THE UNIVERSITY
of EDINBURGH

Mapping Parallelism to Heterogeneous Processors

Kiran Chandramohan

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2016

Abstract

Most embedded devices are based on heterogeneous Multiprocessor System on Chips (MPSoCs). These contain a variety of processors like CPUs, micro-controllers, DSPs, GPUs and specialised accelerators. The heterogeneity of these systems helps in achieving good performance and energy efficiency but makes programming inherently difficult. There is no single programming language or runtime to program such platforms.

This thesis makes three contributions to these problems. First, it presents a framework that allows code in Single Program Multiple Data (SPMD) form to be mapped to a heterogeneous platform. The mapping space is explored, and it is shown that the best mapping depends on the metric used.

Next, a compiler framework is presented which bridges the gap between the high -level programming model of OpenMP and the heterogeneous resources of MPSoCs. It takes OpenMP programs and generates code which runs on all processors. It delivers programming ease while exploiting heterogeneous resources.

Finally, a compiler-based approach to runtime power management for heterogeneous cores is presented. Given an externally provided budget, the approach generates heterogeneous, partitioned code that attempts to give the best performance within that budget.

Lay Summary

The world is witnessing an ever increasing demand for computational resources. The industry has responded by building huge centralised data-centers and increasing computational capabilities of embedded devices. To improve performance, these platforms also adopted multicore architectures where there are many processors of the same kind. Power is often a limiting factor in both these cases. Computer architects have long realised this and have added specialised processors for a particular kind of task. The presence of specialised processors improves the power efficiency but makes it very difficult to create software for these platforms. It is not clear how to partition work among all the processors, unlike homogeneous multicores, where work could be divided uniformly. Also, programming languages and libraries previously used for programming homogeneous multicore processors become ineffective in the presence of heterogeneous specialised processors. Finally, due to thermal effects, all the processors cannot be powered on simultaneously.

This thesis looks at how OpenMP, a programming language extension typically used for programming homogeneous multicore processors, can be used for heterogeneous processors and examines how work can be partitioned among the heterogeneous processors. It also presents methods for achieving good performance in the presence of power budgets.

Acknowledgements

First, I would like to acknowledge the guidance of my PhD supervisor Prof. Michael O'Boyle. He has given me excellent advice on which problems to solve and what areas to concentrate on. He was also a great source of ideas, many of which helped form the research described in this thesis. Last, but not the least, is the air of positivity that he brings to the table, which has helped me tide over many dead-ends and disappointments.

Next, I would like to thank Robert Clark of Redhat and Suman Anna of Texas Instruments. I cannot imagine setting up the framework without the help of these two people. I would also like to thank Prof. Bjorn Franke, Oscar Almer, Chris Thompson, Volker Seeker, Erik Tomusk and Jos van Eijndhoven for sharing their knowledge about Heterogeneous Systems and Power Measurement. Thanks to Prof. Vijay Nagarajan for being a part of my committee, and providing constructive feedback on my work. Thanks to Dr. Zheng Wang for helpful discussions and letting me use his OpenMP to OpenCL translator. Thanks also to Chris Margiolas, Alberto Magni, Vasilios Porpodas, Wen Yuan, and Stan Manilov for discussions about compilers, runtime and GPUs. Thanks to Andrew McPherson for many discussions about memory consistency, compilers and UK/Scotland.

I would like to thank Bharghava Rajaram and Murali Emani for all the help during the time I settled in and also for very useful technical discussions. Thanks also to Karthik, Govind, Konstantina Mitropolou, Praveen Tamanna, Ursula Chalita, George, Luna Ferrari and Rui Li for many discussions.

Finally, I am grateful to my family for being understanding and supportive for the entire duration of the PhD degree.

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 material presented in this thesis has been published in the following papers.

- Kiran Chandramohan, Michael O'Boyle, Zheng Wang. "Power Constrained Heterogeneous Code Generation". Under submission, in Chapter 6.
- Kiran Chandramohan, Michael O'Boyle. "A compiler framework for automatically mapping data parallel programs to heterogeneous MPSoCs", In International Conference on Compilers, Architecture and Synthesis for Embedded Systems(CASES), October 2014, in Chapter 5.
- Kiran Chandramohan, Michael O'Boyle, "Partitioning Data-parallel Programs for Heterogeneous MPSoCs: Time and Energy Design Space Exploration", In SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems(LCTES), June 2014, in Chapter 4.

(*Kiran Chandramohan*)

Contents

1 Introduction **1**

1.1 Problems 4

1.1.1 Partitioning 4

1.1.2 Compiler Framework 5

1.1.3 Power Budget 5

1.2 Contributions 6

1.3 Thesis Organisation 7

1.4 Summary 8

2 Background **9**

2.1 Multiprocessors 9

2.2 Heterogeneous Systems 10

2.2.1 Single ISA 10

2.2.2 Multiple ISA 10

2.3 Parallel Programming Models 13

2.3.1 Data Parallel Model 13

2.4 Parallel Programming Frameworks 14

2.4.1 POSIX threads 14

2.4.2 OpenMP 15

2.4.3 OpenCL 16

2.4.4 Comparison 18

2.5 Compiler 18

2.5.1 Uniprocessors 18

2.5.2 Homogeneous Multiprocessors 18

2.5.3 Heterogeneous Multiprocessors 19

2.5.4 Clang/LLVM 19

2.6 Measurements 19

2.6.1 Runtime 20

2.6.2 Energy . . . 20
2.7 Benchmarks . . . 21
2.8 Summary . . . 22

3 Related Work **23**
3.1 Heterogeneity . . . 23
3.1.1 Single ISA heterogeneity . . . 24
3.1.2 Multiple ISA heterogeneity . . . 24
3.1.3 Prototyping Heterogeneity . . . 25
3.2 Programming Languages . . . 26
3.2.1 Compiler . . . 27
3.2.2 OpenMP to SPMD . . . 27
3.3 Mapping Parallelism . . . 27
3.3.1 Partitioning . . . 27
3.3.2 Scheduling Tasks . . . 28
3.4 Memory Management . . . 29
3.4.1 Multiple Address Spaces . . . 29
3.4.2 Cache Coherence . . . 30
3.5 Power/Energy . . . 30
3.5.1 Power . . . 30
3.5.2 Energy Efficiency . . . 32
3.6 Summary . . . 33

4 Partitioning Data-parallel Programs for Heterogeneous MPSoCs **35**
4.1 Introduction . . . 35
4.2 Motivation . . . 38
4.2.1 Time . . . 38
4.2.2 Energy . . . 39
4.3 Programming Model . . . 40
4.3.1 Data Parallelism and SPMD . . . 40
4.3.2 The OMAP model . . . 42
4.3.3 Mapping SPMD programs to OMAP . . . 43
4.3.4 Code Generation . . . 44
4.4 Partitioning Policies . . . 45
4.5 Metrics . . . 48
4.5.1 Energy . . . 48

4.5.2 Two energy measures . . . 49
4.5.3 Runtime and EDP . . . 50
4.6 Experimental Setup . . . 50
4.6.1 Benchmarks . . . 50
4.7 Matrix Multiplication Case Study . . . 50
4.7.1 Individual Processor . . . 51
4.7.2 Partitioning . . . 53
4.8 Partitioning Policy Results . . . 54
4.8.1 Runtime . . . 55
4.8.2 Energy . . . 56
4.8.3 Analysis of Results . . . 57
4.9 Summary . . . 58

5 Compiler Framework **59**
5.1 Introduction . . . 59
5.2 Motivation . . . 61
5.3 Target Architecture and Runtime . . . 63
5.4 Programming models . . . 64
5.4.1 OpenMP . . . 64
5.4.2 Syslink Model . . . 65
5.4.3 SPMD . . . 66
5.4.4 Example . . . 66
5.5 SPMD as a bridging model . . . 66
5.5.1 Mapping OpenMP to SPMD . . . 67
5.5.2 Mapping SPMD to MPSoC . . . 67
5.6 Compiler Algorithm . . . 68
5.6.1 Data partition . . . 68
5.6.2 Data bounds per processor . . . 69
5.6.3 Computation Partitioning . . . 69
5.6.4 Cache Flush . . . 70
5.6.5 Code Generation . . . 70
5.7 Shared Memory on an MPSoC . . . 71
5.8 Compiler Implementation . . . 72
5.8.1 Source to Source Translation . . . 72
5.9 Setup . . . 75

5.9.1 Benchmarks . . . 75
5.9.2 Measurement . . . 75
5.10 Experiments . . . 76
5.10.1 Syslink and OpenMP Model . . . 76
5.10.2 Our SPMD approach . . . 76
5.10.3 SPMD Barrier optimisations . . . 77
5.10.4 Comparison Summary . . . 78
5.10.5 Energy and EDD . . . 78
5.10.6 Limit Study . . . 80
5.11 Summary . . . 80

6 Power Constrained Code Generation **83**
6.1 Introduction . . . 83
6.2 Motivation . . . 85
6.3 Our Approach . . . 86
6.3.1 Compile-time . . . 87
6.3.2 Runtime . . . 87
6.4 Code generation . . . 88
6.4.1 OpenMP . . . 88
6.4.2 Hybrid SPMD . . . 89
6.4.3 Memory Management . . . 90
6.5 Power Constrained Model . . . 91
6.5.1 Objective . . . 91
6.5.2 Selecting a configuration . . . 92
6.5.3 Policies . . . 92
6.6 Setup . . . 95
6.6.1 Benchmarks . . . 95
6.6.2 Platforms . . . 95
6.6.3 Policies evaluated . . . 96
6.6.4 Methodology . . . 97
6.7 Results . . . 97
6.7.1 Comparison against *DVFS* and Oracle . . . 97
6.7.2 Comparison to Naive and Core selection . . . 99
6.7.3 Speedup Summary . . . 103
6.7.4 Over/Under Shooting . . . 105

6.8 Summary . . . 106

7 Conclusion **109**

7.1 Contributions . . . 109

7.1.1 Partitioning . . . 109

7.1.2 Compiler Framework . . . 110

7.1.3 Power Budget . . . 110

7.2 Critical Analysis . . . 110

7.2.1 Common Issues . . . 111

7.2.2 Partitioning . . . 111

7.2.3 Compiler Framework . . . 112

7.2.4 Power Budget . . . 112

7.3 Future work . . . 112

7.3.1 Scale . . . 112

7.3.2 Partitioning . . . 113

7.3.3 Oracle Computation . . . 114

7.3.4 Multiple Programs . . . 114

7.3.5 Mobile Devices . . . 114

7.4 Summary . . . 115

A Experiments with Snapdragon Platform **117**

A.1 Snapdragon . . . 117

A.1.1 Hardware . . . 117

A.1.2 Runtime . . . 117

A.1.3 Programming Model . . . 118

A.2 Experiments . . . 118

A.3 Summary . . . 119

B Bug fix in Linux Kernel **121**

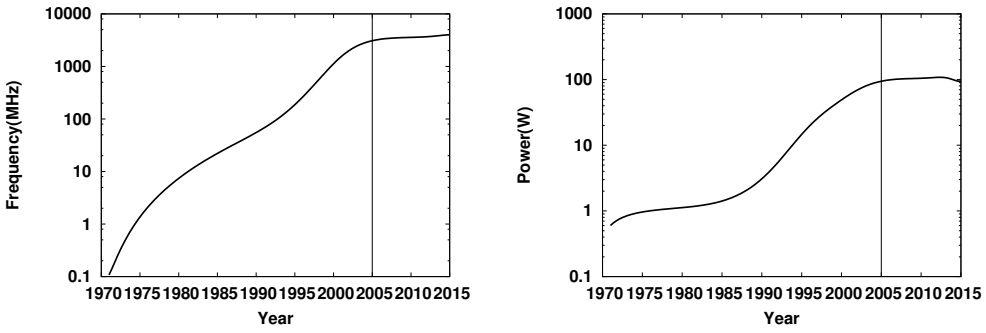
Bibliography **123**

Chapter 1

Introduction

Ever since the birth of microprocessors, there has been a demand for more computation speed. Historically, this demand has been met by improvements in computer architecture design and clock frequency. Some of the improvements in computer architecture seen over the years include pipelining, branch prediction, superscalar processors, out of order execution, speculative execution, etc. The first Intel 4004 microprocessor released in 1971 had a clock frequency of 740 kHz. Thirty years later, the Intel Pentium 4 had clock frequencies more than 2 GHz. These improvements were possible due to advancements in process technology and miniaturisation of transistors which ensured that more and more transistors could be packed on a chip. For e.g., the Intel 4004 was built with 10μm technology and had around 2,300 transistors while the Intel Pentium 4 was built with 180nm and had around 42 million transistors. The increase in transistor count happened approximately in line with Moore's law[84] which predicted a doubling in the number of transistors every year. At the same time, Dennard's scaling [44] predicted that the power density will remain constant as the transistor size reduces. Hence, Dennard's scaling and Moore's law allowed uni-processors to exponentially improve the performance in an energy efficient manner.

However, this well-scripted success story for uni-processors began to break down at the beginning of this century. The difference in the clock frequency of the memory and processor increased and soon memory speed became the bottleneck. Improvements in performance from Instruction Level Parallelism diminished as not enough parallel instruction could be found in a single thread. Increasing frequency led to high power consumption and heat generation. Figure 1.1 shows that the exponential increase in frequency and power has tapered off around 2005 indicating a transition. Hence, at the turn of this century, the processor industry witnessed a major transition from uni-processors to multicore processors. The processor designers concentrated



(a) Progression in clock for Intel Processors (b) Progression in Power for Intel Processors

Figure 1.1: Note the flattening of the curves beginning around 2005.

on obtaining improved performance by increasing the number of cores. Multicore processors are well suited for data-parallel applications and parallel workloads. Researchers suggested adding multiple cores to a system and processors with even a thousand cores[68] have been discussed. In the industry also there has been a proliferation of homogeneous multicores. The high-performance Intel Xeon Processor E7-8880 v3 has 18 cores.

But the success with homogeneous multicores also seems to be short-lived. As process technology improved, Dennard's scaling broke down which caused a huge increase in power densities. It is no more possible to switch on all the transistors simultaneously without causing thermal safety issues. Thus, due to power and thermal limitations, not all portions of the chip can be switched ON at the same time. The portions that cannot be powered ON are called Dark Silicon. When only a subset of the cores can be active at the same time, it is better to have cores of different types than the same type. The presence of heterogeneous cores provides opportunities to match a task with the most efficient core for that task and thus leads to better usage of the power budget. Hence, Dark silicon leads us to heterogeneous multicore processors. Heterogeneous processors differ from homogeneous processors in that the cores are all not of the same type. The difference among cores can be in the ISA or micro-architecture. Research[64] has shown that heterogeneous multicores are better when considering energy delay trade-offs. A multi-core heterogeneous architecture can support a range of execution characteristics not possible in a single-core processor. It is beneficial for workloads with different characteristics. By having processing units with different characteristics, computation can be mapped to specialised devices that perform a specific type of task more efficiently than other devices. Thus, heterogeneous computing

systems can deliver better energy-delay trade-offs [64], [66]. Recently, with the rise of GPGPU (general-purpose computing on GPUs), heterogeneous computing has become more mainstream. An example for a commercial heterogeneous multicore processor is big.LITTLE[18]. It consists of a big (Cortex A15) and a small (Cortex A7) processor. The embedded systems world has also adopted heterogeneous systems. OMAP[6] from TI, Snapdragon[13] from Qualcomm and Tegra[14] from NVIDIA are examples of popular heterogeneous multicores in the mobile embedded systems market.

Heterogeneous processors are found in both the general purpose as well as the embedded space. The design of embedded Multiprocessor System on Chips (MPSoCs) has been largely application driven with specialised units such as ASICs and DSPs targeting media decoding or digital signal applications. Embedded applications developers are typically expert programmers, where the cost of porting and tuning can be amortised across many shipped devices. Because of this, the programming models supported by MPSoCs typically tend to be application driven and change from one platform or generation to the next. The design of general-purpose heterogeneous many-cores has been, by contrast, more conservative. Large scale parallelism is supported, but there is considerably less diversity amongst cores. Same-ISA different scale systems such as big.LITTLE or GPU accelerator approaches have dominated. The reason is that general-purpose programmers require higher-level programming interfaces as the systems are less domain specific. The relative cost of programming is higher and cannot be amortised over many shipped devices

Energy and power have been the driving force behind the gradual convergence of two communities: embedded and general-purpose computing. Both have moved to parallel platforms containing specialised processors. The embedded community were early adopters of such platforms called MPSoCs as they provided cost-effective performance and are energy efficient. The general-purpose community has only recently adopted such platforms, now described as heterogeneous many-cores, as they provide high performance within acceptable power densities. The embedded world has led in system diversity, and it is likely that tomorrow's general-purpose heterogeneous many-core platforms will resemble the MPSoCs of today. This thesis is concerned with connecting these two worlds: the high-level mapping approaches supported in the general-purpose world with MPSoCs. This has the potential to provide today's MPSOC programmers with a portable programming environment and allow the general-purpose programming of tomorrow's heterogeneous many-cores.

