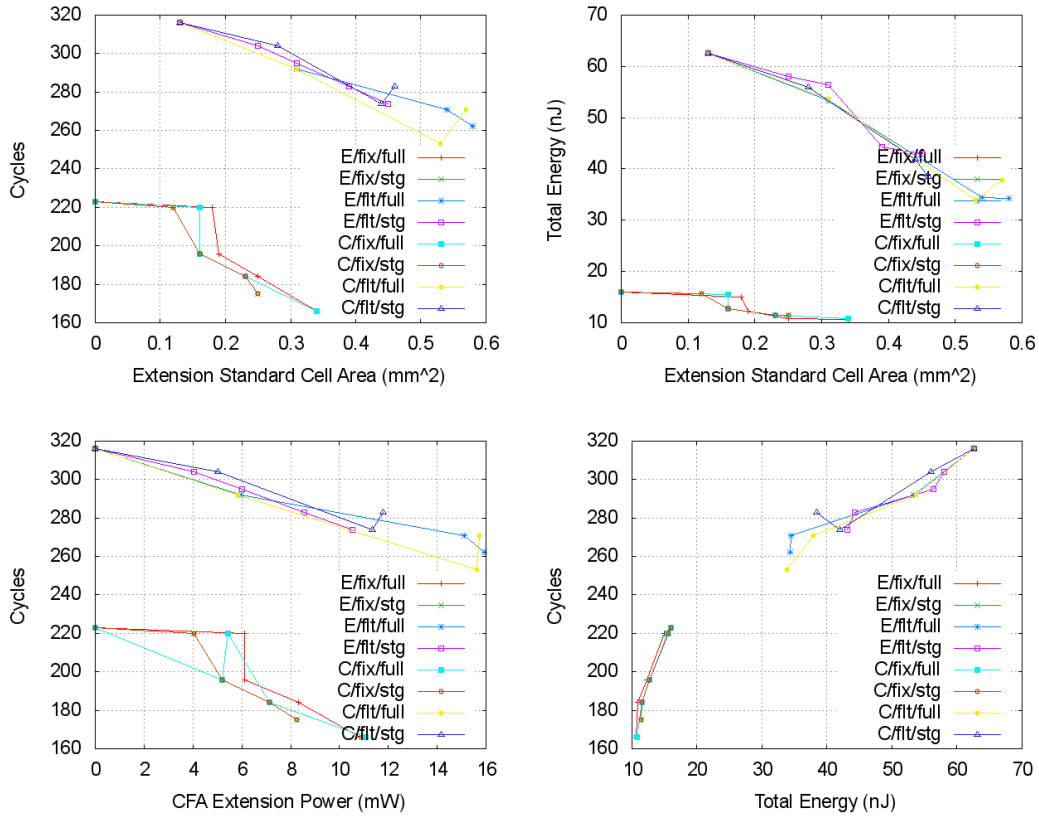


Fig. 6.19: DSPSTONE MATRIX1x3 design points demonstrating cost versus performance.

although ISE does appear to help mitigate the difference by producing greater energy savings for floating-point than fixed-point.

Further observations regarding the energy performance can be made by looking at the trends in the graphs for energy versus cycles and area. We can see in the graphs that in all cases the area trade-off is more beneficial at the limits (e.g. worst and best-case) of energy performance. This does not mean that throwing unlimited area at the problem is actually the best case. We can see that in most of the graphs, the designs utilising greatest area are usually a little less efficient with regards to energy consumption than designs using slightly less. This trend of diminishing and inverting returns is a lot more pronounced in the floating-point data series, due to the more power-hungry nature of those CFA designs for a specific area limit (see conclusions of section 4.2). Not all benchmarks suffer from this effect however, in particular the MATRIX1 and MATRIX1x3 continue to benefit in both cycles and energy up to the limit of area consumed. From the perspective of cycles a related trend can be observed; The designs which take the least number of cycles are not always the designs which have the least energy consumption, as often the power required to scrub out the last few cycles is linearly disproportionate to the runtime saving made. Again, this effect is most noticeable in the floating-point series: where it is visible in the fixed-point equivalent, the inverting return on cost is very

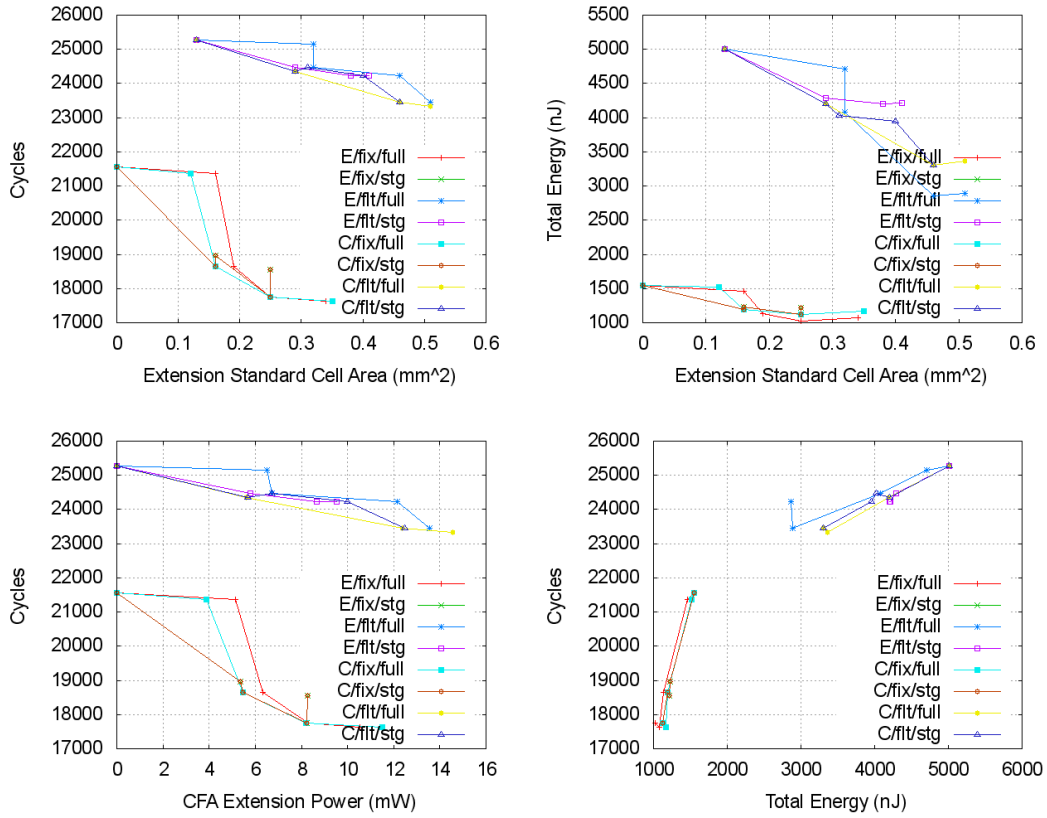


Fig. 6.20: DSPSTONE MATRIX2 design points demonstrating cost versus performance.

minimal.

We can observe a confirmation of the earlier simple linear relation between power and area by looking at the graphs of power and area versus cycles: these graphs in all cases are nearly identical in shape, demonstrating that the simple constant factor derived earlier for power estimation is indeed valid. Because the graphs for power do not include the additional power of the FPU, we can see that in some cases CFAs consume less power for a given area in floating-point rather than fixed-point. Once the additional FPU power and runtime is considered, the result is the energy observations noted: floating-point invariably consumes more cycles, area, power, and energy than the fixed-point alternative regardless of application-specific processing introduced.

Staggering (see section 4.3) was included in this experiment for completeness, to determine whether it would have much effect when combined with small unexpanded kernels. In this case it seems to have a fairly small effect on most of the kernels, with the same general trend as has been observed in prior experiments. The staggering mechanism is intended to act in the face of deep and very numerous ISE templates. Despite the minimal nature of the kernels examined herein, their data-flow provides similar opportunities as the benchmarks considered in section 4.3 in terms of the cost-benefit improvement from staggering. Staggering can be seen here

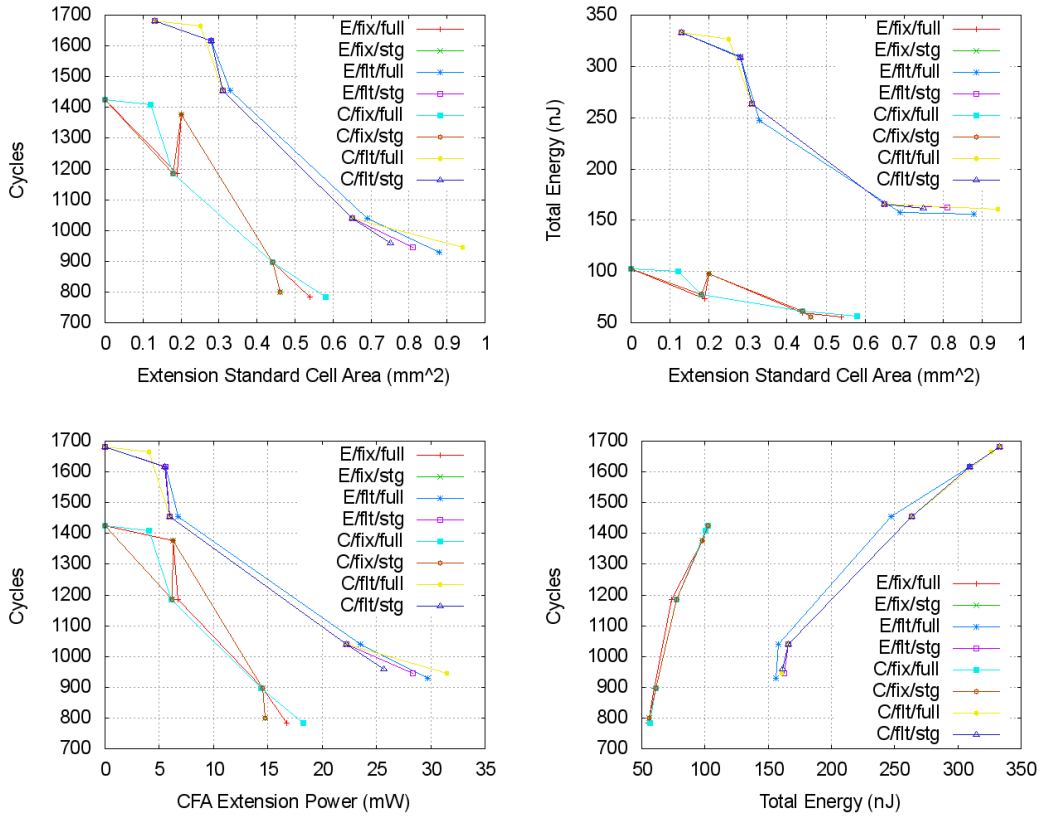


Fig. 6.21: DSPSTONE N COMPLEX UPDATES design points demonstrating cost versus performance.