Though there are advantages for heterogeneous multicores, they are very difficult

to program. The difficulty arises due to lack of high-level programming models, multiple address space, lack of cache coherence and the fact that cores have different ISAs. Hence, it remains to be seen whether running programs on heterogeneous processors provides performance or energy improvements. To obtain benefits from multicores, the programs have to be parallelised. Parallelising a program to achieve performance benefits on multicore processors is a difficult task, and there are several papers dedicated to this task. This thesis deals with the relatively simpler case of data-parallel programs. In particular, this thesis considers data-parallel OpenMP[42] programs. Since the parallelism is already explicitly provided by the OpenMP programs, the most important task is to map the parallelism to the heterogeneous cores.

The next section presents the problems that have to be solved to map data-parallel programs to heterogeneous multicores processors.

1.1 Problems

The following sections present problems to be solved for effectively using heterogeneous multicore processors for data-parallel programs.

1.1.1 Partitioning

The programs are data-parallel. Hence, mapping involves partitioning the work among the processors. So, the first question to answer is how to partition the data-parallel programs such that performance and energy benefits can be obtained. If the target architecture is homogeneous multicores, then a uniform partitioning is desirable. However, with heterogeneous multicores, it is not clear how to partition a program. For best runtime performance it is obvious that work has to be partitioned among the processors but what is not clear is the amount of work to be assigned to each processor. The heterogeneous processors can have different clock frequencies, ISAs, different strengths in processing and different memory configuration.

For energy, the folklore [130] is that running the fastest is best for energy. Since heterogeneous multicores provide opportunities for energy-delay trade-offs, it is not clear whether this property still holds. Is it best to run everything on the fastest processor, or is it best to choose the partition giving the best runtime, or is it best to run on the most efficient processor.

To evaluate partitioning, a runtime framework has to be built to overcome the obstacles posed by heterogeneous processors. Since the address spaces of processors are different, pointers are not portable. The runtime framework has to perform some address translation so that all the processors can work on different partitions of an ar-

ray. If the caches are not coherent, then the runtime should provide API to flush or invalidate caches so that processors read the updated values.

1.1.2 Compiler Framework

Once the partitioning problem is solved, the next task is to develop a compiler framework to automatically map data-parallel OpenMP programs to heterogeneous processors. OpenMP is a popular high-level parallel programming model and is widely supported on parallel hardware. Programmers only need to express parallelism without concern as to the mapping and scheduling of work to the underlying platform. Due to the wide availability of OpenMP libraries, it also offers some degree of performance portability. OpenMP, however, is not directly supported on heterogeneous systems, due to lack of coherent shared memory and multiple operating systems. Exploiting heterogeneity is avoided, and only the CPUs are utilised by the OpenMP runtime. The standard method of accessing non-CPU cores is to use them as accelerators accessed via platform specific libraries. This style of programming requires the application to be first split into separate tasks that fit the library API. The main CPU then acts as a coordinator using a Master/Slave programming model. Although this approach means that the programmer does not have to deal with different operating systems and manage memory coherence, it is highly system specific and may introduce excessive data traffic. Furthermore, it requires the programmer to perform platform-specific low-level partitioning of work into parallel activities. While this approach may work for specialised applications, it is not suitable for general data-parallel programs.

1.1.3 Power Budget

Power is a first class constraint in modern processor design. It is the driving force behind the shift to multi-cores in today's computing systems. Ever-higher clock frequency scaling is unsustainable due to power-density and thermal limitations [105]. Parallel programming on multiple cores has the potential for increased performance without increased power. Power is also the reason for increased processor heterogeneity. Dark silicon [45] suggests that as it will not be possible to simultaneously power-up all the cores on a chip, cores should be specialised to best utilise the available power.

Simple heterogeneous GPU based systems are now common-place. More challenging heterogeneous platforms are likely to become more prevalent in the future like TI OMAP4430 [6] and Samsung Exynos 5422 [3] platforms which contain heterogeneous CPUs, heterogeneous GPUs and DSP cores. The non-CPU processors in these platforms are generally used for offloading to improve power efficiency/performance.

These processors have different micro-architecture, memory hierarchy and/or Operating Systems. There is no single programming language, library or runtime to program these platforms. Hence, these are highly challenging to program but offer the potential for excellent power and performance.

Given that, it is not possible to run all cores at their maximum clock rate frequently for long periods; runtime power-management is needed to keep within the thermal constraints[105]. Such power management is the responsibility of the operating system or hardware which uses *DVFS* or power gating to reduce power at the expense of performance. This approach works well with homogeneous multi-cores or single-ISA heterogeneous cores as there is no need to modify the binaries. However, if, due to dark silicon [45] there are diverse cores each with their own ISA and specialised behaviour, there is no straightforward approach to using the right cores based on program suitability and available power budget. The task is to select the cores and the hardware settings that meet the budget and give the best performance and then allocate the appropriate amount of work to each core.

1.2 Contributions

This thesis presents the following contributions.

1. Develops a framework for manually mapping data parallel programs to OMAP4, a highly heterogeneous systems. It presents a detailed energy measurement methodology. It explores a partitioning design space and shows its dependency on optimisation goal: energy vs time. A straightforward partitioning approach is presented and evaluated. It is shown that improvements can be obtained by partitioning programs for heterogeneous systems.

2. Develops a Clang/LLVM based compiler approach to map OpenMP programs to MPSoCs. This compiler framework bridges the gap between the high-level programming model of OpenMP and the heterogeneous resources of MPSoCs. It takes OpenMP programs and generates code which runs on all the processors. It delivers programming ease while exploiting heterogeneous resources. The compiler is based on a Single Program Multiple Data model where data layout is explicitly determined, allowing reasoning about memory coherence and synchronisation placement. This compiler approach achieves significant performance improvements over existing approaches on the TI OMAP4.

3. Develops a compiler-based approach to runtime power-management for hetero-

geneous cores. Given an externally provided power budget, it generates heterogeneous, partitioned code that attempts to give the best performance within that budget. At runtime, it selects parameters determining the workload on each core. A number of different selection policies that combine program and system information are presented. This approach is applied to parallel OpenMP benchmarks on the OMAP 4430 and Exynos 5422 platforms.

1.3 Thesis Organisation

The thesis is organised as follows.

Chapter 2 presents the background necessary to understand the work presented in this thesis. This chapter presents an introduction to Multiprocessors and then discusses details, examples and challenges of heterogeneous multicores. Compiler issues in heterogeneous multicores are presented next. And finally, details of the measurements and benchmarks used in the experiments in the thesis are discussed.

Chapter 3 presents a survey of related work. Work related to mapping parallelism, Programming and Compilers for heterogeneous programming and Energy measurements are discussed.

Chapter 4 presents a design space exploration in the partitioning of data-parallel programs for heterogeneous multicore processors. Through the exploration, it is shown that the best partitioning varies with optimisation criteria and benchmarks. A straightforward method for partitioning programs is also presented.

Chapter 5 presents a compiler framework for mapping data-parallel programs to heterogeneous MPSoCs. It is shown that smart insertion of cache flushes is necessary to attain good performance in a non-cache coherent heterogeneous multicore processor.

Chapter 6 explores issues in code generation for heterogeneous processors in the presence of a power budget. Typically, the OS or hardware changes the voltage/frequency using DVFS to meet the power budget. Heterogeneity provides another dimension for meeting the power budget. This work presents an approach for code generation in the presence of a power budget for heterogeneous multicore processors.

Chapter 7 concludes this thesis and presents the main contributions. The chapter also includes a critical analysis of the solutions presented as well as discusses possible future work.

1.4 Summary

This Chapter provided an introduction to the work presented in this thesis. The Chapter provided a brief overview of the problems in mapping parallelism to heterogeneous multicores and also summarised the contributions of this thesis. Finally, it also provided the organisation of the Chapters of this thesis.

The next Chapter provides background reading for understanding the work presented in this thesis.

Chapter 2

Background

This chapter presents the technical background necessary to understand the material presented in this thesis. The first Section 2.1 introduces multiprocessors. Section 2.2 covers heterogeneous multiprocessors which is the focus of this thesis. Parallel programming frameworks which can be used for programming multiprocessors are introduced in Section 2.3. Compiler issues relevant to heterogeneous processors are described in Section 2.5. Section 2.6 describes issues in runtime and power measurements. Finally, Section 2.7 briefly discusses the various benchmarks used in the experiments presented in this thesis.

2.1 Multiprocessors

There is always a demand for increased computation speed from processors. Microprocessor architects have traditionally met this requirement by increasing processor frequency, improvements in memory hierarchy (like caching), pipelining, compilers/processors exploiting instruction level parallelism, etc. However, there has been greatly diminished gains in processor performance because of limited instruction level parallelism, difference in speeds of the memory and processor, and an exponential increase in power consumption for every small increase in frequency. Due to the diminishing returns, processors makers have adopted a different approach to improving a computers performance by adding extra processors. A multiprocessor is a processor which contains more than one processor core.

These multiprocessors can directly address and access all the memory in the system. However, depending on the location/speed of memory access, the Multiprocessors are generally categorised into two types. In **Uniform Memory Access (UMA)** machines, all processors require the same time to access any memory location. The multicore systems used on the desktops/workstations and the MPSoCs used in em-

bedded systems fall under this category. In **Non Uniform Memory Access (NUMA)** machines, a processor can access the memory associated with it quickly, but to access memory associated with another processor it requires to wait for more time. In this thesis, the focus is on UMA machines.

Depending on the kind of processors involved, Multiprocessors can also be categorised into two, **Homogeneous** and **Heterogeneous**. If all the processors in a multiprocessor are of the same kind, then it is a homogeneous processor. Otherwise, it is a heterogeneous multiprocessor. Most of the processors in desktops and workstations are homogeneous.

2.2 Heterogeneous Systems

The term Heterogeneous Systems encompasses a wide variety of processor systems. The term is commonly used to define multiprocessor systems where the processors have different capabilities and architecture. Heterogeneous processors provide better power-performance trade-offs. This is advantageous when a program has sections of code with different characteristics or if the workload consists of programs with different characteristics. Heterogeneous systems can be classified into two types based on ISA viz. single ISA and multiple ISA.

2.2.1 Single ISA

Single ISA heterogeneous systems are systems which have processors with a single-ISA but different micro-architecture. e.g., big.LITTLE processors from ARM.

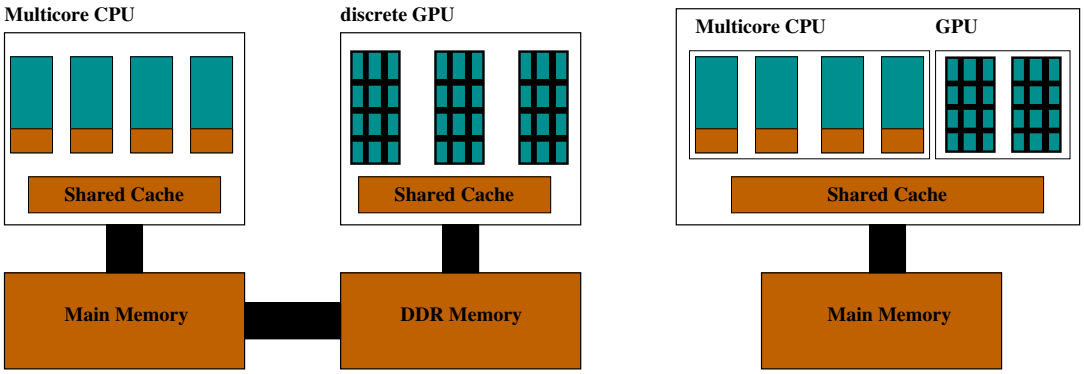
2.2.2 Multiple ISA

Multiple ISA heterogeneous systems are systems which have processors with different ISAs. There are mainly two types of Multiple ISA systems that are commonly seen, CPU-GPU systems and MPSoCs. e.g., Intel IvyBridge and Samsung Exynos.

CPU-GPU Systems

The most common heterogeneous systems are CPU-GPU systems. CPU-GPU systems, as the name suggests, are heterogeneous systems with a multicore CPU and a GPU. Traditionally, the GPUs were used for graphics processing. But over the years, the GPUs were found to be suitable for programming data-parallel applications. These GPUs are composed of a number of Stream Processors/Shader cores. Each core has many hardware threads with a single program counter and works as a Single Instruction Multiple Data (SIMD) machine. CPU-GPU systems can either be discrete or

integrated. In discrete systems (Figure 2.1a), GPUs sit on an external card which is connected to the CPU through the PCI bus. Recently, the GPUs have been integrated (Figure 2.1b) on the same chip like in Intel Ivy Bridge, AMD Fusion and many Embedded MPSoCs like Samsung Exynos. Generally, the CPU is used for computing code with control, and the GPU is used to compute code which has lot of data-parallelism.



(a) discrete (b) integrated

Figure 2.1: GPU

MPSoCs

Multiprocessor System on Chip (MPSoC) is a VLSI system that integrates all of the systems required to run an application on a single chip. It contains several processors that are specialised for various purposes. MPSoCs are widely used in the embedded systems world particularly in cellphones, network and signal processing, and multimedia. MPSoCs started off as homogeneous systems, but later, due to performance and power requirements, these became heterogeneous. A survey of the history of MPSoCs can be found in [124]. In this section two MPSoCs, viz. OMAP and Exynos are discussed in detail.

The TI OMAP4430[7] is a typical mobile application MPSoC consisting of various subsystems including general purpose processors, a programmable multimedia engine plus graphics and image accelerators. Figure 2.2 shows the components of the OMAP4430 relevant to this thesis. A dual core ARM Cortex-A9 provides general purpose computing. Both processors share a common L2 cache and address space. There are two smaller ARM Cortex-M3s available for smaller and lower power tasks. Both share a common L1 cache and are responsible for controlling the graphics and image accelerators. There is also distinct programmable multimedia engine based on a TI mini-C64X+ DSP. The PowerVR GPU cannot be currently programmed, since the

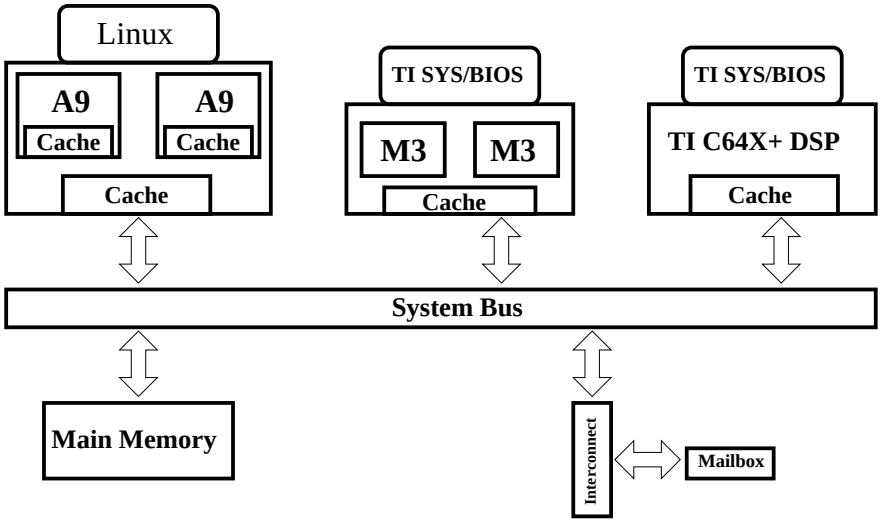


Figure 2.2: OMAP4 block diagram

OpenCL driver is not publicly available, and is not considered in our study.

The Exynos 5422[3] is a Samsung platform consisting of 4 ARM Cortex A15 big CPUs, 4 ARM Cortex A7 big little CPUs and 1 Mali T628 MP6 GPU. The Mali T628 consists of 2 devices: one with 4 shader cores, the other with 2. Also, there are 20 DVFS levels available on the A15s, 15 on the A7s and 6 on the Mali. A program can run on any combination of the processors. Figure 2.3 shows the components of Exynos platform relevant to this thesis.

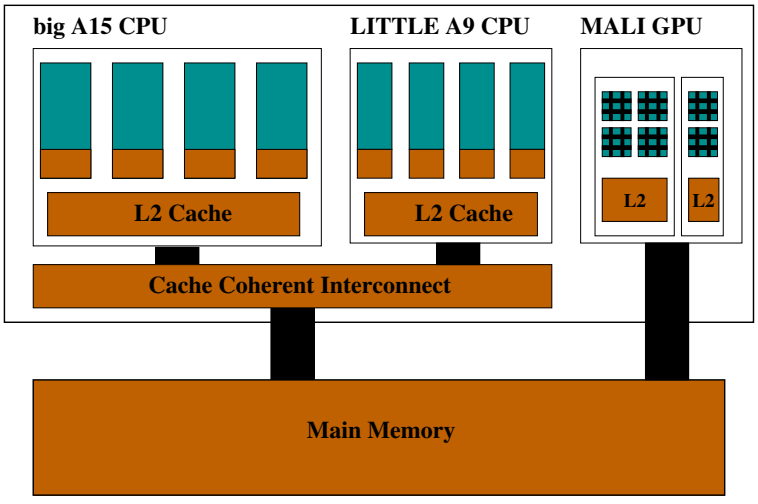


Figure 2.3: Exynos block diagram

2.3 Parallel Programming Models

Parallel programming models are abstractions for modelling parallelism in applications. There are various kinds of parallel programming models like data-parallel model, task parallel model, stream parallel model, pipeline parallel model etc. This thesis focuses on data-parallel model.

2.3.1 Data Parallel Model

In data-parallel model, the same operation is performed on different sections of an array at the same time. This kind of parallelism is normally found in array based programs with corresponding parallel loops. These are programs where individual elements of an array/data structure can be independently computed. Such programs are widely found in embedded systems applications.

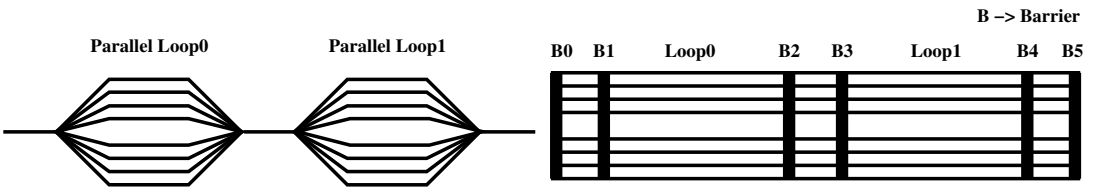
The focus of this thesis is data-parallel programs. There are many ways to implement data-parallelism. Two methods, fork-join and SPMD are discussed in the next two paragraphs. These two methods are pictorially represented in Figure 2.4.

fork-join

In the fork-join model, execution starts as a single master thread and on reaching a parallel region, threads of execution are forked off and later these threads join at a point and sequential execution resumes. By default, the number of threads forked is equal to the number of available processors. However, systems implementing fork-join model typically provide features to control the number of threads. The number of threads are not fixed for the whole program and can vary for different parallel regions. Fork-join model provides ease of programming since it is necessary to only specify the parallel regions. Fork-join does not require the data/work to be partitioned in advance. Work partitioning is implicit in the fork-join model, and typically it is the job of the runtime to partition work among the available processors. Since partitioning is performed by the runtime, the work distribution can adapt to the workload. However, presence of a runtime for partitioning and forking and joining of threads introduces overheads which can negatively impact system performance. eg. OpenMP.

SPMD

The Single Program Multiple Data (SPMD) model of parallelism is well known and used in a variety of settings. In SPMD, parallel tasks run the same program but operate on independent sections of an array. It is a data-centric approach where data is first partitioned and scheduled to processors. Computation is then partitioned and sched-



(a) fork-join (b) spmd

Figure 2.4: Programming Models

uled in accordance with this data allocation. Nearly all schemes use an owner-compute or local-write rule where the code executed on a processor is restricted to write to just local data. Unlike the fork-join model, there is no master thread. At program start itself, all the threads start running in parallel. The number of threads correspond to the number of processing elements and are fixed throughout the execution of a program. Synchronisation is not associated with the beginning or end of parallel loops but whether or not there is a cross-processor data dependence i.e., data written on one processor is read on a different processor. Barriers are inserted to honour such dependences. e.g., Unified Parallel C (UPC) is an extension of the C language which uses the SPMD model.

2.4 Parallel Programming Frameworks

To take advantage of the multiprocessors, a parallel programming framework should be used. A parallel programming framework implements one or more of the parallel programming model described in the previous section. The frameworks are usually implemented as language extensions and libraries on top of existing programming languages like C/C++, Fortran, etc. In this section, we take a look at some relevant parallel programming API/libraries, pthreads, OpenMP and OpenCL.

2.4.1 POSIX threads

POSIX threads (or pthreads)[85] is a portable standard for threads. pthreads API is available for many operating systems like Linux, FreeBSD and MacOS. The API provides functions for creating and manipulating threads. Threads are created using pthread_create and pthread_join is used to wait for termination of threads. The API also provides for mutual exclusion and synchronisation between threads. There are also functions for signalling and waiting on condition variables. An example for vector addition using pthreads is given in Listing 2.1. The function *vector_add* is executed by each thread.

```
struct thread_params {
    int start_indx ;
    int end_indx ;
} ;

void *vector_add(void* t)
{
    struct thread_params* tp = (struct thread_params*)t ;
    int i ;
    for(i=tps->start_indx ; i<tps->end_indx ; i++) {
        C[i] = A[i] + B[i] ;
    }
    return NULL ;
}

void vector_add_threads()
{
    pthread_t thread[NUM_THREADS] ;
    thread_params tp[NUM_THREADS] ;
    int unit_size = N/NUM_THREADS ;
    int cur_start = 0 ;
    for(int i=0 ; i < NUM_THREADS ; i++) {
        tp[i]->start_indx = cur_start ;
        tp[i]->start_indx = cur_start + unit_size ;
        pthread_create(&thread[i], NULL, vector_add,
                       (void*)&tp[i]) ;
        cur_start += unit_size ;
    }
    for(int i=0 ; i < NUM_THREADS ; i++) {
        pthread_join(thread[i], NULL) ;
    }
}
```

Listing 2.1: pthread vector addition

2.4.2 OpenMP

OpenMP[42] is a widely used high-level shared memory pragma based programming language. The popularity of OpenMP is due to its ease of use and wide availability. OpenMP consists of a set of compiler directives in the form of comment like pragmas. These are used to mark parallel loops and tasks. There are also directives for specifying scheduling of work, declaration of both shared and private variables and critical sections. OpenMP supports the *fork-join* programming model. In this model, a program starts as a single master thread. When an OpenMP parallel construct is reached a team of threads is *fork*ed. These threads co-operatively perform the work in parallel. At the end of the construct all the threads *join* and only the master thread continues. Rather than creating and destroying threads at the start and end of parallel regions, most implementations have a pool of threads available which are used and synchronised as

needed. An example for OpenMP vector addition is given in Listing 2.2.

```
#define N 100
void vector_add()
{
    int i ;

    #pragma omp parallel for
    for (i = 0; i < N; i++)
    {
        C[i] = A[i] + B[i] ;
    }

    return 0 ;
}
```

Listing 2.2: OpenMP vector addition

2.4.3 OpenCL

OpenCL is a programming framework for heterogeneous devices. It has gained a lot of popularity for programming GPU devices. In the OpenCL hardware architecture, there is a host which controls various Compute Devices. The Compute Device is composed of many Compute Units, and each Compute Unit is further composed of Processing Elements.

The code that runs on the host is called the host code and that which runs on the device is called kernel code. The host code is responsible for setting up the OpenCL environment, command queues, buffers and launching the kernel. The OpenCL buffers are different from the memory that can be accessed by the host device. So the data from the memory has to be copied to the device buffer before computation can begin and copied back after computation. If the copy overhead has to be overcome then the flag *CL_MEM_USE_HOST_PTR* has to be used when creating OpenCL buffers. The host code performs all these operations using OpenCL API. Command queues have to be created for each compute device. These queues can then be used for submitting OpenCL kernels for execution on the compute devices. The OpenCL kernels execute in parallel on the Processing Elements.

```
//file: vector_add.cl
__kernel void vector_add(__global int* A,
                         __global int* B,
                         __global int* C)
{
    int i = get_global_id(0) ;
    C[i] = A[i] + B[i] ;
    return 0 ;
}
```

```
int main( int argc, char* argv[] )
{
    cl_int err;
    //Create OpenCL queue, context, program etc
    cl_kernel = clCreateKernel(program,"vector_add",&err);

    // Size, in bytes, of each vector
    size_t bytes = N*sizeof(int);
    size_t globalSize = N;

    // Device input and output buffers
    cl_mem dA = clCreateBuffer(context,CL_MEM_READ_ONLY,bytes,NULL,NULL);
    cl_mem dB = clCreateBuffer(context,CL_MEM_READ_ONLY,bytes,NULL,NULL);
    cl_mem dC = clCreateBuffer(context,CL_MEM_WRITE_ONLY,bytes,NULL,NULL);
    // Copy the contents of arrays A and B to device memory
    err = clEnqueueWriteBuffer(queue,dA,CL_TRUE,0,bytes,A,0,NULL,NULL);
    err |= clEnqueueWriteBuffer(queue,dB,CL_TRUE,0,bytes,B,0,NULL,NULL);

    // Set kernel arguments
    err  = clSetKernelArg(kernel,0,sizeof(cl_mem),&dA);
    err |= clSetKernelArg(kernel,1,sizeof(cl_mem),&dB);
    err |= clSetKernelArg(kernel,2,sizeof(cl_mem),&dC);

    // Execute the vector_add_kernel
    err = clEnqueueNDRangeKernel(queue,kernel,1,NULL,&globalSize,NULL,0,NULL,NULL);

    // Wait for kernel to finish execution
    clFinish(queue);

    // Copy results from the device
    clEnqueueReadBuffer(queue,dC,CL_TRUE,0,bytes,C,0,NULL,NULL);
    // release OpenCL resources
    clReleaseMemObject(dA);
    clReleaseMemObject(dB);
    clReleaseMemObject(dC);

    //Free kernel, queue, context, program etc

    return 0;
}
```

Listing 2.3: OpenCL vector addition

A single unit of the OpenCL kernel is called a work-item. Work-items are arranged in an n-dimensional space, and the number of dimensions and the number of work-items in each dimension is specified in the launch command. An example for OpenCL vector addition is given in Listing 2.3. Here *vector_add* is the kernel that gets executed on each processing element. API calls for obtaining the platform and device IDs, creating and destroying the context, creating and destroying the program and building the program

are omitted for brevity.

2.4.4 Comparison

Table 2.1 compares the characteristics of pthreads, OpenMP and OpenCL parallel programming frameworks. For all these frameworks the parallelism has to be explicitly specified. While partitioning is explicit for OpenMP, it is implicit for pthreads and OpenCL. OpenCL follows the SPMD model whereas pthreads and OpenMP follows the fork-join model.

	pthreads	**OpenMP**	**OpenCL**
Parallelism	Explicit	Explicit	Explicit
Partitioning	Explicit	Implicit	Explicit
Model	fork-join	fork-join	SPMD

Table 2.1: Characteristics

2.5 Compiler

Computers are programmed using high-level programming languages. Using programming languages helps raise the abstraction level and makes the programmer more productive. Compilers help hide the details of the microarchitecture of the computer and translate the code in a high-level language to machine code. The Compiler ensures that the code generated is correct and the code generated matches the behaviour of the high-level source code. In multiprocessor systems, the compiler converts the code in a parallel programming framework and maps it to the various processors. The following subsections discuss the role of compilers in various kinds of processors.

2.5.1 Uniprocessors

In traditional single processor systems, besides correctness, it is the job of the compiler to optimise the generated code. The optimisation criteria is generally performance, but in embedded systems it can be size of the generated code or energy consumption.

2.5.2 Homogeneous Multiprocessors

Compiler generates code which works along with a runtime to run threads on different processors. Since all the processors are identical, the processors run the same code on all the processors. Since the performance of all the processors are identical, work can be uniformly partitioned and allocated to each of the processors.

2.5.3 Heterogeneous Multiprocessors

Heterogeneous multiprocessors bring new challenges of its own. And the challenges vary depending on the kind of heterogeneity involved. Since the processors behave differently, code cannot be uniformly partitioned for all the processors. The other option is to use dynamic partitioning. But dynamic partitioning has increased overhead. This might also introduce additional complications for insertion of barriers and cache flushes in non-cache coherent architectures. The following paragraphs discuss issues that a compiler for heterogeneous processors has to deal with.

ISA

If the ISA is the same as in big.Little Processors, then the same binary can run on all the processors. If the ISA is different as in OMAP Processors, then different binaries are required. This issue is overcome by using a source to source translator to convert the code into intermediate form and then using separate compilers for each processor.

Cache coherency

If the caches are not coherent, then, a processor will not be able to read the values updated by another processor. Hence, if there are inter-processor dependencies in the code, it is the job of the compiler to insert flushes so that the processors read the updated values.

Address space

If address space is not the same for all the processors, then pointers are not portable. Hence, support is required from the runtime to convert the address so that all processors have access to the same arrays/memory locations.

2.5.4 Clang/LLVM

LLVM [4] is a compiler framework. Clang is the frontend which does all the type checking and converts the input code into intermediate format. The LLVM optimiser then does optimisations on this IR and then a backend converts the optimised IR to machine code.

2.6 Measurements

The experiments in this thesis mainly measure two quantities, runtime and power. This section dives briefly into the measurement of runtime and power.

2.6.1 Runtime

Runtime is measured separately for each core and the entire system. Runtime is measured by using the *gettimeofday* system call. Calls are made to this function before and after the execution, and the runtime is the difference of the times. All the runtime measurements are made on the CPU side. System calls form part of any communication between the processors. Hence, the runtime of these calls has to be included in our runtime measurement. An API function which measures the system time as well as user time is required. For these reasons, the *gettimeofday* API function is used. This function is called before forking off the threads and after the threads join. The difference in time between these two calls is the execution time.

2.6.2 Energy

Energy is the sum of power over time. Measuring power and hence energy is highly non-trivial. This section describes in detail the methodology used to accurately measure power and discusses how it should be accounted for.

Power consumption is found by measuring the current consumed by the processor and multiplying by the supply voltage. The current consumption cannot be directly measured from the whole board supply since it is very noisy and hence is not accurate enough to distinguish between partition decisions. Also, the individual consumption of each processor has to be measured. Hence, each processor's power consumption is separately measured. The power consumed by the memory has to be measured. To measure the current, a resistor is found in the path of these power rails and the voltage drop across the resistor is measured and divided by the resistance. This method is described in detail by Jos [115].

Power variation

The method described gives a good insight into the power consumption. However in programs with phases, synchronisation and heterogeneous loops, the power consumption varies during the execution of an application. Hence, the voltage has to be continuously monitored. For this purpose, an oscilloscope is used, and readings are taken every millisecond. Figure 4.7a shows the pandaboard with probes attached to one pair of pins to take energy measurements. Figure 4.7b shows the oscilloscope with a measurement of the processor to which the probes are attached in Figure 4.7a. The resistance is measured using a digital multimeter. For our experiments, a Tektronix MSO4104 oscilloscope and Agilent 34410A digital Multimeter is used. Figure 4.7d

shows a voltage reading from the oscilloscope in the regular sampling mode for the CPU when running the floydwarshall benchmark on the pandaboard. As can be seen, there is significant noise in the measurement. This can be overcome using either post-filtering using software or using the High Resolution measurement mode of the oscilloscope. Figure 4.7e shows the same measurement after filtering. All the rails are measured separately and summed up to find the total power/energy consumption.