particularly strongly in the graphs of cycles versus area; series where staggering is applied end earlier in the area axis. A small increase in the number of cycles can be noted alongside this area reduction, for reasons covered in section 4.3. Examples of the effect of staggering can be seen in all graphs of cycles versus area; perhaps the simplest is CONVOLUTION in which we can see the fixed-point series bifurcating between staggered and not, with no difference in acceleration but an 18% saving in area for the staggered design point.

In terms of the efficacy of staggering on the different number formats, the relative effect seems to be roughly the same per-benchmark regardless of number format. This effect can be observed in the visual similarities of the two formats' graph series: you can observe roughly the same progression in both the fixed- and floating-point series. Staggered series end earlier (in the area domain) than their full counterparts, and in all cases the effect is roughly the same in both fixed- and floating-point after the series are transformed to account for the differences in the formats' series sizes.

As discussed earlier, the energy aware heuristic from section 5.5 has been applied in addition to the original ISEGEN heuristic in order to confirm the results of the earlier experiments under different conditions. Although the results for the DSPStone kernels here are not as pronounced as those seen in the earlier section, the objective of the energy aware heuristic to reduce total energy is alive and well. For fixed-point in all cases the energy heuristic (without staggering) generated the lowest energy design. For floating-point in all cases but MATRIX1x3 the same trend is present, and in the case of MATRIX1x3 the original heuristic is only ahead by less than 1%.

In many of the benchmarks, we can see a tendency for staggering to overwhelm the energy-aware heuristic. This reduces the ISEs to a set nearly equal in performance, to the set resulting from staggering over the original heuristic results.

We now move on to looking at a larger slice of the trade-off space with a more fully fledged application: FAAD.

FAAD Application

When looking at the graphs for cycles versus area and cycles versus power (figures 6.23 and 6.26), the classic exponential decay of return on area (and the linearly related power) is visible in all series. We can see from the similarity in shape of these two graphs that the area and power are once again very closely linked, further supporting the conclusions of section 4.2 later used in section 5.5. It should be noted this area-power relation is not the product of the modelling performed in this experiment to derive energy, but rather are taken from PowerCompiler simulation of a representative set of ISEs executed in the CFA under examination.

Perhaps the most stark difference between series under the same number format is the original heuristic at 8/8 I/O in both cases, versus every other series for that format. The same exponential curve shape is present in even these particularly elongated series, with the more

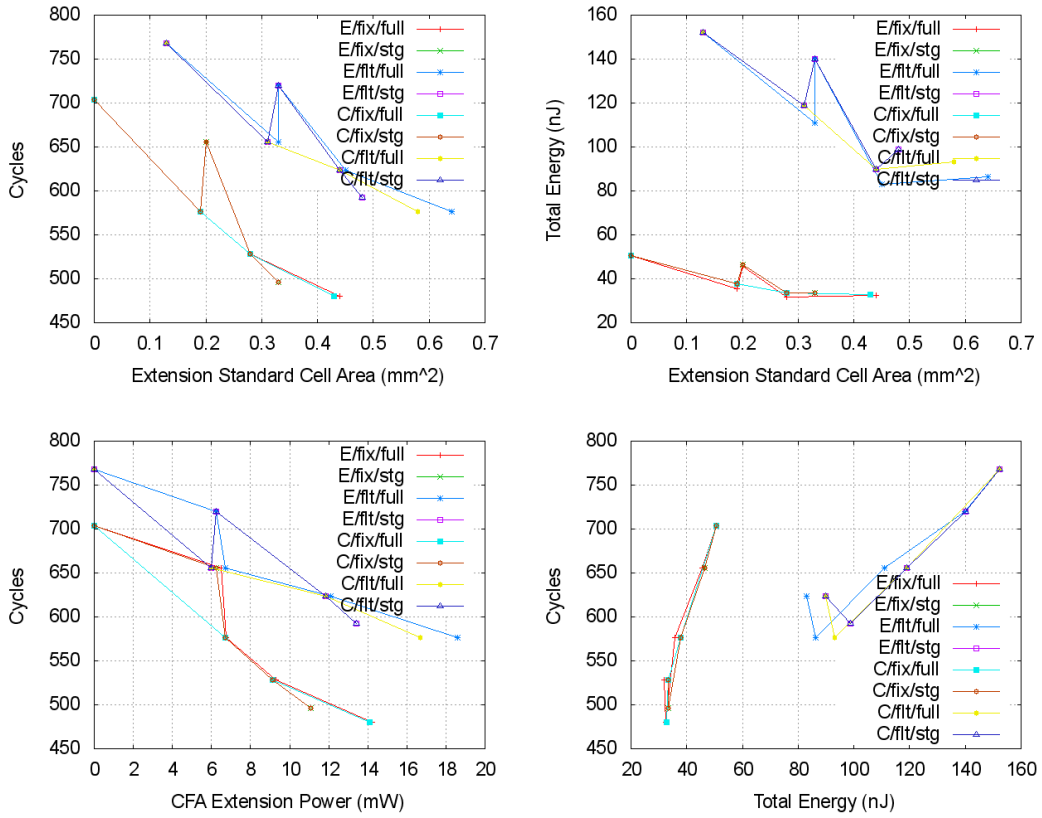


Fig. 6.22: DSPSTONE N REAL UPDATES design points demonstrating cost versus performance.

erratic trend coming from the selection algorithm having to choose between a smaller number of larger ISEs. Due to the large size of the FAAD application, a great deal of varied complexity is present in these instructions which can lead to the selection algorithm behaving rather worse than normal. We can see that the series in both cases peaks at around 13mm^2; more than double the next nearest for each format. Once again, the complexity of larger ISEs has amounted to a greater cost with little actual benefit over smaller ones, and the idea of "bigger is better" that has been espoused by some (e.g. [18]) is well and truly debunked.

In combination with the results obtained in the earlier sections (especially section 4.2), we can see a trend with regards to the trade-off between I/O constraint, extension area utilisation when unlimited, and the acceleration produced. First considering only I/O and the limit of acceleration, we can see from section 4.2 that an I/O of 12/8 leads to a maximum acceleration of around 52% (for the fixed-point FAAD). From this experiment we can see that I/O of 4/4 yields a maximum of 59.8%, and an I/O of 8/8 yields a maximum of 68.31%. The seemingly anomalous observation that the 12/8 result is lower than the 4/4 and 8/8 is due to the lack of good heuristic weighting in section 4.2. We can see from this that being excessively lax with regards to I/O constraints does not necessarily impart much in the way of acceleration, and that other factors such as the efficacy of the search can yield far greater sway over the quality of the resulting design. As with area, there are diminishing returns on relaxing I/O constraints in many contexts. The relaxed I/O constraints generally lead to larger and more complex ISEs being identified. Larger ISE are harder to share resources between effectively for reasons discussed in section 4.3. If one considers staggering to be the reduction of complexity through reducing the depth of ISE DFG, then reducing I/O is the reduction of complexity through reducing ISE DFG width. Here we see that the 4/4 fixed-point non-staggered area-unlimited design consumes only 5.01mm^2. The 8/8 equivalent consumes 13.42mm^2. The I/O 12/8 templates from the earlier section 4.2 had a less effective acceleration by over 10%, and their non-staggered area-unlimited implementation consumed 11.9mm^2 gate area. Relaxing I/O constraints therefore is not as important as getting the search algorithm right. There is a lot of merit in restricting the number of I/O ports during identification as whilst the acceleration limit is slightly increased for higher I/O, the resulting cost/benefit with regards to the area needed to reach that limit is considerably worse.

Staggering can be seen here as a shortening of the tail in the various series, ultimately bringing down the complexity of the underlying ISEs by breaking deep ISEs up into a series of more shallow ISE intended to be executed in series. The efficacy of staggering where area is unlimited is demonstrated in section 4.3, and here we can see again that it brings down the area significantly towards the limit for each series. The 4/4 limit for fixed point is brought down from 5.01mm^2 to 1.32mm^2, with a reduction in acceleration from 59.8% to 59.6% which is so small as to be insignificant. The same 4/4 limit for floating point is brought down from 5.94mm^2 to 2.53mm^2 with an acceleration reduction from 54.35% to 53.9%, which is nearly

identical behaviour to the fixed equivalent. The 8/8 limit for fixed point is taken from 13.42mm^2 to 2.34mm^2 and 66.34% to 64.28%; the equivalent for floating point goes from 12.53mm^2 to 3.73mm^2 and 71.54% to 67.48%. Staggering hurts the performance of the higher I/O a little more, but this is because the higher I/O is far more likely to have had a larger depth on the more critical ISEs. Combination of staggering and the energy heuristic is examined a little later.