2.7 Benchmarks

Benchmark	Short Name	Arrays	Suite
Matrix Multiplication	mxm matm	A[1024][1024], B[1024][1024], C[1024][1024]	DSPstone
Dotproduct	dotp	A[4194304], B[4194304]	DSPstone
Edgedetect	edge	image_buffer1[2048][2048], image_buffer2[2048][2048], image_buffer3[2048][2048]	UTDSP
Histogram	hist	image[4096][4096]	UTDSP
Doitgen	dgen	A[64][64][64], sum[64][64][64]	Polybench
Regdetect	regd	diff[64][64][2048], sum_diff[64][64][2048], mean[64][64], path[64][64]	Polybench
FloydWarshall	flwl	path[512][512]	Polybench

Table 2.2: Benchmarks

The benchmarks that are used in this thesis are data-parallel benchmarks taken from DSPstone[113], UTDSP[70] and Polybench [92]. These benchmarks are parallelised by annotating with OpenMP pragmas and then used for our experiments. Table 2.2 shows the name of the benchmark, short name used in the diagrams, the size and name of the arrays partitioned and the benchmark suite it belongs to.

Matrix Multiplication and Dotproduct are from the DSPStone benchmark. Matrix Multiplication multiplies two matrices and stores the result in a third matrix. Dotproduct multiples two vectors and stores the result in a third vector. Dotproduct has a reduction. Edgedetect and Histogram are from the UTDSP benchmark suite. Edgedetect is an edge detection algorithm for images; it has a few convolutions and application of a threshold. Histogram computes the histogram of an image and uses the histogram

to make a grayscale mapping and finally uses the mapping to create the output image. Doitgen, Regdetect and FloydWarshall are from the Polybench suite. Doitgen is a multiresolution analysis kernel, and it contains some matrix operations. Regdetect is a regularity detection algorithm for 2D images. FloydWarshall is the all-pair shortest path algorithm.

2.8 Summary

This Chapter presented the necessary background for understanding the work presented in this thesis. The next Chapter presents a survey and commentary of published work related to the work in this thesis.

Chapter 3

Related Work

There is a large amount of related work in various areas associated with mapping parallelism for heterogeneous systems. A survey and discussion of the related work are presented in this Chapter. Section 3.1 presents a survey of work in heterogeneous multicore architecture. Related work in the area of Programming Languages for heterogeneous multicores is presented in Section 3.2. Section 3.3 presents related work in mapping parallelism. Section 3.4 presents an overview of work related to memory management issues. Finally, Section 3.5 presents an overview of work related to power and energy.

3.1 Heterogeneity

Multicore processors have now become ubiquitous in desktops and workstations. At the same time, these processors consume a lot of power. For e.g., Intel Core i7-5960X Extreme Edition based on Haswell-E Micro-architecture using 22 nm technology at 3 GHz with 8 Cores consumes around 140 W. Research in the last decade has advocated using heterogeneous multicores to get better power efficiency. Heterogeneous multicores are also widely seen in the Industry. In this section, prior research on heterogeneous multicore architectures is surveyed. Heterogeneous processors can be broadly classified into two categories, single ISA and multiple ISA. As the name suggests, the cores in a multiple ISA heterogeneous multiprocessor have different ISA while that on a single ISA heterogeneous multiprocessor have the same ISA. Venkat et.al[118] advocate heterogeneous-ISA multiprocessors by showing that they are more efficient. Single-ISA heterogeneous architectures are easier to program, and the runtime scheduling is also simplified because the same binary can be scheduled on different processors.

3.1.1 Single ISA heterogeneity

Different applications have different resource requirements. Using the same processor architecture for different applications does not seem to be optimal. Even different sections of a single application have different resource requirements. Based on this argument, Kumar et.al[65] advocated single ISA heterogeneous processors for power efficiency. Single ISA heterogeneous processors are composed of cores which have the same ISA but different micro-architectural features or parameters. Single ISA Heterogeneity can be accomplished primarily by two methods. i) Varying the hardware on-board. For e.g., varying the issue-width, size of caches, reorder buffer, register file, etc. ii) Varying the frequency of the cores by DVFS. Lukefahr et.al[80] showed that using heterogeneous cores is better than DVFS.

While designing heterogeneous single ISA processors, there are many questions to answer. The space for a design space exploration is large, and it is not possible to exhaustively search. Metrics are also needed to measure and compare the various hardware configurations. Tomusk et.al[108] argues that choosing heterogeneous cores is a difficult task since the optimality of cores varies with programs. Various metrics (flexibility, non-uniformity, gap overhead, set overhead and generality) for evaluating heterogeneous processors are presented in [110] and [109].

3.1.2 Multiple ISA heterogeneity

Multiple ISA heterogeneity has been widely investigated, and there exist many proposals to implement heterogeneity. One such way is to add specialised cores to the system. Traditionally, microprocessors were designed with a goal to achieve the best performance for 90% of the workloads. This makes these processors less energy efficient. 10x10[32] is a heterogeneous architecture which splits the workload into 10 different categories and design specialised cores for each of these 10 categories. GreenDroid[49] is a heterogeneous architecture where a processor is augmented with several cores specific to the applications.

Many research groups have explored multiple ISA heterogeneity by creating a heterogeneous system by combining more than one type of processors. Pangaea[125] is a heterogeneous chip multiprocessor with tightly coupled IA32 CPU cores and nonIA32 GPU cores. Marisabel et.al[53] explores the design space of a heterogeneous mix of server-class Xeon processors and mobile-class Atom processors and finds an optimal balance that achieves performance and energy-efficiency. Guo et.al[55] presents a heterogeneous architecture combining a LEON3 host processor and a Data Parallel

Coprocessor (DPC). Twin Peaks[128] is a hardware-software co-designed heterogeneous multicore virtual machine which is an architecture with a wide in-order core and a narrow out of order core. Chung et.al[36] reports greater energy efficiency by combining traditional processors with unconventional cores like custom logic, FPGAs, or GPGPUs.

Another approach is to create heterogeneity by having a shared reconfigurable logic along with uniform cores. ReMAP[122] is a reconfigurable heterogeneous multicore architecture. In this architecture, a multicore is augmented with some reconfigurable logic. This reconfigurable logic is shared by all the cores. The cores can be separately augmented with additional functionality using this shared reconfigurable logic. The reconfigurable logic can also be used for inter-core communication and synchronisation.

The industry has also widely adopted heterogeneous multicore processors. Cell[62] was an early example of a heterogeneous multicore architecture which contained two types of processors, the Power Processing Element (PPE) and the Synergistic Processing Element (SPE). The PPE acts as a controller for the SPEs. A common kind of heterogeneity that is found is CPU-GPU systems such as Intel Ivy Bridge and AMD Fusion. An extreme example of heterogeneity is Xilinx Ultrascale MPSoC[15] which has ARM Cortex-53 cores, ARM Mali GPU, ARM Cortex R5 micro-controllers and programmable logic. The Embedded systems community has been the champions of Heterogeneous Processors. OMAP[6] from TI, Snapdragon[13] from Qualcomm, Exynos[3] from Samsung and Tegra[14] from NVIDIA are popular examples.

This thesis focuses on compiler approaches for using heterogeneous systems. In this work, TI OMAP and Samsung Exynos are used for experiments.

3.1.3 Prototyping Heterogeneity

A framework for quickly prototyping heterogeneous computers is presented in [33]. Bakker et.al [20] presents a study where the Intel Single Chip Cloud Computer can be used for modelling a heterogeneous MPSoC and making power and runtime measurements. Fabscalar[35] is a toolset for composing heterogeneous RTL cores in a canonical superscalar template. HaDes[112] is an efficient approach for synthesis of dark silicon heterogeneous chip multiprocessors. The HaDes system determines the number of cores of each type such that the power and area budgets are met, and performance is maximised.

3.2 Programming Languages

There are many programming frameworks and APIs for heterogeneous systems. This section surveys a few of these.

Cuda[2] and OpenCL[10] are APIs for CPU-GPU systems and use a master/slave accelerator model. These languages have special features to express task and data-parallelism. C++ Accelerated Massive Parallelism (AMP)[50] from Microsoft is a new C++ language feature and an STL like library for writing programs that execute efficiently on data-parallel hardware like GPUs. Open Accelerators (OpenACC)[9] developed by a consortium of Cray, CAPS, Nvidia, and PGI is an API for parallel computing for heterogeneous CPU/GPU systems. The code is annotated with pragmas/compiler directives to identify parallel regions of code that should be accelerated. A study[41] shows that even though languages like C++ AMP and OpenACC raise the abstraction level, they do not yet offer enough flexibility to extract the required performance.

Cell-Ss[21] exploits task level parallelism and requires source annotations as input, for identifying parallelism. OmpSs[8] is a programming model that supports asynchronous parallelism and heterogeneity. OmpSs is designed by combining features from OpenMP and the StarSs[90] Model. MAPS[103] is a compiler framework for programming MPSoCs using the streaming programming Model. It provides extensions to the C language for supporting Process Networks. Multicore Asynchronous Runtime Environment (MARE)[5, 96] from Qualcomm is a C++ API for parallel programming. It provides features for taking advantage of the full power (use all processors) of a heterogeneous system. The API provides features for creating tasks and setting dependencies between them. OpenCL code which can be run on GPUs can be wrapped in a task.

OptiML[29] is a Domain Specific Language (DSL) for machine learning which can be used to program heterogeneous systems. With this DSL, the domain knowledge can be used to express implicit parallelism.

This thesis chooses OpenMP as the source language. OpenMP is a widely used parallel API for programming shared memory multiprocessors. It is also a very easy to use API. OpenMP cannot be directly used on heterogeneous multicore processors. In this thesis, OpenMP is converted to SPMD form to run on heterogeneous multicore processors.

3.2.1 Compiler

Compilers and runtimes for heterogeneous processors have to perform additional work. There are two main issues that have to be considered by the compiler. i) Often the parallelism present is not suitable for execution on a heterogeneous processor. Hence, it has to be transformed to a form that is suitable. In the work presented in this thesis, an OpenMP data-parallel program is converted to SPMD form. ii) Some heterogeneous processors do not have a cache coherent system. For these systems, it is the job of the compiler to handle coherency in software.

3.2.2 OpenMP to SPMD

Liu et.al[78] presents a method for the translation of OpenMP Code into SPMD style with Array Privatisation. The translation is performed for use on High Performance Computing with uniform cores and a shared coherent address space. SPMD code consists of a set of threads that synchronise at barriers. To improve performance, the number of barriers has to be minimised. Prior work[111], [87] discusses techniques for reducing the number of barriers. This thesis builds on techniques presented in these papers for synchronisation and cache coherence.

3.3 Mapping Parallelism

Mapping parallelism present in a program to the underlying hardware architecture is a challenging task. The best mapping varies with architecture and is also dependent on program properties. Parallelism mapping can be broadly categorised into partitioning and scheduling.

3.3.1 Partitioning

Bodin et.al[23] and Agarwal et.al[17] present theoretical frameworks for automatically partitioning parallel loops to minimise cache coherency traffic on shared-memory multiprocessors. Michel et.al [48] uses ILP to partition software for energy efficiency. MAPS[72] deals with programming for MPSoCs. The MAPS system extracts parallelism from sequential applications by partitioning and generates tasks for parallel execution on MPSoCs. Ravi.et.al[95] looks at offloading some work to be executed on a co-processor. In one of the techniques they discuss, sub-offload, they partition a loop for execution on a co-processor for better performance.

CPU-GPU Systems

Grewe et.al[52] presents a machine learning based method for partitioning programs between CPUs and GPUs. A two level predictor is employed where the first predictor decides whether the program has to be partitioned and the second predictor decides the ratio of partitioning. Qilin[79], constructs linear performance models at runtime. These performance models are sensitive to the size of the tasks also. The model is constructed by running a portion of the program on the CPU and the GPU. Jiang et.al.[60] and Shirahata et.al[104] runs tasks on both the CPU and the GPU to compute the individual runtime the first time a task is run. The observed runtimes are then used to proportionally partition the work between the CPU and the GPU. Mitra et.al[93] also partitions programs based on linear performance models and individual runtimes. In addition, they also calculate the degradation in performance of the CPU when running in parallel with the GPU and vice-versa. The work presented in this thesis also uses individual throughput(derived from runtime) of the processors to calculate the partitioning. Our work differs from the others in the following ways. 1) Rather than computing partitions on a per-kernel basis we partition the whole program. 2) The presence of inter-processor communication introduces overheads and hence affects the partitioning. We capture this aspect in our partitioning formula. 3) Our partitioning method is generalised for a set of heterogeneous processors and is not restricted to CPU-GPU systems.

3.3.2 Scheduling Tasks

Prior work has explored runtime scheduling of tasks compared to compile time partitioning. Bower et.al[24] advocates tackling the heterogeneity at the OS scheduler level. Most other proposals involve a runtime.

Judit et.al[91] presents a Self Adaptive framework for choosing different versions of code at runtime for heterogeneous processors. In work presented in [106], phase transition points are marked and the application is switched to another core at these transition points for better performance. Kwon et.al [69] discusses issues in virtualisation on heterogeneous systems and presents a hypervisor scheduler which is heterogeneity aware and achieves improvement in performance. Whare-map[82] presents scheduling techniques for scheduling applications in a heterogeneous warehouse. The presented technique solves the mapping problem by formulating as an optimisation problem. The optimisation process uses profiling information.

Cong et.al[38] discusses energy efficient scheduling techniques for heterogeneous

multicore architectures. They develop a regression model to predict the energy consumption of each processor and use it along with code instrumentation and static analysis to perform scheduling at runtime. ACCESS[61] is a smart scheduling method for asymmetric cache CMPs. Applications with a small working set size and those which are streaming do not need a large size cache. Higher energy efficiency can be achieved by scheduling such applications on the cores with the smaller cache, switching off larger caches on lighter loads and running the processor at a lower voltage since the smaller cache can be operated at a lower voltage. Twin Peaks[128] dynamically chooses either a wide in-order machine or a narrow out of order machine for execution to achieve higher energy efficiency. The scheduling algorithm uses current and historical performance data. Raghunathan et.al[94] presents results which show that the best cores to choose vary with the job arrival rate.

This thesis deals with partitioning of data-parallel programs, and the focus is on minimising runtime, energy and meeting a power budget. Dynamic scheduling based approaches suffer from overheads particularly on cache incoherent systems and systems with multiple address spaces.

3.4 Memory Management

The main issues in memory management for heterogeneous systems are multiple address spaces and handling non-cache coherent systems.

3.4.1 Multiple Address Spaces

Gelado et.al[47] suggests maintaining shared logical memory space for CPUs to access objects in the accelerator physical memory but not vice-versa. This method has the disadvantage that pointers from CPUs cannot be passed to the GPUs. Heterogeneous Systems Architecture (HSA)[98] is a standard developed for seamless integration of CPUs, GPUs and hardware accelerators. Traditionally, CPUs and GPUs have different address spaces, but in HSA-compatible systems, the GPUs can access the CPU virtual addresses. This enables passing of pointers between CPUs and GPUs. It also provides for optional coherence between the CPU and GPU through acquire and release instructions.

This thesis overcomes multiple address spaces issue by allocating memory in contiguous memory and appropriately mapping to each processor. The details of the method are discussed in Section 5.7.

3.4.2 Cache Coherence

Igor et.al[107] provides a survey of software cache coherence mechanisms and suggests a classification. Veidenbaum[117] gave a formal definition and proof about the necessary conditions for cache incoherence. Lynn et.al[34] proposes compiler directed cache coherence with additional but minimal hardware support for task parallel programs. Hock-Beng et.al[75] proposes compiler directed cache coherence through prefetching. Dynamic Binary Translators frequently cache the translations, but when the limit is reached some translations have to be selectively flushed. Guha et.al[54] discuss selective flushing in this scenario. Reflex[76] and K2[77] are two systems for heterogeneous processors which perform cache coherence with the help of Operating System. They are more concerned with light-weight sensing applications and do not consider the mapping of user data-parallel programs.

This thesis assumes no additional hardware support and employs a simple but effective cache-flush policy. The compiler inserts system calls to flush the caches.

3.5 Power/Energy

This section takes a look at related work in the area of Power and Energy.

3.5.1 Power

First, Dark silicon issue is introduced. Dark Silicon leads to power constraints/budgets in the future, related work in power constraints are discussed next. The final paragraph discusses related work in Power Measurement.

Dark Silicon

The increase in power-densities has led to thermal safety issues in processor chips. Hence, all portions of a chip cannot be simultaneously powered on. The portions which cannot be powered on are called Dark Silicon. Dark silicon [45] suggests that as it will not be possible to simultaneously power-up all the cores on a chip, cores should be specialised to best utilise the available power. Shafique et.al[101] surveys challenges posed by dark silicon in hardware/software co-design.

Power Constraints

Yihia et.al[56] recommends a compiler based approach for meeting power constraints in superscalar processors. The compiler inserts instructions to provide hints to the hardware about high and low power code regions.

Prior work[59][81] for homogeneous multi-cores and GPUs[71] have looked at

meeting power constraints using DVFS. Machine Learning methods[19] have been used for selecting hardware configurations. This method predicts which processor to use, either CPU or GPU and the DVFS setting for it. It does not partition the workload across the processors. The *Core-type* policy presented in Chapter 6 is similar to this policy.

Wang et.al[121] partitions power and workload across CPU-GPU OpenCL based systems. They use a dynamic online method to determine the most efficient partitioning for the power budget. However, multiple executions are needed at runtime to find the optimal partitioning for a budget, while the work presented in this thesis runs the program once on each processor offline in the *Profile-directed* approach. They have also restricted their study to a single budget for maximising the throughput, whereas this thesis performs a detailed study over the entire power budget range.

Li.et.al[74] has suggested methods for optimising power consumption given performance constraints. This thesis concentrates on maximising the performance given power constraints.

Pagani et.al[88] argues that using a single power budget can result in performance losses. They advocate a budget called Thermal Safe Power (TSP) which varies based on the number of cores active. The work in this thesis provides the ability to change the power budget.

The presence of power budget emphasises the need for power measurement and control. Accurate power measurement methods are also needed to measure energy and efficiency. Related work in this area is presented next.

Power Measurement

Intel[99] provides a framework called RAPL for making power measurements. AMD provides APM (Advanced Power Management) for monitoring and measuring power. PAPI[123] is a framework for measuring power and performance. Various power management techniques are detailed in [22]. [16] describes tools and techniques for power measurement. Kambadur et.al[63] experiments with various techniques for power reduction across the stack. Energy Management Techniques in modern mobile handsets are surveyed in [114]. An empirical model for a low power mobile platform is presented in [119]. A case study for energy efficiency and performance in heterogeneous processors is presented in [83]. [46] analyses chip power and performance for various generations of architecture and a big set of benchmarks.

The power measurement method used in this thesis is described in Section 2.6.2.

3.5.2 Energy Efficiency

The next two paragraphs discuss related work in the area of two techniques for energy efficiency, i.e., heterogeneity and DVFS. The final paragraph of this section talks about energy performance trade-offs.

Heterogeneity

Research[28][30] has also been conducted for mapping applications to heterogeneous MPSoCs for energy efficiency. These work do not consider DVFS levels of processors. Chandramohan et.al[31] partitions programs based on throughput and energy efficiency of processors. This work does not consider power budgets when partitioning programs.

Much research in heterogeneous processors has focused on CPU-GPU systems. Wang et.al[120] distributes work across CPU-GPU system to maximise power efficiency. Paul et.al[89] presents a coordinated approach for maximising power/energy efficiency in CPU-GPU heterogeneous systems. These work focus on improving the energy/performance efficiency but do not take into consideration a power budget. Prakash et.al[93] presents a work for energy minimisation in heterogeneous cores with OpenCL. They also target the Exynos 5422 platform but uses OpenCL on both the CPU and the GPU. Using OpenCL on the CPU introduces overhead which our method avoids by using the Hybrid SPMD model. In this thesis, pthreads are used on the CPUs and OpenCL on the GPUs.

Kundu et.al[67] discusses how performance/energy can be improved by partitioning work between an ARM processor and DSP. Experiments are performed using OMAP3 SoC on a beagleboard for a single application. The work presented in this thesis has looked at applications with and without synchronisation, and also the variability in the best partitions.

Reflex [76] performs simple tasks on low-power micro-controllers. To simplify programming they use a Distributed Shared Memory. They report code simplification and power consumption improvements on an OMAP4 SoC and a custom platform for sensing applications. This thesis differs in the use of static partitioning, barriers for improving programmability and use of data-parallel programs.

The work in [26] considers single-ISA AMP processors and shows improvement in performance and energy using synergies between the heterogeneous processors and virtual machine service. The work presented in this thesis differs, it uses multiple-ISA architecture and considers static partitioning of data-parallel programs.

Some work has considered offloading code sections belonging to a particular domain to the remote co-processor for efficiency. For e.g., the research work in [102] offloaded machine learning to the DSP co-processor. This thesis treats all processors as first class and partitions work among these processors to attain the best performance, efficiency and meet power budgets.

DVFS

DVFS techniques are popular for obtaining good power/energy efficiency. Analytical[126], Compiler[127], Control-theoretic[116] and Online [40] methods have been proposed for utilising DVFS. Pack & cap[37] selects DVFS and thread groups using logistic regression. All these methods target homogeneous cores only. Ionnau et.al[58] proposes a method for DVFS power management where frequency and voltage are switched at runtime according to phases.

In this thesis, DVFS is used along with heterogeneity for energy efficiency and for meeting the power budget.

Energy Performance trade-off

Yuki et.al [130] states that for most machines of today, running as fast as possible is best regarding energy and this observation holds for all machines where the dynamic energy is comparable to the static energy. This work was conducted for homogeneous processors. This is similar to our observation also, when measuring energy with a large amount of idle energy. However, if the idle energy is much lower than dynamic energy and there are heterogeneous processors, then heterogeneity can be used to obtain energy efficiency.

3.6 Summary

This Chapter presented a survey and discussion of the related work.

The next Chapter begins the core of this thesis and presents the first contribution. It details a design space exploration in the mapping of Data-parallel programs to heterogeneous MPSoCs.

Chapter 4

Partitioning Data-parallel Programs for Heterogeneous MPSoCs

This Chapter presents a design space exploration in the partitioning of data-parallel programs for the OMAP4 SoC. Straightforward techniques for partitioning data-parallel programs for performance and energy are also presented.

Section 4.1 introduces the work in this Chapter. Section 4.2 presents the motivation for the work. Section 4.3 presents the programming model and some background. Section 4.4 describes the partitioning policies. Section 4.5 and Section 4.6 provides metrics used for experiments and the setup. Section 4.7 and Section 4.8 present experimental results. Finally, Section 4.9 concludes the Chapter.

4.1 Introduction

Many of the embedded systems available today are based on MPSoCs. The processors employed are diverse and consist of devices such as CPUs, DSPs, micro controllers and GPUs. The OMAP[6] from TI, Snapdragon[13] from Qualcomm and Tegra[14] from NVIDIA are examples of popular MPSoCs in the mobile embedded systems market.

In general-purpose computing, there has also been a recent cautious move to heterogeneous many-core systems where diversity is currently less extreme. Typically, GPGPUs are used as programmable accelerators and there is variation in CPU micro-architecture while maintaining the same ISA e.g., ARM's big.LITTLE [18].

In fact, energy and power have been the driving force behind this gradual convergence of the two communities. The embedded world has lead in system diversity and it is likely that tomorrow's general-purpose heterogeneous many-core platforms will resemble the MPSoCs of today.

The design of embedded MPSoCs has been largely application driven with specialised units such as ASICs and DSPs targeting media decoding or digital signal

applications. Embedded applications developers have traditionally been expert programmers where the cost of porting and tuning can be amortised across many shipped devices. Because of this, the programming models supported by MPSoCs typically tend to be application driven and change from one platform or generation to the next. As embedded MPSoCs become used for different purposes than they were initially designed for and enter the mainstream, the complexity of programming them becomes an increasingly important issue. Future platforms will have higher levels of parallelism and will contain more specialised cores. What we would like is to ease the programming burden needed to exploit increasing levels of diversity.

This Chapter proposes the use of an SPMD model of computation for data-parallel programs and explores the mapping of such applications to the highly heterogeneous TI OMAP platform. This platform has different memory domains, has multiple operating systems, specialised processors with different ISAs and local compilers and rudimentary systems software support. It is a difficult programming target.

We first develop a framework that allows an SPMD compiler model to map down to such a hardware platform. We then explore the mapping space and show that the best mapping depends on the metric involved. By using a highly accurate energy measurement, we show that the best mapping varies depending on whether time or energy is the optimisation goal. It also depends on the level of code optimisation available and whether power gating is available. It is frequently believed [76] that running the fastest is best for energy. Since heterogeneous multicores provide opportunities for energy-delay tradeoffs, we show in fact that this property no longer holds. We develop a partitioning approach that, when applied to a set of benchmarks, gives, on average, a 2.2x speedup over sequential execution time and reduces energy consumption by 1.45x.

This Chapter makes the following contributions:

- Develops a framework to map data parallel programs to the OMAP platform.
- Provides a detailed energy evaluation methodology.
- Explores a partitioning design space and shows its dependency on optimisation goal: energy vs time.
- Develops and evaluates a partitioning approach that is within 10% of the best across benchmarks and optimisation criteria.

```
for(int k=0;k<N;k++)
{
  #pragma omp parallel for
  for(int i=0;i<N;i++)
    for(int j=0;j<N;j++)
      path[i][j]=path[i][j]<path[i][k]+path[k][j]?
                 path[i][j]:path[i][k]+path[k][j];
}
```

(a) OpenMP FloydWarshall

```
for(int k=0;k<N;k++)
{
  call_barrier(1);
  for(int i=start_indx;i<end_indx;i++)
    for(int j=0;j<N;j++)
      path[i][j]=path[i][j]<path[i][k]+path[k][j]?
                 path[i][j]:path[i][k]+path[k][j];
}
```

(b) SPMD FloydWarshall

Figure 4.1: Programming Model

4.2 Motivation

MPSoCs have the potential for high performance but are hard programming targets. Consider Figure 4.1a which shows a simple data parallel program, floydwarshall, which we wish to map to the OMAP4430 platform shown in Figure 4.5, a typical MPSoC platform. The processors have ifferent ISAs, do not share a common address space and even support different operating systems.

Currently, due to programming complexity, programmers are discouraged from exploiting the full potential of the platform; instead run the program on the A9 core or possibly the DSP.

We, however, are interested in using all of the processing elements based on a data partitioning approach. In Figure 4.1a, the floydwarshall program has a parallel middle loop which is equivalent to partitioning the path array on the first index, e.g., by rows. Applying this partitioning to the original program gives the code as shown in Figure 4.1b.

This is the local code that each processor will run. The local loop bounds of i are restricted so that only local data is written to. This means that the cores run the same code with different $start\_indx_p$ and end_indx_p depending on the amount of data allocated to them. Here $p \in \{A9_1, A9_2, M3_1, M3_2, DSP\}$. A barrier synchronisation is inserted in line 3 due to the cross processor flow dependence from path[k][j] to path[i][j].

Once we have generated the local programs, the key issue is determining the amount of data to be allocated to each processor and hence its workload. Figure 4.2 and Figure 4.3 show the runtime and energy for various partitioning policies relative to sequential execution on the A9. A naive partitioning approach, *hom*, partitions data uniformly across all cores, *freq* partitions across cores in proportion to their clock frequency, *iter* is the partitioning scheme developed in this Chapter and *best* is the best partitioning found by exhaustive search. The *best* policy is not realistic but provides a useful upper-bound on performance.

4.2.1 Time

If the programmer were to select the DSP or even the M3, then this will lead to slowdown relative to the A9. In Figure 4.2, the DSP is 5x times slower than just running the code on the A9. Surprisingly, running on the M3 is faster than the DSP but still 4x times slower than on the A9. If we now consider data partitioning across all processors, then *hom* gives the worst performance, 2x times slower than the A9. The *freq*

approach is a little better but still slower. Our *iter* scheme is able to nearly achieve the 2x speedup available from exhaustive search, the *best* scheme.

4.2.2 Energy

If we now look at energy, a different pattern emerges. The programmer selecting the DSP as their target results in worse energy efficiency than the A9. On the other hand, the M3 would give 2x improvement, though this is at the cost of much slower execution, The *hom* and *freq* policies continue to give poor outcomes with 2x as much energy used as the A9 baselines. Our approach, however, achieves the same level of energy efficiency as the M3 and *best* schemes i.e., 2x less energy than the baseline. Thus, our approach is able to achieve significant improvements regardless of the metric of interest.

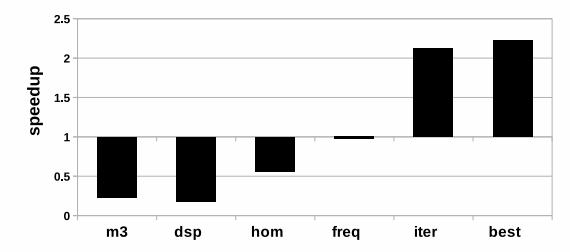


Figure 4.2: Execution time speedup relative to sequential execution on the A9. The M3 and DSP alone are significantly slower. Equal partitioning, hom and frequency based partitioning perform poorly. Selecting the right partition gives 2x speedup

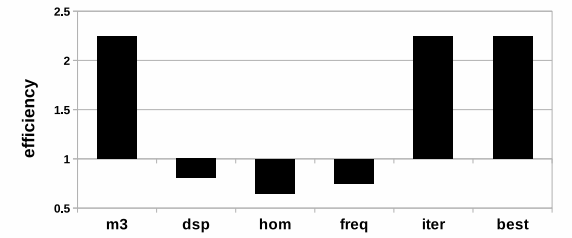


Figure 4.3: Energy relative to sequential execution on the A9. The M3 alone is significantly more energy efficient than the A9. The DSP alone is less efficient. Equal partitioning, hom and frequency based partitioning perform poorly. Selecting the right partition gives 2x improved energy performance

This example shows that, even when we overcome the difficulties of different processors and operating systems, determining the right partitioning is a difficult task for heterogeneous systems. Methods used in homogeneous systems such as uniform par-

titioning are ineffective. The next sections describe the SPMD model in more detail and describe how this can be mapped to the challenging OMAP platform.

4.3 Programming Model

In this Section, we describe how data parallel programs can be partitioned and mapped using an SPMD model of computation. This is followed by a description of the OMAP programming model and how we can map an SPMD model on to it.

4.3.1 Data Parallelism and SPMD

Data Parallelism

Throughout this Chapter, we focus on data parallel programs. These are programs where individual elements of a data structure can be independently computed. They are normally found in array based programs with corresponding parallel loops. Such programs are frequently found in embedded applications. We are not concerned in how the data parallelism is determined. This can either be performed by the programmer who inserts a parallel pragma or determined by automatic parallelisation. Instead, we focus on how this potential parallelism can be mapped to a heterogeneous system.

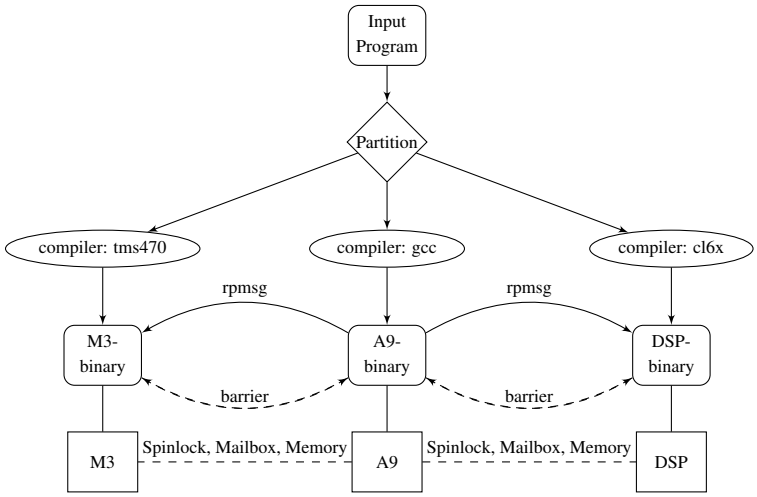


Figure 4.4: Partitioning a program

SPMD

The SPMD (single program, multiple data) model of parallelism is well known and used in a variety of settings. Unified Parallel C (UPC)[27], Co-Array Fortran[86], and Titanium[129] are a few examples. It consists of a set of parallel tasks which can be

either processes or threads, that run the same program but operate on independent sections of an array. The array is partitioned across processors such that each processor works on a separate array section. We consider the case when each task is implemented as a separate thread. The set of threads is fixed throughout the entire program execution and typically corresponds to the number of processing elements. The threads are forked off at the start of execution and only join at the end, unlike traditional fork/join parallelism. Synchronisation is inserted whenever there is a cross-processor dependence, which is achieved by the use of barriers.