Reducing the I/O limit brings the knee of the graph closer to the y-axis, and staggering brings the area-limit closer to the y-axis. Further investigation is required to determine why staggering does not also always bring the knee towards the y-axis, as this result was expected at this point. This result could suggest that staggering is not always something which is effective at a particular area utilisation, but rather something which is effective at a certain diversity of complexity in DFGs implemented on CFAs. The effect on the original heuristic at 8/8 I/O for both number formats is inarguable: staggering both brings down the area-limit and brings the knee towards the y-axis. One potential reason for the disparity here is that DFG which are particularly deep are not particularly high in merit due to a lack of data-parallelism: These DFG are therefore not considered for inclusion until the more expensive area levels, at which point they make very little difference (the 0.2% seen earlier, for example). In order for staggering to be of greater effect and to move the actual knee of the graph, DFG need to be *both* wide *and* deep. Achieving such properties may be more prevalent in the face of new heuristic vectors tuned to such conditions (favouring deep ISEs as much as wide); this again is left for future efforts to answer. Having a higher I/O constraint will, however, increase the width of what is in many cases approximate to a binary tree; the depth of a binary tree is relative to the number of leaf-nodes, so increasing the I/O constraint from 4/4 to 8/8 should double the depth of binary trees, and hence provide twice the depth for staggering to operate over.

As far as the new energy heuristic is concerned, this single application explored in this manner does a great deal to demonstrate the better search behaviour undertaken. We can see two things which more than anything lift this approach above the original combinational one, even when staggering is considered:

- The extent of the reduction in total area for the energy heuristic versus the original heuristic for 8/8 at unlimited area is 13.42mm^2-3.58mm^2 for fixed point and 12.53mm^2-4.63mm^2 for floating point; this comes to 3.74x and 2.70x respectively. The original heuristic plus staggering is of a similar magnitude, but is slightly more efficacious in this regard when considered alone, however:

- The acceleration obtained when using the energy heuristic instead of the original heuristic is greater at the lower area in the result of the former. The peak for acceleration for the original heuristic at 8/8 is 66.34% for fixed-point and 71.54% for floating-point. The peak for acceleration for the energy heuristic at 8/8 is 68.38% and 72.64%. Whilst these results are not impressive when considering only the acceleration afforded, once you re-

alise that this is done in conjunction with a 3x reduction in area and all resulting energy implications, the true worth of the energy heuristic is apparent.

The difference between area limits with and without the energy heuristic is somewhat interesting, but hard to make absolute conclusions from with only this one benchmark. The original heuristic gives a smaller area limit for floating than fixed, whereas the energy heuristic is the other way round. This is an interesting effect, suggests that the energy heuristic is more in tune with engineer expectations wherein a floating point design would consume considerably more area.

Once again we can see that basically the same trends exist for floating and fixed point arithmetic, and that the two versions of the benchmark achieved roughly the same performance in terms of relative improvement from ISE versus their baselines. From this we can conclude that the previously assumed engineering heuristics regarding the use of floating point stand even in the face of ISE: Floating point is for those who cannot afford the extra software engineering time of fixed-point, and fixed point is for those who cannot afford the extra hardware cost of floating-point.

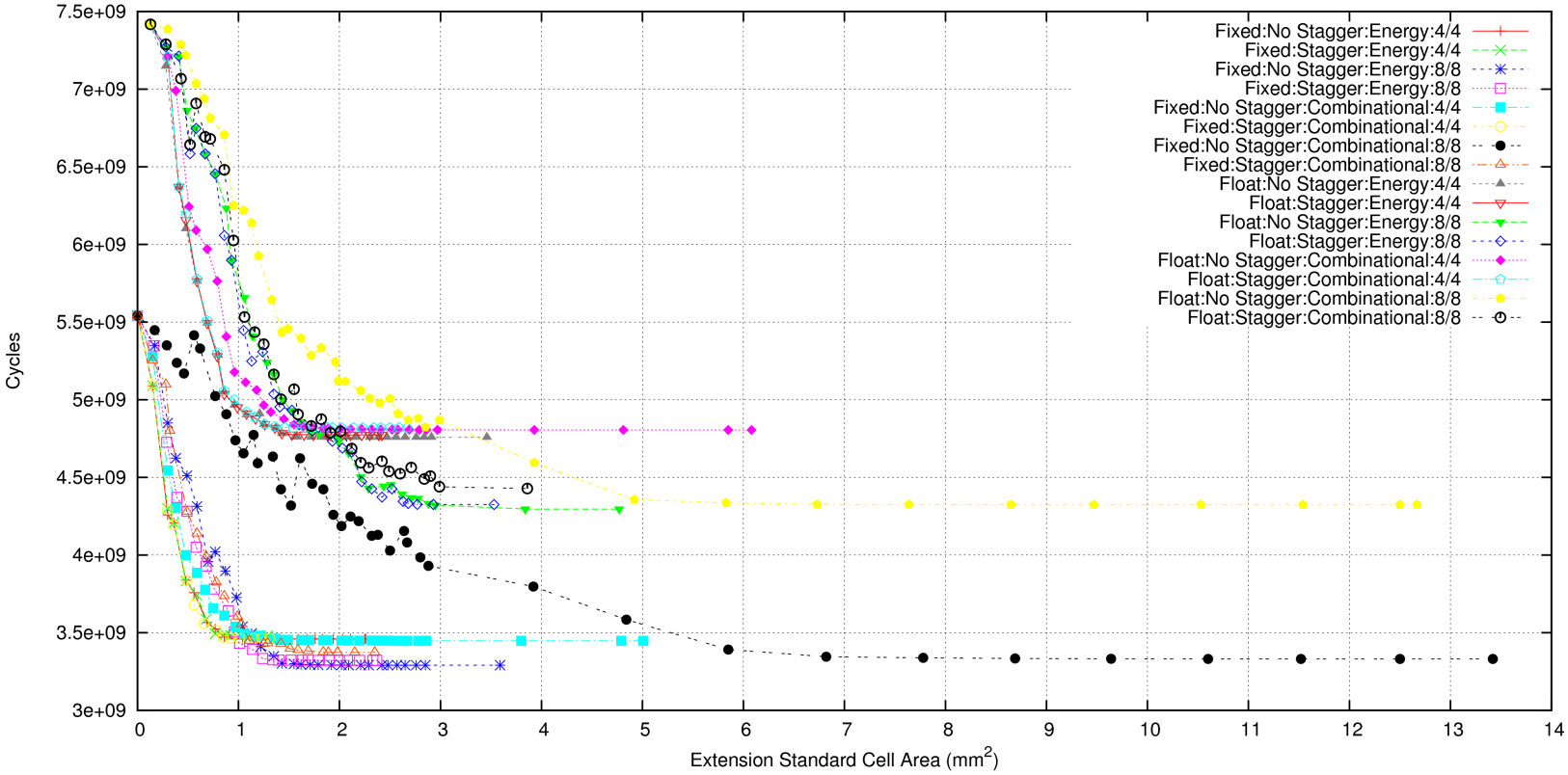


Fig. 6.23: FAAD: Extension Area versus Application Cycles; Regular Sampling of Area at 0.1mm^2 Target Interval from 0.2mm^2 to 3.0mm^2, and at 1.0mm^2 from 4.0mm^2 to maximum. The excess in area of the higher I/O when using the original heuristic is apparent in both floating- and fixed-point. The maximum size of extension logic for both is remarkably similar, implying that the extension of a floating-point application should be of similar area cost to an equivalent fixed-point application. The energy-aware heuristic produces a much smaller design in both formats, with floating-point being slightly bigger than the fixed-point but still largely the same. The effect of staggering on the energy-aware heuristic appears to reduce the result to roughly the same performance as the result of staggering with the combinational heuristic.

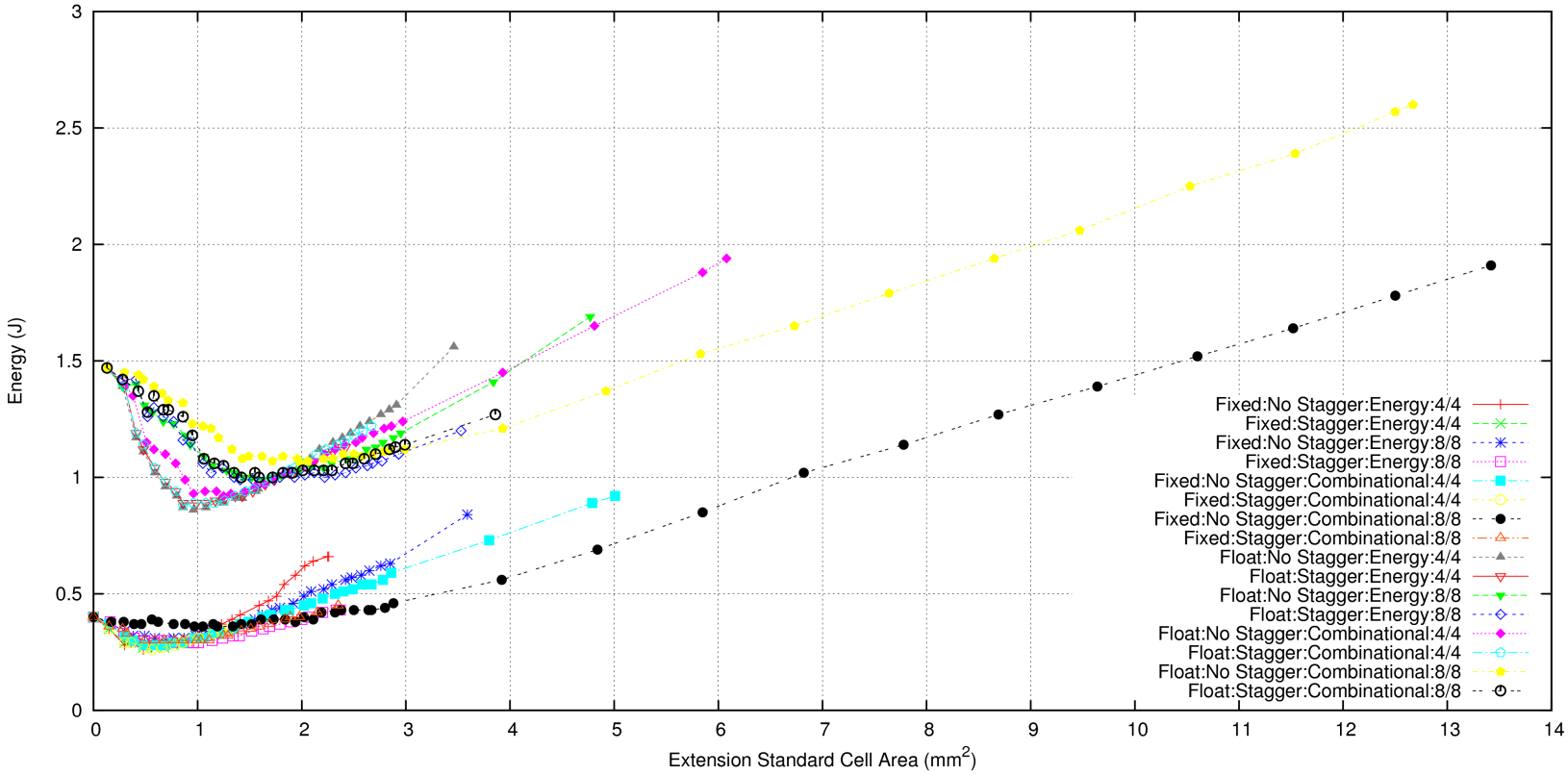


Fig. 6.24: FAAD: Extension Area versus Application Energy; Regular Sampling of Area at 0.1mm^2 Target Interval from 0.2mm^2 to 3.0mm^2, and at 1.0mm^2 from 4.0mm^2 to maximum. It appears with this graph alone that for a given area, the energy resulting from the energy-aware heuristic is greater than that for the combinational heuristic. Whilst this is true, the acceleration afforded by the designs at the same area in those two heuristics is drastically different, as seen in the previous figure 6.23.

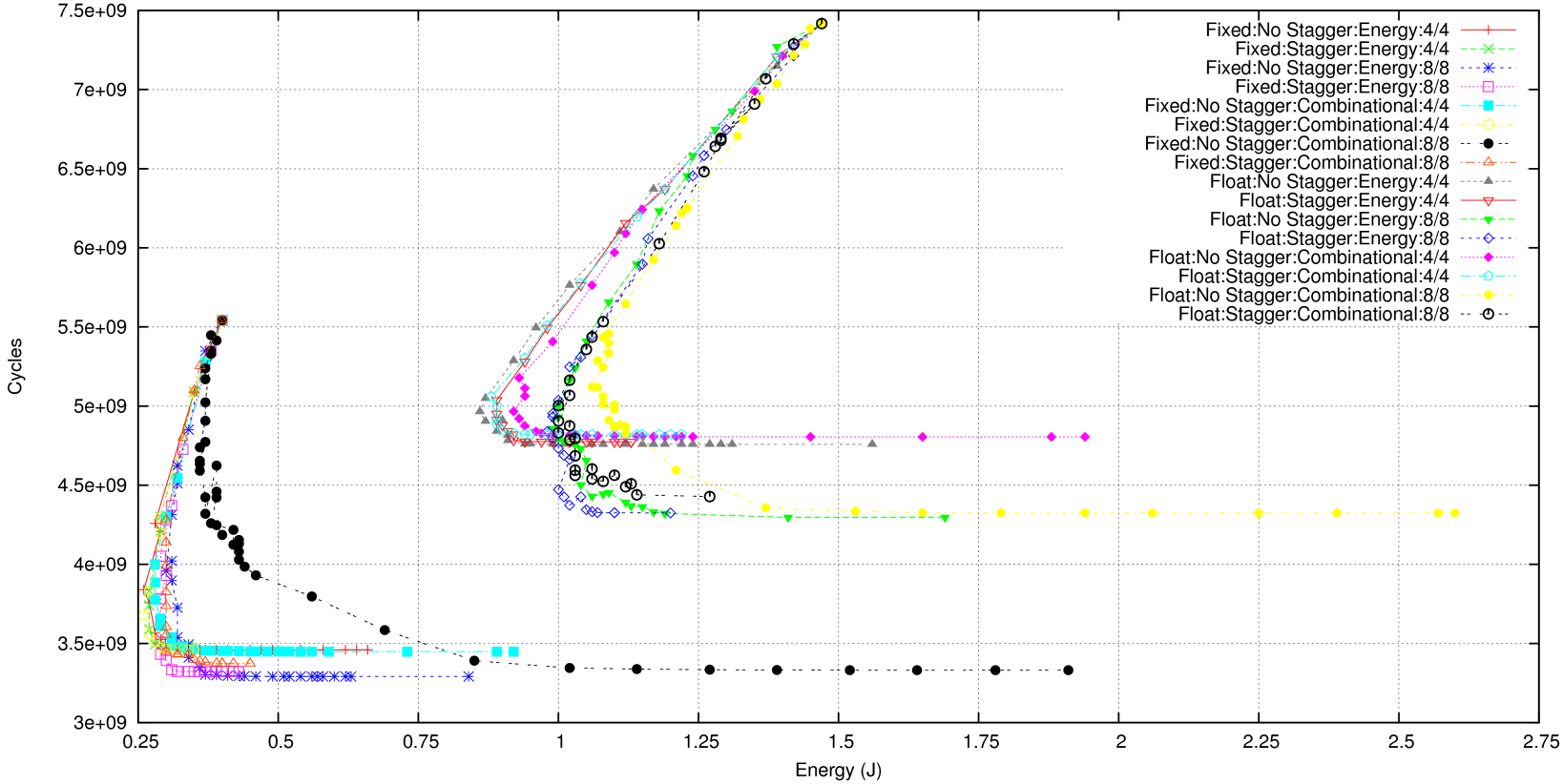


Fig. 6.25: FAAD: Application Cycles versus Energy; Regular Sampling of Area at 0.1mm^2 Target Interval from 0.2mm^2 to 3.0mm^2, and at 1.0mm^2 from 4.0mm^2 to maximum. Here we can see the cost of acceleration in terms of energy, removing the area question from the picture. This graph demonstrates the most extreme example of a trade-off knee seen in this thesis. There is a point in every series, some more pronounced than others, where acceleration goes from improving energy performance, to hurting it. The flattening out in the cycle axis demonstrates the point at which further ISEs cover a trivial execution, and are really not worthwhile including. This corresponds with the exponential decay of figure 6.23, and confirms that not only are these tails useless for acceleration, they also actively damage energy, power, and area performance.