Data Partitioning

The key issue is how to partition the data. This can be broken into 2 separate stages (i) which dimensions of the data to partition, (ii) how many elements of each dimension to allocate processor. The first stage has been considered by many researchers [17]. In this Chapter, we restrict ourselves to just 1 dimensional partitioning. We consider each dimension in turn and determine the most parallel dimension, i.e., partitioning along this dimension incurs the least amount of synchronisation. If there is more than one candidate, we select row partitioning as rows are contiguous in 'C'. The second stage depends on the performance and energy consumptions of the processor. We consider a number of different policies in Section 4.4.

Computation Partitioning

Once the data is partitioned, we need to determine the computation to be performed by each processor. We use the local write rule, each thread only writes to its local data, placing a constraint on the thread's local loop bounds. Figure 4.1b shows an example of such loop bounds. One important consequence of this mapping rule is that there are no remote-writes, and all output dependences are by definition within a processor.

Synchronisation

Synchronisation is needed whenever there is cross-processor dependence i.e., the source of the dependence is in a different thread to the sink. As there are no output-dependences, only flow and anti-dependences need be considered. Furthermore, if a read access is aligned on a partition to a write access, it is guaranteed to be local. In Figure 4.1b, the read of path[i][k] is local as it has the same reference on the partitioned first dimension [i], i.e., it is aligned. The reference path[k][j] is remote as it has a different reference

[k] on the partitioned dimension. Once we have the cross-processor dependence graph, barriers are inserted at the highest lexical level that covers the dependence. For more details on optimising barrier placement see [87].

4.3.2 The OMAP model

Hardware

The TI OMAP4430[7] is a typical mobile applications MPSoC consisting of various subsystems including general purpose processors, a programmable multimedia engine plus graphics and image accelerators. Figure 4.5 shows the components of the

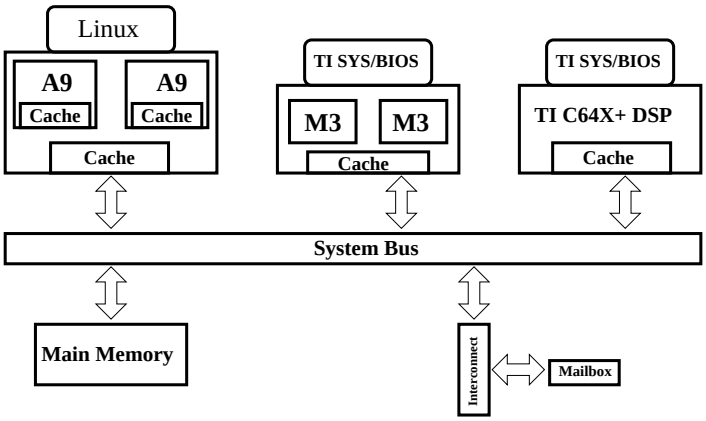


Figure 4.5: OMAP4 block diagram

OMAP4430 relevant to this Chapter. A dual core ARM Cortex-A9 provides general purpose computing. Both processors share a common L2 cache and address space. There are two smaller ARM Cortex-M3s available for smaller and lower power tasks. Both share a common L1 cache and are responsible for controlling the graphics and image accelerators. There is also distinct programmable multimedia engine based on a TI mini-C64X+ DSP. The PowerVR GPU cannot be currently programmed, and is not considered in our study.

Operating System

The ARM Cortex-A9 cores are configured to run as a Symmetric Multiprocessing system with Ubuntu Linux as the Operating System. In contrast, the M3s and DSPs run a TI RTOS called SYS/BIOS. The M3s are configured to run the RTOS in SMP fashion and hence the RTOS is also called SMP/BIOS. Programming the OMAP requires managing not only different address spaces but different operating systems.

The communication between the remote processors (M3s and DSP) and the A9s is managed using an Inter Processor Communication Protocol (IPC) called Syslink. This IPC framework was developed by TI to offload processor-intensive tasks to hardware accelerators/remote processors. The new version of Syslink is called Remote Processor Messaging (RPMsg)[1]. RPMsg works by sending and receiving messages through shared memory. The notification of messages is performed via a Mailbox interrupt mechanism.

Mapping applications to the OMAP therefore, currently, requires decomposing programs into tasks that fit the IPC model.

4.3.3 Mapping SPMD programs to OMAP

Figure 4.4 summarises the partitioning and mapping of an SPMD program onto the OMAP architecture. Each local program is compiled by the processor's host compiler. The resulting binaries are executed by each processor which synchronises via a spin-lock library. There are four main issues to consider, i) thread management ii) data partitioning iii) sharing memory and iv) synchronisation, each of which is described below.

Threads

We use the pthread library for parallel programming which offers a fine control over parallelism. One thread is created for each A9 core, one thread for the M3 and another for the DSP. There are in fact two M3 cores; this is handled in the M3 SYSBIOS where we create two threads for computation. The threads for M3 and DSP call remote procedure calls with addresses of the partitioned arrays as arguments. These remote procedure calls activate the remote processors, and they perform their computation.

Shared memory

The addresses of the processors are by default distinct. We use dmabuf[100] and some special properties of graphics memory addresses to overcome this. Memory is allocated in the Linux kernel side by creating a GEM buffer object. This can be memory mapped and used from the user space. Using the dmabuf API, physical addresses corresponding to this memory are obtained and propagated to the DSP or M3 using RPMsg/Syslink. This address will be in the tiler/dmm region which is setup with 1:1 physical to virtual mapping, so this can be used as the virtual address on the M3/DSP side.

Synchronisation

Synchronisation is performed using barriers. For implementing barriers, there are two requirements: 1) Mutual Exclusion 2) A common state variable to count the number of processors which reached the barrier. In OMAP, mutual exclusion is achieved using a hardware spinlock. On the A9 CPU side, we built a linux kernel driver to access the hardware spinlock and provided a user space interface. On the M3 and DSP side, the SYSBIOS RTOS provides access to the OMAP spinlock. The common state variable is implemented by using uncached shared memory. The barrier is made reverse sensing so that it can repeatedly be used.

A9 and M3 have two cores and hence besides the interprocessor barrier described in the previous paragraph, intraprocessor barriers are also implemented. Since, the A9 processor runs pthreads; pthread barriers are used. For the M3 processor, the barrier is constructed using semaphores provided by the RTOS and global variables.

4.3.4 Code Generation

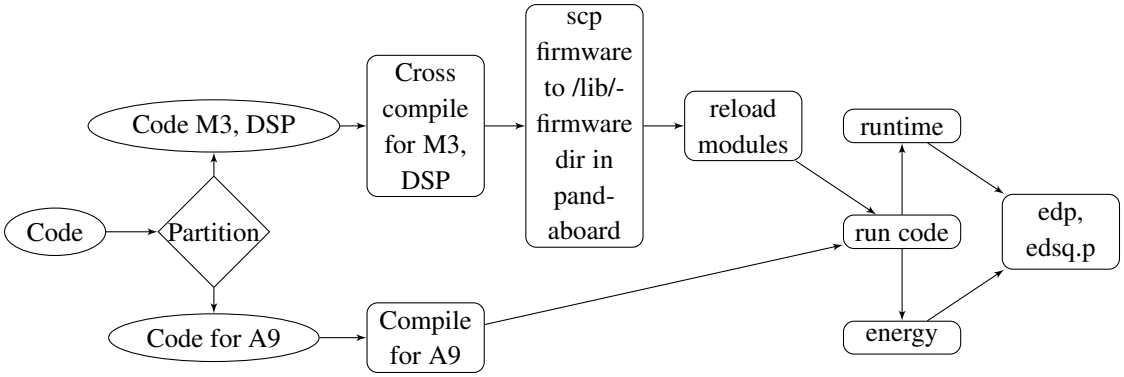


Figure 4.6: Code generation and runtime

Once the partitioning is decided, code is generated for the A9, M3 and DSP processors. We compile with gcc for A9s. The A9 code initiates the threads for all the processors. For the M3 and DSP a cross compilation is done using tools available as part of TI Code Composer Studio. Details of the compilers used are in Table 4.1. For these remote processors, the program is compiled along with the SYSBIOS operating system and made into binary firmware called *tesla-dsp.xe64T* and *ducati-m3-core0.xem3*. These binaries are placed in the directory */lib/firmware/* on the Linux machine. Some modules are reloaded to boot this firmware on the M3 and DSP processors. During this process, some shared memory location is agreed upon for exchanging messages

between A9 and the remote processors (M3, DSP) for the Syslink protocol. When the program is run, connections are opened to the A9 and M3 processors and the threads for M3 and DSP make remote procedure calls to execute the code on those processors.

Processor	Cortex A9	Cortex M3	C64X+ DSP
Vendor	ARM	ARM	TI
Frequency	1GHz	200 MHz	466 MHz
Compiler	gcc 4.6.3	CCS tms470	CCS cl6x
OS	Ubuntu 12.04-3	SMP/BIOS	SYS/BIOS

Table 4.1: Frequency and Compiler details

Runtime

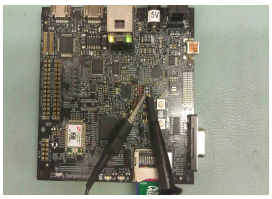
At runtime each program performs the following steps: 1) Allocate buffers for the arrays that have to be partitioned in the program with dmabuf. 2) Open connection to remote processors with Syslink. 3) Attach the allocated buffers to the remote processors and find physical addresses for these buffers so that they can be passed on to the remote processors. 4) Create threads for A9, M3 and DSP. While the A9 threads do the work locally, M3 and DSP threads will use Syslink to make remote procedure calls so that the work will be done on the remote processors. The remote procedures do cache synchronisation before and after execution. 5) Wait for threads to join. 6) Detach buffers from the remote processors. 7) Close connection to the remote processors with Syslink. 8) Deallocate the buffers.

Figure 4.6 shows the complete code-generation and runtime workflow of our approach.

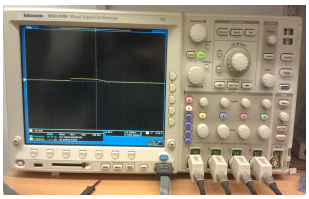
4.4 Partitioning Policies

In this section we look at four partitioning policies that are used in our experiments. Partitioning policies divide the data among all the cores for processing. The intuition behind these partitioning policies are provided. We also give the formulas that are used to compute the size of the partitions. In these formulas **part** is the function which computes the size of partition of a core.

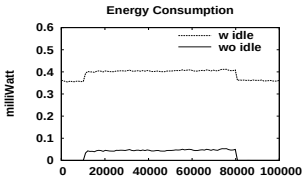
1. **hom** This is the partitioning we would have intuitively used on a homogeneous multicore processor. Partitioning is uniformly performed for all the cores. We



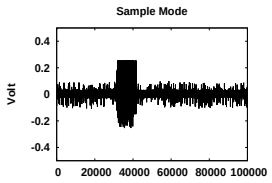
(a) pandaboard



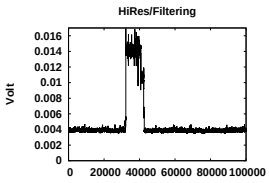
(b) oscilloscope



(c) with vs. without idle



(d) sample



(e) filtering

Figure 4.7: Board and Oscilloscope

have 5 cores in the OMAP4430 SoC viz. two A9s, two M3s and one DSP. So if the data-size is *size* then we assign all the cores the same partition as follows.

$$part(a9) = part(m3) = part(dsp) = \frac{size}{5} \tag{4.1}$$

2. **freq** This method tries to capture the heterogeneity by using the clock frequency of the cores to guide the partitioning decision. The clock frequencies of each processor is given in Table 4.1. Each core is assigned a partition which is in proportion to its frequency compared to the others. Hence in this partitioning method the A9 cores always gets the largest share followed by the DSP and the M3 cores. The intuition behind this method is that faster clock frequencies leads to faster execution and hence lower runtime and energy. The partitions for each core is made as per the following formula. In the formula, $proc \in A9_1, A9_2, M3_1, M3_2, DSP$, $core\_freq$ is the frequency of *core*, and 2866 is the sum of the frequencies of all the cores.

$$part(core) = (size) * \frac{core\_freq}{2866} \tag{4.2}$$

3. **unip** We know that certain types of programs are better suited to some processors than others. With the *unip* policy, we run the program sequentially on each core separately and use the information gained from that to guide the partition decision. The information gained is different for runtime and one of the energy cases. For runtime, we first determine the throughput of each core defined as

the ratio of the amount of computation to the time taken for each core. This is roughly analogous to FLOPS. Once we have found each core's throughput, a simple way to partition is to assign data partitions to each core in proportion to its throughput . If m is the throughput of M3 relative to A9 and n is the throughput of DSP relative to the A9 and *size* is the size of the total work to be partitioned, the partitions for the A9, M3 and DSP are given by the following formula.

$$\begin{aligned} part(a9) &= \frac{size}{2+2*m+n} \\ part(m3) &= m*partition(a9) \\ part(dsp) &= n*partition(a9) \end{aligned} \tag{4.3}$$

Using this partitioning each core will roughly finish at the same time, reducing load imbalance and execution time. This approach comes at the cost of a single run per processor.

When considering idle energy, we employ the same method as for runtime since running faster will ideally lead to lower idle power consumption. When there is no idle power then running fast is no longer important. In this method, we allocate all the work to the most efficient processor, the one which consumes the least energy for 1 unit of work.

4. **iter** This is the policy we propose in this Chapter. It is a simple extension to the *unip* policy. We first use *unip* to determine a starting partition. We then evaluate this partition on each processor in turn. This will give a new throughput value, as execution time is not always linear with respect to the amount of work. This approach is applied iteratively till the best solution is found. It was experimentally observed that for all the benchmarks, the best solution can be arrived at with a maximum of 3 iterations. Due to the three iterations, this method is more expensive than *u*nip.

5. **best** This is the best partitioning policy which gives the best possible partitioning for various optimising criteria. This policy is implemented by performing a design space exploration for runtime, energy and EDP. Since an exhaustive exploration of all the partitions for all the benchmarks will have too many points to measure, we sample at discrete points which are multiples of a fixed quantity that varies with benchmarks. This provides a useful upper bound for our evaluation.

4.5 Metrics

We are particularly interested in how mapping is effected by different metrics such as time and energy and how to accurately measure these.

4.5.1 Energy

Energy is the sum of power over time. Measuring power and hence energy is non-trivial. This section describes in detail the methodology used to accurately measure power and discusses how it should be accounted for.

Power consumption is found by measuring the current consumed by the processor and multiplying by the supply voltage. The OMAP system is mounted on the Pandaboard, which is driven by a supply voltage of 4.2V. However, we cannot measure the current consumption directly from the whole board supply since it is very noisy and hence is not accurate enough to distinguish between partition decisions. Also, we would like to know the individual consumption of each processor. Hence, we measure each processor's power consumption separately. We also have to measure the power consumed by memory. For this, we concentrate on five power rails that supply power i.e., VCORE1 for the A9, VCORE2 for the DSP, VCORE3 for the M3, V1V29 for memory interface and VMEM for the memory. To measure the current, we find a resistor in the path of these power rails and measure the voltage drop across the resistor and divide by the resistance. Since the resistors are attached to the board and are very small, pins are soldered to the resistors so that readings can be taken easily. This method is described in detail by Jos [115].

This approach gives a good insight into the power consumption. However in programs with phases, synchronisation and heterogeneous loops the power consumption varies during the execution of an application. Hence, we need to continuously monitor the voltage. For this purpose an oscilloscope is used, and readings are taken every millisecond. Figure 4.7a shows the pandaboard with probes attached to one pair of pins to take energy measurements. Figure 4.7b shows the oscilloscope with a measurement of the processor to which the probes are attached in Figure 4.7a. The resistance is measured using a digital multimeter. For our experiments, a Tektronix MSO4104 oscilloscope and Agilent 34410A digital Multimeter is used. Figure 4.7d shows a voltage reading from the oscilloscope in the regular sampling mode for the A9 processor when running the floydwarshall benchmark. As can be seen, there is significant noise in the measurement. This can be overcome using either post-filtering by software or the High Resolution measurement mode of the oscilloscope. Figure 4.7e shows the same mea-

surement after filtering. All the rails are measured separately and summed up to find the total power/energy consumption. We measure energy consumption in two different ways, viz. with and without idle energy.