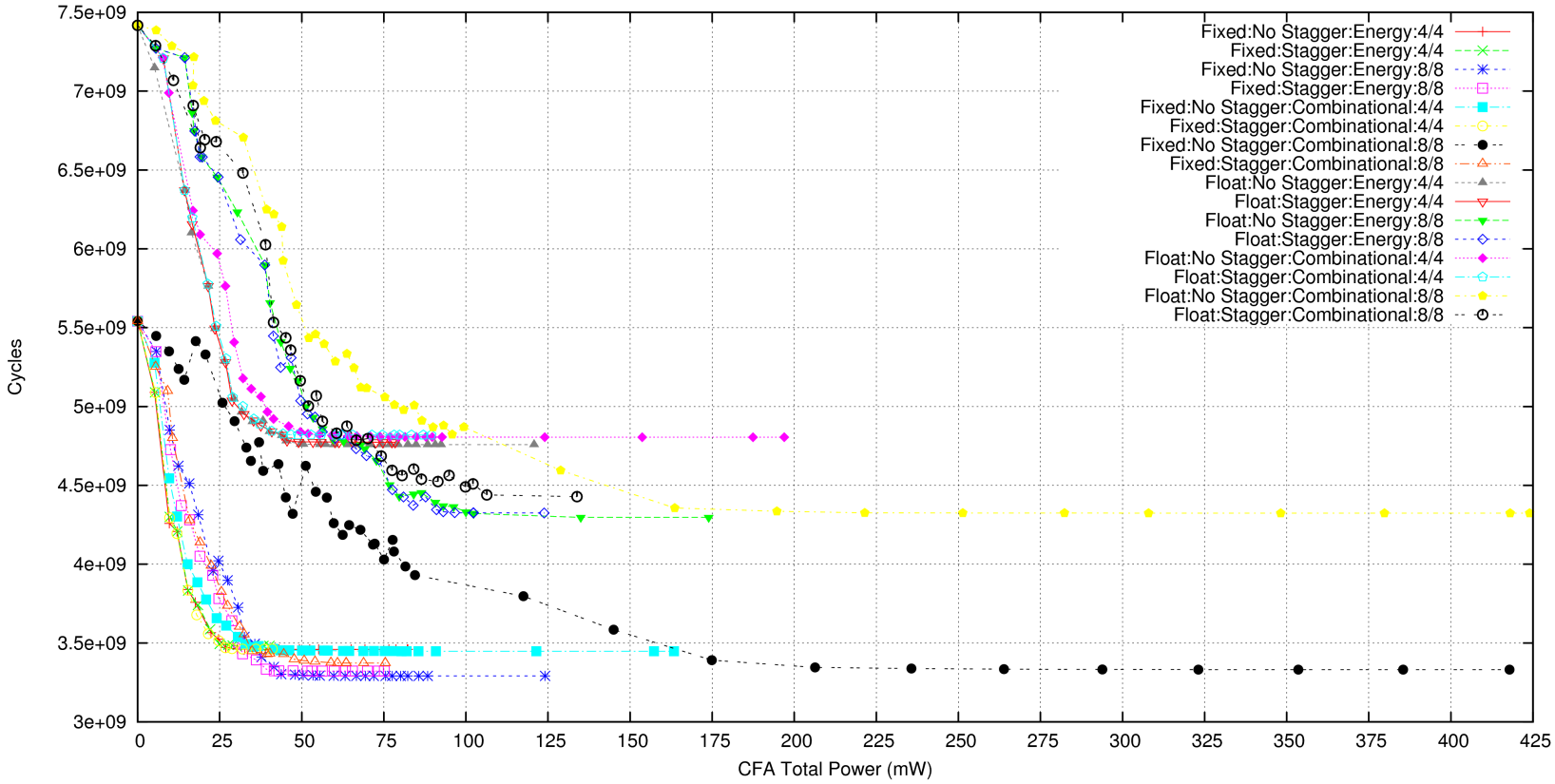


Fig. 6.26: FAAD: CFA Power versus Application Cycles; Regular Sampling of Area at 0.1mm^2 Target Interval from 0.2mm^2 to 3.0mm^2, and at 1.0mm^2 from 4.0mm^2 to maximum. The similarity of this graph in shape to figure 6.23 serves to confirm the earlier conclusion that area and power are near-linearly correlated with CFA designs.

6.3.4 Conclusions

Following this study of the differences between fixed and floating point number formats when utilising ISE, the following conclusions have been made:

- Floating point is the more expensive option in terms of hardware cost; both silicon area and battery size will need to be greater in the face of floating point.
- Floating point at higher area constraints for AISE can equal or surpass the baseline performance of fixed point in terms of run-time.
- No amount of ISEs as CFAs can make floating-point the same or better energy consumption as the baseline of fixed point.
- The more inefficient (i.e. top half of the area limit for 8/8 I/O) CFA designs cause fixed point to have a higher energy consumption than floating point, despite executing considerably faster.
- The energy heuristic is very effective at addressing energy concerns versus the original heuristic, but probably deserves a different name as it also impacts area, power, and runtime positively.
- Both heuristics have their greatest energy improvement prior to the area limit, and the knees representing the best trade-off ratio seen are even lower in terms of area.
- The combination of energy heuristic and staggering is roughly the same as the combination of original heuristic and staggering in terms of the result. It would seem that whatever special design considerations take place for the energy heuristic are annulled by staggering.
- Despite the relatively large comparative cost, floating point CFAs are not prohibitively expensive, and require extension logic of roughly the same size on average as that produced for fixed point for roughly the same relative acceleration.

6.4 Summary

This chapter has looked at two different areas of source code form which constitute decisions made at the design time of a system that will effect the efficacy of AISE and hence the whole design performed during HW/SW co-design.

A range of software transformations were explored in combination with AISE over a number of different benchmarks, all kernels. In particular, this resulted in two outcomes: The motivation of further exploration of this space to derive more specific trends, and the acknowledgement that AISE is particularly good at mopping up the inefficacies of transformation at the expense of area.

The two principal number formats for representing the binary point in a number format were explored in terms of their performance merit when considering an architecture designed to include ISE. It has been confirmed that AISE has roughly the same trends in both fixed- and floating-point. AISE is shown to bring the two formats closer together or further apart in performance depending on the care taken to select suitable constraints and heuristics.

7 CONCLUSIONS

"Each problem that I solved became a rule, which served afterwards to solve other problems."
– René Descartes

A round-up of the work produced in this thesis follows, making conclusions with regard to the contributions made in this work and the research leading from here.

7.1 Contributions

A number of contributions have been made in this work, with both engineering and scientific implications. The umbrella hypothesis for this work has been: "The efficacy of ISE can be increased by improving the microarchitecture, identification algorithm, and software form". Chapters 4, 5, and 6 have addressed the components of this hypothesis directly, with the following contributions made:

- *The CFA is demonstrated repeatedly to be a cost-effective design for ISE implementation.* The CFA includes explicit reconfigurability and pipelining, which distinguish it from the CCA. The full design and synthesis methodology is presented within section 4.2 and is carried forwards throughout this thesis.

- *A temporal partitioning algorithm called "staggering" is proposed and demonstrated on average to reduce the area of CFA implementation by 37% for only an 8% reduction in acceleration.* Staggering has considerable benefits for the unlimited area consumed by a CFA when implementing a set of ISEs. Larger applications gain more from the approach than smaller ones such as kernels so this approach is particularly useful for realistic applications. Sections 4.3 and 6.3 both investigate the efficacy of staggering.

- *A methodology for finding a good static weighting vector for ISEGEN is proposed and demonstrated. Up to 100% of merit is shown to be lost or gained through the choice of vector.* The work of section 5.2 leads to a solid result which later efforts use as a strong baseline for comparison. The static heuristic weighting vector promoted in the original ISEGEN publication [7] is not good in comparison to many of the vectors explored in section 5.2.

- *ISEGEN early-termination is introduced and shown to improve the runtime of the algorithm by up to 7.26x, and 5.82x on average.* Through careful analysis of the I/O constraint and its effects on the ISEGEN algorithm, early termination is added in section 5.3. Due to the polynomial big-O complexity of the section removed by early termination when triggered, the early termination is more effective with larger DFG as might be

encountered in real application optimised heavily to expose OLP. This is a particularly strong result as it better enables an iterative approach. Around six times more points on average can be considered in any one period of time, a great result for designers performing DSE.

- *An extension to the ISEGEN heuristic to account for pipelining is proposed and evaluated, increasing acceleration by up to an additional 1.5x.* The ISEGEN algorithm originally does not consider any inter-ISE parallelism, such as could be exploited by a microarchitecture like the CFA which contains multiple pipeline stages as opposed to one large multi-cycle stage. Two scheduling heuristics are considered in section 5.4, both performing As-Soon-As-Possible scheduling based on the availability of inputs to operations, but differing in the event that there are multiple choices on whether to choose the shortest (ASAP-SF) or longest (ASAP-LF) latency first. Evaluation demonstrates that the ASAP-SF approach is the strongest, that the new heuristics work better than even a properly calibrated incarnation of the original heuristic.

- *An energy-aware heuristic is added to ISEGEN, which reduces the energy used by a CFA implementation of a set of ISEs by an average of 1.6x, up to 3.6x.* The cost considerations of automatically adding ISEs to a microarchitecture are usually omitted in AISE identification. The energy-aware heuristic is originally presented in section 5.5 and conclusions are later confirmed in section 6.3. Improvements are made through a combination of design concerns: energy involves power which ultimately stems from area as shown in section 4.2, and the time spent in the various sections of the design (CFA and baseline core). Using this new identification heuristic for area, ISEs became leaner: meaning smaller ISE covering less area but with a greater individual acceleration factor. This result both encourages the further investigation of energy effects with ISE, and discourages the further pursuit of "bigger-is-better" AISE identification philosophies such as that of [18].

- *A methodology for combined exploration of source transformation and ISE is presented, and demonstrated to improve the acceleration of the result by an average of 35% versus ISE alone.* Source transformations have been in the repertoire of compiler engineers for decades as an approach to finding a better fit between the code that represents the software and the hardware on which it is meant to run. An exploration as to the effects of combining different combinations of source transformations and AISE is performed in section 6.2 and demonstrates that there is a critical link between AISE and the transformations applied. Transformations when applied wrongly can often lead to a situation where AISE "mops up" the inefficiency; adding hardware is far less attractive than getting the source transformations right in the first place. This heavily motivates the further study of combined compiler transformations and AISE, as there are interacting mecha-

nisms between the two which must be better understood.

- *Floating point is demonstrated to perform worse than fixed point, for all design concerns and applications studied here, regardless of ISEs employed.* Number format is a decision which has to be made in any project incorporating DSP, amongst others. The original engineering rule of thumb regarding the trade-offs between the two number formats have been confirmed in section 6.3 both with and without AISE. Using CFA-based ISE can reduce the relative gap between the two formats in terms of their runtime speed, but floating-point will always be a more expensive option in terms of hardware. Anyone considering this decision for a design should now be better informed to do so when using ISE in the face of the trends uncovered herein.

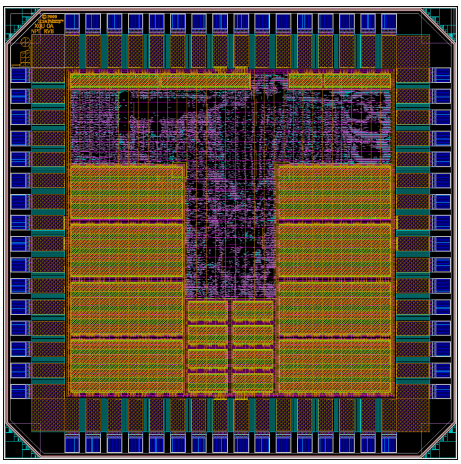


Fig. 7.1: The Castle revision of the EnCore microarchitecture, with CFA included targeted at the FAAD application. Yellow boxes along the top are CFA configuration memories. The large yellow box on the left is the instruction cache memory, on the right is the data cache memory. Smaller yellow boxes along the bottom are the tag memories for instruction and data caches. The logic in the bottom half of the "T" is the EnCore itself, whereas the logic along the top is the CFA.

7.2 EnCore and CFA integrated: Castle

As a proof of concept, the EnCore CPU has been extended with a single CFA targeted at accelerating the fixed-point version of the FAAD application. The integration has been dubbed the "Castle" revision of the EnCore microarchitecture. The design was submitted to fabrication in October 2009, and is expected to be complete in the second quarter of 2010, at which point further tests may be run on the performance of the architecture. The Castle microarchitecture utilises a 90nm standard-cell implementation, unlike the work of this thesis which is constructed in 130nm. Figure 7.1 illustrates the layout of the Castle chip. The CFA utilised in the Castle design is estimated to achieve around 1.4x acceleration over the baseline EnCore architecture via the techniques used in this thesis. Staggering as in section 4.3 was used to produce the CFA utilised in Castle, further demonstrating the validity of that approach.

Other specifications of the Castle microarchitecture include:

- Die Area: 1.875x1.875mm utilising eight metal layers.
- Instruction Cache Power Saving Scheme.
- Static Branch Prediction.
- 32KB 4-way data-cache and instruction-cache.
- 580MHz typical Fmax.
- 350MHz worst-case Fmax.
- 56uW/MHz including memories and CFA.

7.3 Further Work

The conclusions of the various experiments performed for this thesis have shed light on the potential progression from the current state-of-the-art to a more refined solution to the problems of ISE efficacy. The areas in which these refinements and alterations may be made are well defined, but overlap with regards to their application to the techniques outlined in this thesis:

- We have introduced the CFA as a microarchitecture for reconfigurable ISE implementation, but *most techniques or evaluations introduced herein could easily be applied to a number of other implementations*. This in itself forms one likely fruitful avenue of further work: to compare the efficacy of these techniques on other microarchitectural realisations (e.g. FPGA, combinational) of ISE. Techniques expected to be applicable are those of sections 4.3, 5.2, 5.3, 5.4, 5.5, 6.2, and 6.3.

- *Dynamic ISEGEN Weighting Vector Tuning*, which may be realised via any number of function-approximation techniques, is very likely to supply additional efficacy to the ISEGEN algorithm. Work performed for this thesis in sections 5.2, 5.4, 5.5, and 6.3 has demonstrated the need for a dynamic weighting vector to defeat the inefficiency introduced in finding a common weighting vector. The ability to set a good weighting vector on a per-DFG or even a per-ISEGEN-iteration basis rather than setting a single static vector for a whole application is likely to have a significant positive impact on the quality of the overall result. There are many statistical (e.g. regression) and structural (e.g. neural networks) approaches to this problem, in addition to iterative approaches wherein the *isegen* tool itself would adopt an iterative refinement approach to the weighting vector. A combination of these approaches is very likely to yield a better result than one in isolation, so the combination itself should be investigated via experimentation.

- *Further improving the CFA microarchitecture*, originally conceived as a DSP-accelerating reconfigurable unit, the CFA has several issues which could be overcome in order to make it either more effective in the DSP domain, or to be suitable for further domains. The following is a non-exhaustive list of the potential improvements and modifications which could and should be investigated:

    1. *Complex combinational ALU or FPGA inside the CFA to support the efficient inclusion of bitwise operators.* It has been well established that bitwise operators gain the greatest advantage from serial slack aggregation, allowing multiple operations in series to occur in a single clock. This is due to bitwise operations by their very definition having only a single gate-delay in a combinational circuit.
    2. *Direct Memory Access and a scratch-pad* to allow for streaming, removing pressure from the register file.

3. *Less general connectivity to reduce the expense of permutation layers.* Rather than allowing every operator in one echelon to pass its result to every operator in the next, use application-specific data to prune multiplexors.
4. *Internal explicit data-forwarding to counter the inefficiency of forwarding layers external to the CFA*. Such an approach could only cover CFA-CFA forwards.
5. *An improved implicit data-forwarding circuit external to the CFA*, to effectively replace the final layer of multiplexing in the CFA and the forwarding circuitry in the baseline core; this would be a more efficient option, but would require considerable design effort to produce.

- *Energy-aware heuristic inclusion in selection stage* as well as in the already implemented identification stage as in section 5.5, to allow for efficacious selection under constraints such as encoding or area.

- *Heuristic and constraint support for pipelined inputs*; contrary to the previous work of Pozzi and Ienne [23], pipelined inputs (i.e. multi-cycle input to an ISE) could be modelled internal to ISEGEN rather than iterating over them externally. This would greatly reduce the run-time of the algorithm with regard to a particular bandwidth, but is an alternative to the pipelining heuristic presented in this thesis as it addresses the same problem.

- *Less greedy selection algorithm*; currently *uarchgen* utilises a variant of the greedy box-packing algorithm to construct CFA. Devoting more computation to this problem could provide better designs.

- *Less greedy identification algorithm*, in particular search should have a wider context than a single basic block and ISE. For example, simultaneous identification of multiple ISEs within a single DFG. This approach would combine well with the new heuristics presented in sections 5.5 and 5.4, because these could make good use of more global scope.

- *Resolve issues of inaccuracy when using linear models for evaluation*, instead of a cycle-accurate fully-integrated simulation and synthesis. Issues with the linear models include:

  – Due to the level at which the DFG are represented (GIMPLE/SSA before any kind of machine-specific lowering), *the determination of Load/Store latency is estimated*. The problem is that these (and indeed all) DFG nodes have only a single per-type latency, and no distinction is made between reads from registers (zero latency), a register move (1 cycle latency) cache (varies depending on level), or main memory (may have multiple latencies depending on the address mapping). At present, the load/store nodes are not allowed to be covered because no CFA

scratch-pad or DMA exists yet. For the software portion of the execution time modeled, all loads and stores count as a single cycle of latency. In the face of function-level register allocation (as in GCC) it is very likely that a great majority of the loads will be already present in registers at the beginning of each basic block.

– *Actual simulation would be useful*, but was not possible due to the lack of a working post-ISE code generator.

– *The energy model used should be verified through actual integration* with the EnCore; in particular the energy model relies on forwarding to be better-implemented than it was with the Castle chip sent for fabrication in 2009, so this at least must be resolved before the model is entirely ratified.

BIBLIOGRAPHY

[1] C. Kozyrakis and D. Patterson. Vector vs superscalar and vliw architectures for embedded multimedia benchmarks, 2002.

[2] Jerzy Rozenblit and Klaus Buchenrieder. *Codesign - Computer-Aided Software/Hardware Engineering*. IEEE Press, New York, 1995.

[3] F. Theeuwen and E. Seelen. Power reduction through clock gating by symbolic manipulation, 1996.

[4] Kurt Keutzer, Sharad Malik, and Richard Newton. From asic to asip: The next design discontinuity. In *ICCD*, January 2002.

[5] Advanced Micro Devices. Amd64 architecture programmer's manual volume 6: 128-bit and 256-bit xop and fma4 instructions. 2009.

[6] Intel Corporation. Intel advanced vector extensions programming reference. 2008.

[7] Partha Biswas, Sundarshan Banerjee, Nikil D. Dutt, Laura Pozzi, and Paolo Ienne. Isegen: An iterative improvement-based ise generation technique for fast customization of processors. *IEEE Transactions on VLSI*, 14(7), 2006.

[8] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini. An integrated open framework for heterogeneous mpsoc design space exploration. In *Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2006.

[9] Gang Qu. What is the limit of energy saving by dynamic voltage scaling? In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 560–563, Piscataway, NJ, USA, 2001. IEEE Press.

[10] Nigel Topham, Oscar Almer, Freddie Qu, and Richard Bennett. The encore cpu: Arc compatible embedded processor – http://groups.inf.ed.ac.uk/pasta/hw_encore.html.

[11] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vintecelli. System level design: Orthogonalisation of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 19(12), 2000.

[12] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application specific processing on a general purpose core via transparent instruction set customization. In *Proc. 37th Intl. Symposium on Microarchitecture (MICRO)*, pages 30–40, 2004.

[13] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. 32nd Intl. Symposium on Computer Architecture (ISCA)*, pages 272–283, 2005.

[14] N. Clark, A. Hormati, S. Mahlke, and S. Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *Proc. 2006 Intl. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 147–157, 2006.

[15] S. Yehia, N. Clark, S. Mahlke, and K. Flautner. Exploring the design space of lut-based transparent accelerators. In *Proc. 2005 Intl. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 11–21, 2005.

[16] A. Hormati, N. Clark, and S. Mahlke. Exploiting narrow accelerators with data-centric subgraph mapping. In *Proc. 2007 International Symposium on Code Generation and Optimization (CGO)*, pages 147–157, 2007.

[17] *Stretch S6000 Architecture White Paper http://www.stretchinc.com/_files/s6ArchitectureOverview.pdf.*

[18] Ajay K. Verma, Philip Brisk, and Paolo Ienne. Rethinking custom ise identification: a new processor-agnostic method. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 125–134, 2007.

[19] Armita Peymandoust, Laura Pozzi, Paolo Ienne, and Giovanni De Micheli. Automatic instruction set extension and utilisation for embedded processors. *In Proceedings of the 14th International Conference on Application-specific Systems, Architectures and Processors, The Hague, The Netherlands.*, 2003.

[20] Kubilay Atasu, Gunhan Dundar, and Can Ozturan. An integer linear programming approach for identifying instruction-set extensions, 2005.

[21] Laura Pozzi, Kubilay Atasu, and Paolo Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1209–1229, 2006.

[22] Ajay K. Verma and Paolo Ienne. Towards the automatic exploration of arithmetic circuit architectures. In *In Proceedings of the 43rd Design Automation Conference, San Francisco, California*, 2006.

[23] Laura Pozzi and Paolo Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *In Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, San Francisco, Calif*, pages 2–10, 2005.