4.5.2 Two energy measures

In this section, first idle energy is defined and then the two different energy measures are introduced.

Idle Energy

Even when the processor is not performing any work, it consumes some energy, this energy is referred to as the idle energy. This energy can be measured when the processor is not performing any work using the measurement method described earlier in this section.

With idle energy

This is the energy that is obtained directly during measurement when the processor is performing some work. The energy measured is composed of two quantities, i) the idle energy and ii) the dynamic energy. If there is no idle energy, energy consumption could be minimised by assigning the entire work to the processor which is most energy efficient. When idle energy measure is considered, there are two factors to optimise, i) minimise the runtime to reduce the idle energy consumption ii) allocate more work on the most energy efficient processor to reduce dynamic energy consumption.

Without idle energy

This is computed by first obtaining with idle energy measure and then removing the idle energy consumption. Using this energy measurement is important because the idle energy of a processor is not just the static energy but also the energy consumed by other parts of the board. Measuring without idle energy can give a fairer reflection of each component's energy contribution. For example, VCORE3 (M3's power supply) provides power not only to the M3 but also to GPIO, UART, L3 interconnect, etc. Hence, the M3 has a higher idle power allocated to it than other processors and hence there is a dissipation of energy even when the system is idle but switched ON. This will result in an unfair comparison since other processors have a lower idle energy allocated to them as they have less additional devices on their power rails. Figure 4.7c shows the energy consumption of the M3 processor with and without idle energy when the

floydwarshall benchmark is run entirely on the M3 processor. The dashed line is with idle energy, and the solid line is without idle energy. In both cases, the processor starts in the idle state, and when the program starts running, we see a transition to a higher state of energy consumption. In the without idle energy case, we can see that when the processor is in the idle state, the idle energy is almost zero. But in the with idle energy case, there is almost 0.3milliWatt power dissipation extra.

Which energy metric to use?

As we are physically unable to account for energy used by additional devices on some individual power lines, we present results for both with and without idle energy in the remainder of this Chapter.

4.5.3 Runtime and EDP

Runtime is measured separately for each core and for the entire system. Runtime is measured by using the *gettimeofday* system call. Calls are made to this function before and after the execution and the runtime is the difference of the times. All the runtime measurements are made on the A9 side. EDP is measured by computing the product of Runtime and Energy.

4.6 Experimental Setup

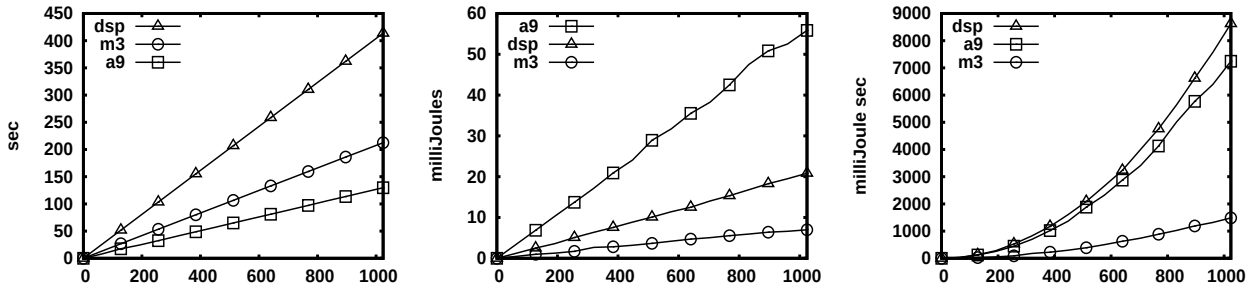
Here we briefly describe the benchmarks used throughout the evaluation. Details of the hardware platform have been provided in Section 4.3.2

4.6.1 Benchmarks

The benchmarks that are used for experiments were presented in Table 2.2. The Table shows the name of the benchmark, short name used in the diagrams, the suite it belongs to and the default size of the arrays that are partitioned in these benchmarks. The benchmarks are taken from DSPstone[113], UTDSP[70] and polybench[92]. Benchmarks are parallelised and then used for experiments. All the benchmarks used are integer based. While ARM Cortex-A9 CPU supports floating point, Cortex M3 microcontroller does not support floating points and the TI C64x+ DSP supports fixed point only. Hence in the latter two cases, floating point is emulated.

4.7 Matrix Multiplication Case Study

In this section, we study one benchmark, Matrix Multiplication (*mxm*), in detail before presenting the results for all benchmarks in Section 4.8. We first compare the



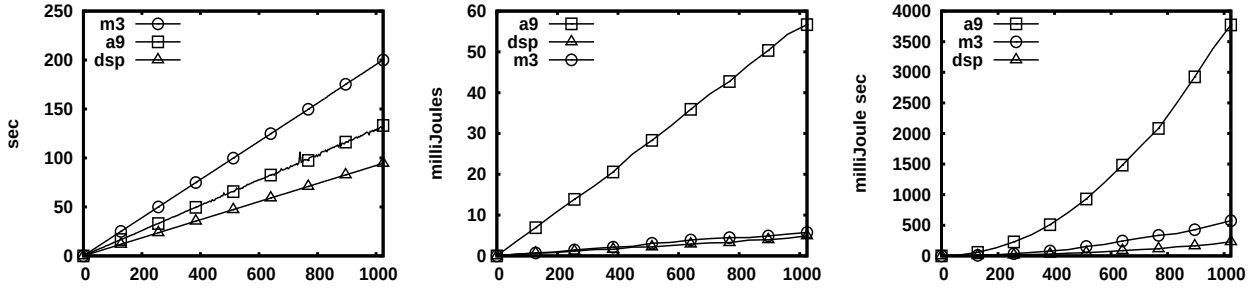
(a) Runtime (b) Energy wo.static (c) EDP wo.static

Figure 4.8: Unoptimised Matrix Multiplication Benchmark: runtime, energy, edp without idle energy

performance of *mxm* on each distinct processor before evaluating partitioning across processors.

4.7.1 Individual Processor

Figures 4.8 to 4.10 show the performance of *mxm* for runtime, energy and EDP for varying scenarios, in each case N=1024



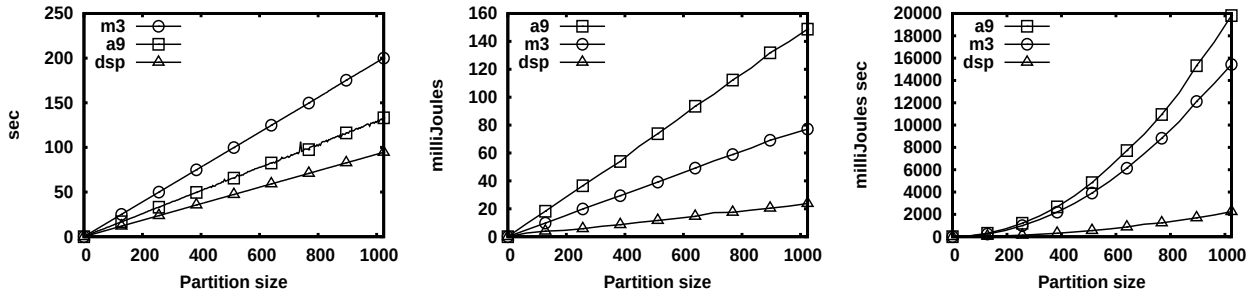
(a) Runtime Opti (b) Energy wo.static (c) EDP wo.static

Figure 4.9: Optimised Matrix Multiplication Benchmark: runtime, energy, edp without idle energy

Default optimisation level

Initially, we use the default compiler flags for each processor's compiler: O2 for gcc and O3 for the TI compilers. The results are shown in Figure 4.8(a). Here the A9 is clearly the fastest processor followed, surprisingly, by the M3 and then the DSP. For energy without idle power, the relative performance changes. The M3 is the best processor followed by the DSP and then the A9 as shown in Figure 4.8b. Regarding EDP, the M3 is the best processor followed by the A9 and DSP as shown in Figure 4.8c.

This shows that processors behave differently for different optimising criteria. While



(a) Runtime Opti (b) Energy w.static (c) EDP .static

Figure 4.10: Optimised Matrix Multiplication Benchmark: runtime, energy, edp with idle energy

the A9 is the fastest, and it is also the worst for energy.

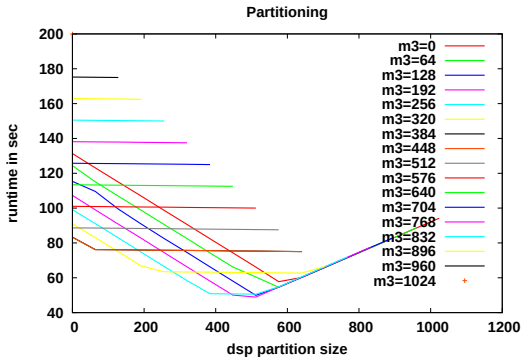
Effect of Optimisations

The runtime performance of the DSP is surprising since it should perform well for *mxm*. The compile logs indicated that the loop was not software pipelined due to memory dependences. The compiler thinks that pointers to array parameters can alias and generated a conservative schedule. We, therefore, added the *restrict* keyword to these pointers which significantly improved the performance of the DSP. No significant improvement was observed with the M3 or A9 since they lack the hardware to gain from this information. Figure 4.9a shows the performance after optimisation. Now the DSP is the fastest followed by the A9 and M3. For energy (without idle), the DSP and M3 are very similar while the A9 is inefficient as can be seen from Figure 4.9b. Overall for EDP, DSP is now the best processor rather than the worst followed by M3 and A9 as shown in Figure 4.9c.

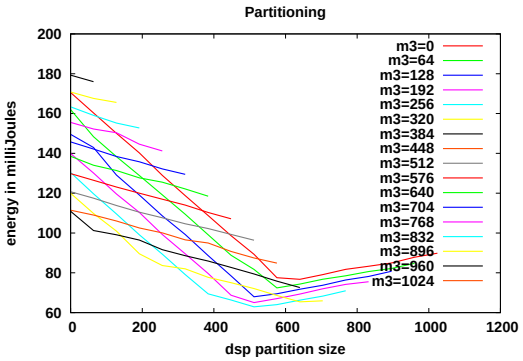
This shows that the best processor performance depends on backend compiler optimisations.

With Idle Energy

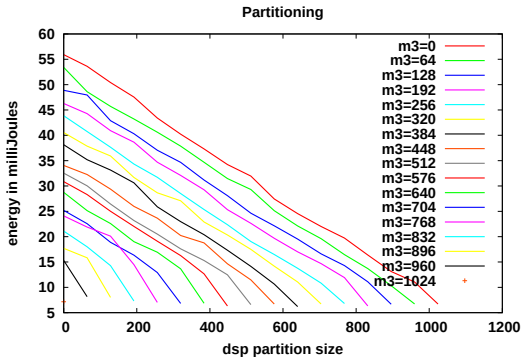
The above results do not consider idle energy and focus only on a processor's dynamic energy consumption. In reality, processors dissipate energy even when they are idle and this has to be factored in during measurement. Figure 4.10 shows the results including idle energy. Here we find that the M3 has a significantly larger amount of idle energy than before. The DSP has the least energy followed by the M3 and A9 as shown in Figure 4.10b. Runtime is unaffected by how we calculate energy, so overall for EDP, the DSP is the best processor followed by the M3 and A9 as shown in



(a) Time: Best DSP=512, M3=192, A9=320



(b) With idle. Best DSP=512, M3=256, A9=256



(c) Without idle. Best DSP=640, M3=384, A9=0

Figure 4.11: Matmul exploration: Best partitions give more than double runtime and energy (with idle) performance relative to just using A9. If idle energy is discarded the improvement is more than 8 fold.

Figure 4.10c.

This shows that when determining which processor is the best for energy, it also depends on how energy is measured. If only dynamic energy is considered, then the M3 and DSP are competitive, but if idle energy is also considered then the DSP is the best by a large margin.

4.7.2 Partitioning

This section considers the platform performance when *mxm* is partitioned across all cores. It first explores the design space of different partition sizes and then explores the power-runtime trade-off when partitioning.

Exploration

The results of a detailed exploration results are graphically given in Figure 4.11 for the input size N=1024 and DSP enabled backend optimisations. In each graph, the x-axis

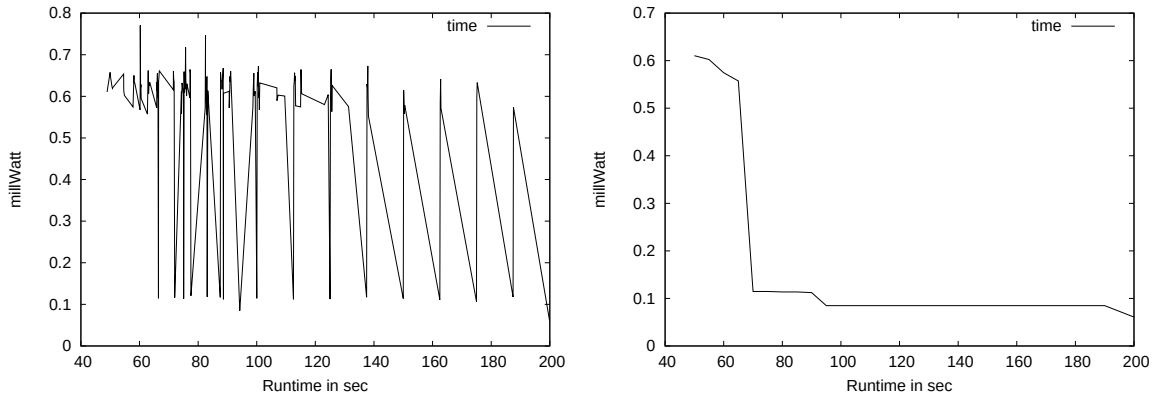
denotes the amount of data allocated to DSP and each curve represents the amount of data allocated to the M3. The y-axis represents runtime, energy with idle power and energy without idle power respectively. In Figure 4.11a, we see the best runtime of 48.92s is achieved when half the data (512 rows) is allocated to the DSP and the M3 has 192 rows allocated to it. The A9 has the remaining 320 rows allocated to it. If all the data were to be allocated to the A9, i.e. M3=DSP=0, then the 2 threads of the A9 would take 131s - more than twice as long. In the case of energy with idle power, we see a similarly shaped trade-off graph. Again, the DSP has 512 rows allocated to it. This time, the best energy result of 63 mJ is obtained when the M3 has slightly more work allocated to it, 256 rows. Allocating all work to the A9 would increase the energy usage more than double to 170 mJ. In the case of without idle energy we see a different trend. As we do not account for increasing idle energy, we see a monotonic behaviour. The best partition is when the DSP has the most work allocated to it, 640 rows, the remainder being allocated to the M3. No work is scheduled to the A9 as it is too energy hungry. This results in just 6 mJ compared to massive 55 mJ if the default approach of allocating to the A9 is employed.

Power Runtime Tradeoff

Figure 4.12a shows the tradeoff between power and runtime, an important consideration in thermally constrained mobile devices. The figure shows considerable oscillations between 0.6-0.76milliWatt and 0.1-0.2milliWatt. The 0.6-0.76milliWatt range represents those configurations which use the A9 processor and those in the range 0.1-0.2 mW have the A9 switched off. From this figure, we can see that power and runtime are not simply correlated. If there are power constraints, we should avoid configurations with high power consumption. Figure 4.12b shows the tradeoff-curve Pareto frontier. When allowing increased runtime, we keep the best power configuration seen so far. When the runtime is around 65 seconds, there is a configuration which has only the M3 and DSP running and hence results in a low power state of around 0.1mW. The next lowest, around the 95 second mark, is when the DSP is used alone. Finally, the configuration with the lowest power consumption is when the M3 alone is used resulting in a runtime of around 200 seconds. This graph gives us the best configuration for power for a given runtime budget.

4.8 Partitioning Policy Results

This section evaluates the various partitioning policies described in Section 4.4 across the benchmark suite for different optimisation criteria.