[24] Tony Givargis, Frank Vahid, and Jorg Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *ICCAD*, pages 25–30, 2001.

[25] Tensilica Inc. The xpres compiler: Triple-threat solution to code performance challenges. *Tensilica Inc Whitepaper*, 2005.

[26] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. Compiler-in-loop architecture exploration for efficient application specific embedded processor design, 2004.

[27] T. Glokler, A. Hoffmann, and H. Meyr. Methodical low-power asip design space exploration. *VLSI Signal Processing*, 33, 2003.

[28] *ACE CoSy Website - http://www.ace.nl/compiler/cosy.html.*

[29] *CoWare LISATek Datasheet - http://www.coware.com/PDF/products/LISATek.pdf.*

[30] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Laura Pozzi, and Paolo Ienne. Fast automated generation of high-quality instruction set extensions for processor customization. In *In Proceedings of the 3rd Workshop on Application Specific Processors, Stockholm*, 2004.

[31] Ieee standard verilog hardware description language. *IEEE Std 1364-2001*, 2001.

[32] Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, 26 2009.

[33] Philip Brisk, Adam Kaplan, and Majid Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 395–400, New York, NY, USA, 2004. ACM Press.

[34] M. Zuluaga and N. Topham. Resource sharing in custom instruction set extensions. In *Proceedings of the 6th IEEE Symposium on Application Specific Processors*, Jun. 2008.

[35] N. Moreano, E.and Cid de Souza Borin, and G. Araujo. Efficient datapath merging for partially reconfigurable architectures. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 24:969 – 980, Jul. 2005.

[36] Paolo Ienne and Ajay K. Verma. Arithmetic transformations to maximise the use of compressor trees. In *Proceedings of the IEEE International Workshop on Electronic Design, Test and Applications, Perth, Australia*, 2004.

[37] Paolo Bonzini and Laura Pozzi. Code transformation strategies for extensible embedded processors. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 242–252, New York, NY, USA, 2006. ACM Press.

[38] Yijian Wang and David Kaeli. Source level transformations to improve I/O data partitioning. In *International Workshop on Storage Network Architecture and Parallel I/Os*, 2003.

[39] E. Chung, L. Benini, and G. De Micheli. Energy efficient source code transformation based on value profiling. In *International Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, USA, 2000.

[40] C. Kulkarni, F. Catthoor, and H. De Man. Code transformations for low power caching in embedded multimedia processors. In *12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, pages 292–297, 1998.

[41] B.D. Winters and A.J. Hu. Source-level transformations for improved formal verification. In *IEEE Internation Conference on Computer Design*, 2000.

[42] Björn Franke and Michael O'Boyle. Combining program recovery, auto-parallelisation and locality analysis for C programs on multi-processor embedded systems. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, New Orleans, September/October 2003.

[43] Heiko Falk and Peter Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[44] Victor De La Luz and Mahmut Kandemir. Array regrouping and its use in compiling data-intensive embedded applications. *IEEE Transactions on Computers*, 53(1):1–19, 2004.

[45] Markus Schordan and Daniel J. Quinlan. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference*, 2003.

[46] Alexandre Borghi, Valentin David, and Akim Demaille. C-transformers - a framework to write C program transformations. *ACM Crossroads*, 2004.

[47] Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin. Probabilistic source-level optimisation of embedded programs. In *2005 Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'05)*, 2005.

[48] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Michael F.P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K.I. Williams. Using machine learning to focus iterative optimization. In *4th Annual International Symposium on Code Generation and Optimization (CGO)*, 2006.

[49] Clifford E. Cummings. New verilog-2001 techniques for creating parameterized models (or down with 'define and death of a defparam!). In *HDLCON 2002: Proceedings of the 2002 Conference on Hardware Description Languages*.

[50] Ieee std 1666 - 2005 ieee standard systemc language reference manual. *IEEE Std 1666-2005*, 2006.

[51] Stuart Sutherland. Integrating systemc models with verilog and systemverilog models using the systemverilog direct programming interface. In *SNUG Europe 2004: Proceedings of the 2004 Synopsys Users Group.*, 2004.

[52] Sami Boukhechem and El-Bay Bourennane. Tlm platform based on systemc for starsoc design space exploration. *Adaptive Hardware and Systems, NASA/ESA Conference on*, 0:354–361, 2008.

[53] J. Groschdl. Instruction set extension for long integer modulo arithmetic on risc-based smart cards. *Computer Architecture and High Performance Computing, Symposium on*, 0:0013, 2002.

[54] K. Koutsomyti, V.A. Chouliaras, S.R. Parr, J.L. Nunez-Yanez, and S. Datta. Accelerating speech coding standards through systemc- synthesized simd and scalar accelerators. In *Consumer Electronics, 2006. ICCE '06. 2006 Digest of Technical Papers. International Conference on*, pages 279–280, Jan. 2006.

[55] Jeffrey P. Hammes, Bruce A. Draper, and A.P. Willem Bhm. Sassy: A language and optimizing compiler for image processing on reconfigurable computing systems. In *in International Conference on Vision Systems. 1999. Las Palmas de Gran Canaria*, pages 522–537. Springer, 1999.

[56] W. Bohm Y, J. Hammes, B. Draper, M. Chawathe, C. Ross, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems, 2002.

[57] Agility Design Solutions Inc. *Handel-C Language Reference Manual*.

[58] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[59] Ian Page. Hardware-software co-synthesis research at oxford, 1996.

[60] Jeremy Hinton and Alan Pinder. *Transputer hardware and system design*. Prentice-Hall, Inc., 1993.

[61] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized generation of data-path from c codes for fpgas. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 112–117, 2005.

[62] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Weui Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12), 1994.

[63] Benjamin Arai and Conley Read. Dwt design exploration via roccc. *Technical Report*, 2005.

[64] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466, January 2003.

[65] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Yi Lu, and S. Vassiliadis. Dwarv: Delftworkbench automated reconfigurable vhdl generator. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 697–701, August 2007.

[66] Monika Lam et al. An overview of the suif2 compiler infrastructure – http://suif.stanford.edu/suif/suif2. *Computer Systems Laboratory, Stanford University*, 2000.

[67] Arcilio J. Virginia, Yana D. Yankova, and Koen L.M. Bertels. An empirical comparison of ansi-c to vhdl compilers: Spark, roccc and dwarv.

[68] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa - machine description language and generic machine model for hw/sw co-design. In *in Proceedings of the IEEE Workshop on VLSI Signal Processing*, pages 127–136, 1996.

[69] *CoWare Website - http://www.coware.com*.

[70] JJ. Ceng, W. Sheng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. Modeling instruction semantics in adl processor descriptions for c compiler retargeting. In *SAMOS*, pages 463–473, 2004.

[71] A. Chattopadhyay, W. Ahmed, K. Karuri, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr. Design space exploration of partially re-configurable embedded processors.

In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 319–324, 2007.

[72] O. Schliebusch, A. Chattopadhyay, M. Steinert, G. Braun, A. Nohl, R. Leupers, G. Ascheid, and H. Meyr. Rtl processor synthesis for architecture exploration and implementation. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE) - Designers Forum*, 2004.

[73] H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, J. Zeng, K. Karuri, R. Leupers, G. Ascheid, and H. Meyr. Asip architecture exploration for efficient ipsec encryption: A case study. In *Proceedings of the 8th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, 2004.

[74] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors with Lisa*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[75] O. Schliebusch, A. Chattopadhyay, E.M. Witte, D. Kammler, G. Ascheid, R. Leupers, and H. Meyr. Optimization techniques for adl-driven rtl processor synthesis. In *IEEE Workshop on Rapid System Prototyping (RSP)*, 2005.

[76] *The Modern Embedded Systems: Compilers, Architectures, and Languages Project - http://www.gigascale.org/mescal*.

[77] Matthias Gries and Kurt Keutzer. *Building ASIPs: The Mescal Methodology*. Springer Publishers, 2005.

[78] Yujia Jin, Nadathur Rajagopalan Satish, Kaushik Ravindran, and Kurt Keutzer. An automated exploration framework for fpga-based soft multiprocessor systems. In *Proceedings of the 2005 International Conference on Hardware/Software Codesign and System Synthesis (CODES-05)*, pages pp 273–278, September 2005.

[79] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 76–103, 2008.

[80] David F. Bacon. Kava: a java dialect with a uniform object model for lightweight classes. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 68–77, 2001.

[81] *Trimaran: An Infrastructure For Research In Instruction-Level Parallelism - http://www.trimaran.org*.

[82] Bhuvan Middha, Anup Gangwar, Anshul Kumar, M. Balakrishnan, and Paolo Ienne. A trimaran based framework for exploring the design space of vliw asips with coarse grain functional units. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 2–7, New York, NY, USA, 2002. ACM Press.

[83] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 49–56, 2000.

[84] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus: efficient realization of streaming applications on fpgas. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 41–50, 2008.

[85] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[86] Peter Grun, Ashok Halambi, Asheesh Khare, Vijay Ganesh, Nikil Dutt, and Alexandru Nicolau. Expression: An adl for system level design exploration. Technical report, 1998.

[87] Joseph A. Fisher, Paolo Faraboschi, and Giuseppe Desoli. Custom-fit processors: letting applications define architectures. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 324–335, 1996.

[88] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. A dag based design approach for reconfigurable vliw processor, 1999.

[89] Laura Pozzi, Miljan Vuletic, and Paolo Ienne. Automatic topology-based identification of instruction-set extensions for embedded processors. In *In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, Paris*, 2002.

[90] Carlo Galuzzi, Koen Bertels, and Stamatis Vassiliadis. A linear complexity algorithm for the generation of multiple input single output instructions of variable size. In *SAMOS*, pages 283–293, 2007.

[91] Carlo Galuzzi, Dimitris Theodoropoulos, Roel Meeuws, and Koen Bertels. Automatic instruction-set extensions with the linear complexity spiral search. *Reconfigurable Computing and FPGAs, International Conference on*, 0:31–36, 2008.

[92] C. Galuzzi, K. Bertels, and S. Vassiliadis. The spiral search: A linear complexity algorithm for the generation of convex mimo instruction-set extensions. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, Dec. 2007.

[93] R.V.Bennett. Automated asip extension generation through compiler-in-loop architecture exploration, 2006.

[94] Brendan McKay. Practical graph isomorphism. In *Congressus Numerantium*, volume 30, pages 45–87, 1981.

[95] Paolo Bonzini and Laura Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *DATE '07: Design Automation and Test in Europe*, 2007.

[96] Tensilica Inc. *XTensa Customisable Processors Overview - http://www.tensilica.com/products/xtensa-customizable.htm*, November 2009.

[97] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137–147, 2003.

[98] Paolo Ienne, Laura Pozzi, and Miljan Vuletic. On the limits of automatic processor specialisation by mapping dataflow sections on ad-hoc functional units. *Technical Report 01/376, Swiss Federal Institute of Technology Lausanne (EPFL)*, 2001.

[99] Jong eun Lee, Kiyoung Choi, and Nikil D.Dutt. Automatic instruction set design through efficient instruction encoding for application-specific processors. Technical Report 02–23, CECS, University of California, Irvine, 2003.

[100] Diviya Jain, Anshul Kumar, Laura Pozzi, and Paolo Ienne. Automatically customising vliw architectures with coarse grained application-specific functional units. In *In Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems, Amsterdam*, 2004.

[101] Michael Gschwind. Instruction set selection for asip design. In *CODES '99: Proceedings of the seventh international workshop on Hardware/software codesign*, pages 7–11, New York, NY, USA, 1999. ACM Press.

[102] Nathan T. Clark and Hongtao Zhong. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Trans. Comput.*, 54(10):1258–1270, 2005. Member-Scott A. Mahlke.

[103] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customization. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 129, 2003.

[104] Samik Das, P. P. Chakrabarti, and Pallab Dasgupta. Instruction-set-extension exploration using decomposable heuristic search. In *VLSID '06: Proceedings of the 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design*, pages 293–298, 2006.

[105] Uwe Kastens, Dinh Khoi Le, Adrian Slowik, and Michael Thies. Feedback driven instruction-set extension. *SIGPLAN Not.*, 39(7):126–135, 2004.

[106] Carlos Galuzzi and Koen Bertels. The instruction set extension problem: A survey. In *International Workshop on Applied Reconfigurable Computing (ARC)*.

[107] *Xilinx Website - http://www.xilinx.com*.

[108] Vaughn Betz and Jonathan Rose. Using architectural "families" to increase fpga speed and density. *Field-Programmable Gate Arrays, International ACM Symposium on*, 0:10–16, 1995.

[109] Jianshe He and Jonathan Rose. Advantages of heterogeneous logic block architectures for fpgas. In *Custom Integrated Circuits Conference*, pages 7–4, 1993.

[110] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Paolo Ienne, and Laura Pozzi. Performance and energy benefits of instruction set extensions in an fpga soft core. In *In Proceedings of the 19th International Conference on VLSI Design, Hyderabad, India*, pages 651–656, 2006.

[111] Georgi Kuzmanov, Georgi Gaydadjiev, and Stamatis Vassiliadis. The molen processor prototype. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:296–299, 2004.

[112] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E.M. Panainte. The molen polymorphic processor. *Computers, IEEE Transactions on*, 53(11):1363–1375, Nov. 2004.

[113] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *Computer*, 33(4):62–69, Apr 2000.

[114] M.B. Gokhale and J.M. Stone. Napa c: compiling for a hybrid risc/fpga architecture. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 126–135, Apr 1998.

[115] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, and R. Laufer. Piperench: a coprocessor for streaming multimedia acceleration. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 28–39, 1999.

[116] Robert G. Dimond, Oskar Mencer, and Wayne Luk. Custard - a customisable threaded fpga soft processor and tools. In *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL), Tampere, Finland, August 24-26, 2005*, pages 1–6, 2005.

[117] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. *Field-Programmable Logic and Applications LNCS 2778*, pages 61–70, 2003.

[118] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M.C. Filho. Morphosys: a reconfigurable architecture for multimedia applications. In *Integrated Circuit Design, 1998. Proceedings. XI Brazilian Symposium on*, pages 134–139, 1998.

[119] Gerard J. M. Smit, Pascal T. Wolkotte Andre B. J. Kokkele and, Philip K. F. Holzenspies, Marcel D. van de Burgwal, , and Paul M. Heysters. The Chameleon architecture for streaming DSP applications. *EURASIP Journal on Embedded Systems*, 2007.

[120] Frederick C. Furtek, Eugene Hogenauer, and James Scheuermann. Interconnecting heterogeneous nodes in an adaptive computing machine. In *FPL*, pages 125–134, 2004.

[121] Paul Master. A quick look into quicksilver's acm architecture – http://www.qstech.com/pdfs/a_look_into_quicksilver.pdf. *EETimes*, 2002.

[122] Tensilica Inc. *XTensa LX3 Product Brief - http://www.tensilica.com/products/xtensa-customizable/xtensa-lx.htm*, November 2009.

[123] Tensilica Inc. *XTensa 8 Processor for SoC Design - http://www.tensilica.com/products/xtensa-customizable/xtensa.htm*, November 2009.

[124] Michael Wehner, Leonid Oliker, and John Shalf. Towards ultra-high resolution models of climate and weather. *International Journal of High Performance Computing Applications*, 22(2):149–165, 2008.

[125] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. Pact xpp—a self-reconfigurable data processing architecture. *Journal of Supercomputing*, 26(2):167–184, 2003.

[126] Jurgen Becker and Alexander Thomas. Scalable processor instruction set extension. *IEEE Des. Test*, 22(2):136–148, 2005.

[127] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22:25–35, March 2002.

[128] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Mapping applications onto reconfigurable kress arrays. In *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pages 385–390, London, UK, 1999. Springer-Verlag.

[129] Nagaraju Pothineni, Anshul Kumar, and Kolin Paul. Application specific datapath extension with distributed i/o functional units. In *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, pages 551–558, Washington, DC, USA, 2007. IEEE Computer Society.

[130] A. Chattopadhyay, D. Kammler, E.M. Witte, O. Schliebusch, H. Ishebabi, G. Geukes, R. Leupers, G. Ascheid, and H. Meyr. Automatic low power optimizations during adl-driven asip design. In *IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2006.

[131] Ramkumar Jayaseelan, Haibin Liu, and Tulika Mitra. Exploiting forwarding to improve data bandwidth of instruction-set extensions. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 43–48, 2006.

[132] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Paolo Ienne, and Laura Pozzi. Performance and energy benefits of instruction set extensions in an fpga soft core. In *VLSID '06: Proceedings of the 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design*, pages 651–656, 2006.

[133] Paolo Bonzini, Dilek Harmanci, and Laura Pozzi. A study of energy saving in customizable processors. In *SAMOS*, pages 304–312, 2007.

[134] Yunsi Fei, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Energy estimation for extensible processors. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10682, Washington, DC, USA, 2003. IEEE Computer Society.

[135] Christian Laetsch. A multi-layer intermediate representation for ASIP design. Master's thesis, EPFL, Lausanne, Switzerland, September 2003.

[136] Corinna G. Lee. *UTDSP Benchmarks - http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html*, 1998.

[137] Guohua Jin, Xuejun Yang, and Fujie Chen. Loop staggering and compacting: Restructuring techniques for thrashing problem. In *ICPP (1)*, pages 678–679, 1991.

[138] Joao M.P. Cardoso. On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures. *IEEE Transactions on Computers*, 52(10):1362–1375, 2003.

[139] Christophe Bobda. Temporal partitioning and sequencing of dataflow graphs on reconfigurable systems. In *DIPES '02: Proceedings of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems*, pages 185–194, 2002.

[140] Farhad Mehdipour, Hamid Noori, Morteza Saheb Zamani, Kazuaki Murakami, Koji Inoue, and Mehdi Sedighi. Custom instruction generation using temporal partitioning techniques for a reconfigurable functional unit. 2006.

[141] Hamid Noori, Farhad Mehdipour, Kazuaki Murakami, Koji Inoue, and Morteza Saheb Zamani. An architecture framework for an adaptive extensible processor. *J. Supercomput.*, 45(3):313–340, 2008.

[142] J. E. Thornton. Parallel operation in the control data 6600. volume 26, 1964.

[143] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, 1967.

[144] Marcela Zuluaga, Theo Kluter, Philip Brisk, Nigel Topham, and Paolo Ienne. Introducing control-flow inclusion to support pipelining in custom instruction set extensions.

[145] K. Atasu, R.G. Dimond, O. Mencer, and W. Luk. Optimizing instruction-set extensible processors under data bandwidth constraints. In *Design, Automation and Test in Europe: DATE07*, pages 1–6, 2007.

[146] *SNU-RT Real-Time Benchmarks - http://archi.snu.ac.kr/realtime/benchmark/.*

[147] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, 2008.

[148] Analog Devices Inc. Tigersharc embedded processor adsp-ts101s – http://www.analog.com/static/imported-files/data_sheets/adsp-ts101s.pdf. 2010.

[149] V. Zivojnovic, H. Schraut, M. Willems, and R. Schoenen. Dsps, gpps, and multimedia applications - an evaluation using dspstone. In *In Proceedings of the International Conference on Signal Processing Applications and Technology*, 1995.