(a) Power-Time Tradeoff

(b) Power-Time Tradeoff Pareto

Figure 4.12: power runtime tradeoff

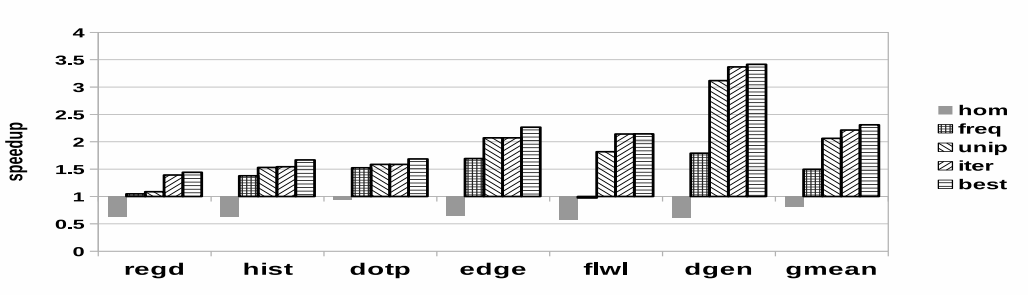


Figure 4.13: Results of partitioning policies for runtime: All policies except *hom* give performance improvements. *iter* is the best policy with 2.2x speedup just short of the 2.3x achievable by exhaustive search of *best*. *unip* is competitive with *iter* on most benchmarks.

4.8.1 Runtime

Figure 4.13 shows the runtime performance of the partitioning decisions as speedups over sequential execution on the A9. The uniform allocation policy *hom* performs uniformly poor across all benchmarks. It slows down *dotp* the least but gives an average slowdown of 0.7. The frequency based approach *freq* is better again, performing relatively well on *dotp*. However, it slows down on *flwl* and has an average speedup of just 1.5x. The other schemes that require runs on each processor perform better. The *unip* policy gives an average 2.1x speedup across benchmarks and is competitive with our *iter* approach, particularly on `edge`. Overall, *iter* is the best policy with 2.2x speedup just short of the 2.3x achievable by trying all partitions. It performs particularly well relative to the other policies on *regd*.

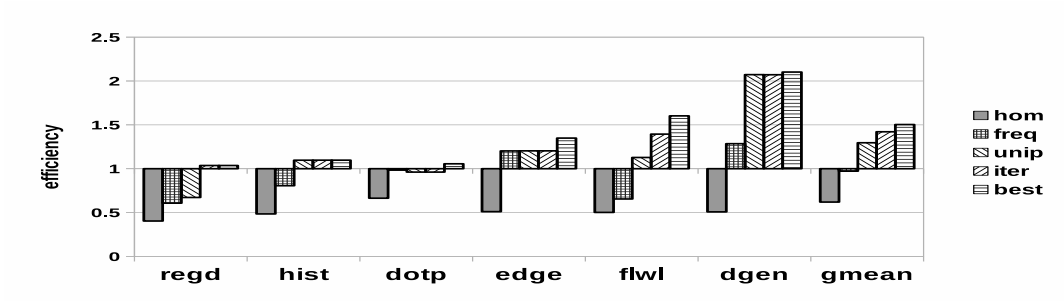


Figure 4.14: Results of partitioning policies for metric energy with idle. *unip* gives a slowdown on *regd* and is not competitive for *flwl* compared to *iter*. On average *unip* gives an overall energy improvement of 1.3x compared to 1.45x of the *iter* scheme. *iter* approaches the efficiency of the *best* policy. *hom* and *freq* policies are inefficient compared to the sequential execution on A9.

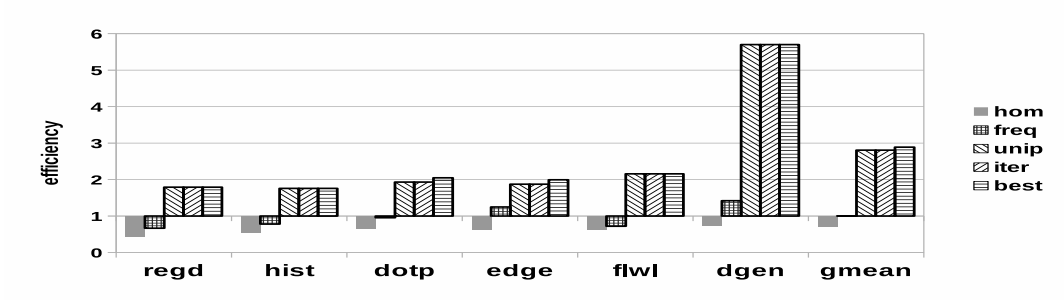


Figure 4.15: Results of partitioning policies for metric energy without idle. *iter* leads to almost a 3-fold energy improvement on average over sequential execution on A9 and is within 8% of *best*. Partitions of *hom* leads to degradations in all cases. On average *freq* gives no improvement at all.

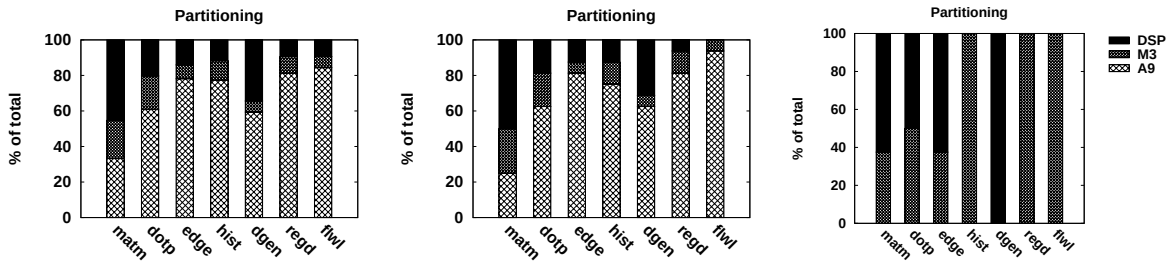
4.8.2 Energy

Here we examine energy usage with and without idle for each policy, across the benchmarks.

With Idle

When there is idle power, it is often best for the processors to run as fast as possible. Figure 4.14 shows the results for each policy.

The *hom* and *freq* policies give the same partition decisions here as when optimising for runtime. The *hom* policy is uniformly poor with an average 0.4x the energy efficiency of running sequentially on the A9. Compared to runtime performance, the *hom* policy performs, on average, even worse for energy. Overall, it is slightly less efficient than running sequentially on the A9. Both *unip* and *iter* approach the performance of



(a) time (b) with idle (c) without idle

Figure 4.16: Partitioning overview of all benchmarks for runtime, energy with and without idle. Partitioning is different for each benchmark. A9 gets the major share for runtime in all benchmarks except matm. But for energy without idle, A9 gets no share at all due to its high dynamic energy. The partitioning is similar for metrics runtime and energy with idle.

the *best* exhaustive scheme. However, *unip* again performs badly on *regd* leading to on overall energy improvement of 1.3x compared to 1.45x of the *iter* scheme.

Without Idle

When there is no idle power to consider, then, running fast is no longer important. Figure 4.15 shows the results for partitioning policies for metric energy without idle. As we can see from the figure, the results are poor for *hom* and *freq*. This happens because these policies allocate a significant amount of computation to the A9 processor which consumes a large amount of dynamic energy. Both *unip* and *iter* have almost identical performance; 0nly 0 to 8 percentage worse off than the best partition found by an exhaustive search *best*. On average, this leads to almost a 3-fold energy improvement over sequential execution over the A9.

4.8.3 Analysis of Results

In this section, we examine the partitioning decisions that lead to best performance.

The best partitions for each metric for runtime is plotted in Figure 4.16a. As can be seen from the figure, the best partitions are different for different benchmarks. In percentage terms, the data partition size on A9 varies from around 33% to 84%; the M3 has between 6% to 21% of the data allocated to it while the data allocated to the DSP varies from 9% to 45%.

Benchmarks such as *mxm* and *dgen* have three or more loop nests where there is little or no synchronisation. There is also significant instruction level parallelism,

and hence, the DSP performs well, resulting in a large computation allocation to the DSP. There is less parallelism available in benchmarks such as *flwl*, *regd* and *hist* due to dependences requiring synchronisation. The M3 receives the most data on the *dotp* benchmark. This is probably due to the other processors being less able to exploit either their better memory hierarchy or internal parallelism. It is the program's properties that dictate the partition size that should be allocated to each processor.

The best partitions for energy with idle are plotted in Figure 4.16b. As we can see, the with-idle partitioning is similar to the runtime partitioning with small changes in the contribution of M3.

The best partitions for energy without idle are plotted in Figure 4.16c. The partitions are completely different from the others. The A9 is excluded due to its high dynamic energy. While the entire data is allocated to the M3 for *regd*, *hist* and *flwl* benchmarks, it is completely allocated to the DSP for *dgen*. For the other benchmarks, it is a mix of M3 and DSP. While there is no idle energy considered, there is energy expended in the memory system and splitting between the M3 and DSP helps these benchmarks to run faster and hence result in lower memory energy consumption.

This shows that the best partitioning for each benchmark and each criterion is different. There is no one size fits all solution for all benchmarks/criteria and hence all the fixed partitioning policies fail to give satisfactory performance.

4.9 Summary

This Chapter presented a framework for partitioning data parallel programs for heterogeneous processors that address access to shared resources and synchronisation. It describes a method for accurately measuring runtime and energy consumption of programs and used this to evaluate different partitioning policies. We show from our design space exploration that the best partitions change with optimisation, benchmarks and optimisation criteria. This Chapter presented a simple partitioning approach that is within 10% of the best partitioning scheme across all optimisation criteria. On average, we achieve a 2.2x speedup and a 1.45X energy improvement.

This Chapter has shown that partitioning provides benefits in a heterogeneous platform like OMAP4. The next Chapter presents a Compiler Framework for automating the partitioning of data-parallel programs for heterogeneous multiprocessors.

Chapter 5

Compiler Framework

This Chapter presents a Compiler Framework for automatically partitioning data-parallel programs for Heterogeneous MPSoCs.

Section 5.1 introduces the work in this Chapter. Section 5.2 presents the motivation for the work. Section 5.3 presents the target architecture. Section 5.4 presents the programming model and Section 5.5 presents usage of the SPMD Model as an intermediate form. Section 5.6 describes the compiler algorithm. Section 5.7 and Section 5.8 describes implementation details of shared memory and the compiler. Section 5.9 describes the experimental setup and Section 5.10 presents experimental results. Finally, Section 5.11 concludes the Chapter.

5.1 Introduction

We continue to witness an ever-increasing use of embedded devices. The majority of these embedded devices are based on heterogeneous MPSoCs. Nvidia Tegra[14], Qualcomm Snapdragon [13], Samsung Exynos[3] and TI OMAP [6] are well-known examples of popular heterogeneous MPSoCs. These MPSoCs contain a variety of processors like CPUs, micro-controllers, DSPs and specialised accelerators. The heterogeneity of these systems helps in achieving good performance and energy efficiency. The micro-controllers are used to control sensors, while the specialised accelerators are targeted at graphics, image or video processing. This specialisation comes at a programming price. These processors have different instruction set architectures (ISA), clock frequencies and operating systems. Additionally, these systems do not support cache coherent shared memory. Programming heterogeneous devices is therefore inherently difficult, and shared memory programming used in general purpose computing cannot be directly used.

The standard way of accessing non-CPU cores is to use them as accelerators ac-

cessed via platform specific libraries. This style of programming requires the application to be first split into separate tasks that fit the library API. The main CPU then acts as a coordinator using a Master/Slave programming model. Although this approach means that the programmer does not have to deal with different operating systems and manage memory coherence, it is highly system specific and may introduce excessive data traffic. Furthermore, it requires the programmer to perform platform-specific low-level partitioning of work into parallel activities. While this approach may work for specialised applications, we show in Section 5.10 that it is not suitable for data parallel programs and leads to poor performance.

OpenMP[42] is a popular high-level parallel programming model and is widely supported on parallel hardware. Programmers need only express parallelism without concern as to the mapping and scheduling of work to the underlying platform. Due to the wide availability of OpenMP libraries, it also offers some degree of performance portability. OpenMP, however, is not directly supported on heterogeneous systems, due to lack of coherent shared memory and multiple operating systems. Exploiting heterogeneity is avoided, and only the CPUs are utilised by the OpenMP runtime.

In this Chapter, we present a compiler framework which bridges the gap between the high-level programming model of OpenMP and the heterogeneous resources of MPSoCs. It takes OpenMP programs and generates code which runs on all the processors. It delivers programming ease while exploiting heterogeneous resources. Our compiler is based on a Single Program Multiple Data model where data layout is explicitly determined and allows reasoning about memory coherence and synchronisation placement.

We apply our compiler to a set of benchmarks and evaluate its performance on the TI OMAP4 architecture. We compare it against using the existing OpenMP implementation available on the dual A9 CPUs and the Syslink interface provided by TI for the accelerators. We examine the impact of compiler-directed smart synchronisation placement and cache flushing. Across the benchmarks, on average, it gives a 2.75x speedup over using the low-level Syslink library approach. Furthermore, it gives a speedup of 1.38x and an improved energy efficiency of 1.4x over using the default OpenMP implementation on the 2 A9 cores alone.

This Chapter makes the following contributions:

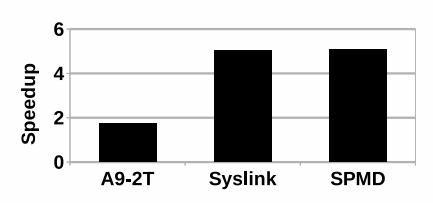
- Presents a Clang/LLVM based compiler approach to map OpenMP programs to MPSoCs.

```
#pragma omp parallel for
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
      C[i][j]+=A[i][k]*B[k][j] ;
```

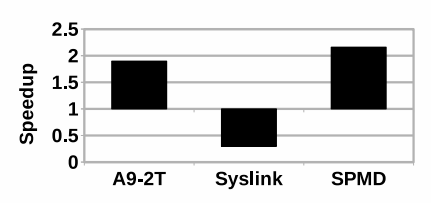
(a) OpenMP MatrixMultiplication

```
for(i=start_indx;i<end_indx;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
      C[i][j]+=A[i][k]*B[k][j] ;
```

(b) SPMD MatrixMultiplication



(c) Speedup MatrixMultiplication



(d) Speedup FloydWarshall

Figure 5.1: Motivational Example

- Demonstrates the importance of smart cache flushing for obtaining good performance.

- Presents significant performance improvements over existing approaches on the TI OMAP4.

5.2 Motivation

This section illustrates the programming complexity and performance available when using existing approaches and our scheme.

Consider the program in Figure 5.1a. It shows a simple matrix multiplication example with an OpenMP pragma denoting that the outer loop may be parallelised. If this program is executed on the TI OMAP 4 (Figure 5.2), it achieves a speedup of 1.9x relative to sequential execution as shown in Figure 5.1c under the name A9-2T. The current OpenMP implementation is restricted to the 2 A9 cores and hence speedup is limited.

To utilise all the OMAP cores, the programmer currently has to use the Syslink programming model as shown in Listing 5.1. While this achieves good speedup -around 4.9x as shown in Figure 5.1c, it is clear that this is at the cost of significant coding complexity. The function *fxn* performs the actual matrix multiplication computation. Each processor's threads are created in function *syslinkfxn*. All the threads execute the function *compute*.

```
void fxn(buffer* buffers, int start_indx, int end_indx)
{
  int(* restrict A)[1024] = get_address(buffers[0]) ;
  int(* restrict B)[1024] = get_address(buffers[1]) ;
  int(* restrict C)[1024] = get_address(buffers[2]) ;

  int i,j,k;
  for(i = start_indx ; i < end_indx ; i++)
    for(j = 0; j < SIZE; j++)
      for( k = 0; k < SIZE; k++)
        C[i][j] += A[i][k] * B[k][j] ;
}

void remote_compute_stub(void* msg)
{
  if(msg->fd == DSP) {
    fxn(msg->buffers,msg->start_indx,msg->end_indx) ;
  } else {
    foreach core c in (M3-1, M3-2)
      Task_create(task_desc(c),fxn,message(c)) ;
    foreach core c in (M3-1, M3-2)
      Event_pend(task_desc(c)) ;
  }
}

void compute(void* msg)
{
  if(msg->fd == M3 || msg->fd == DSP) {
    rpmsg_remote_compute(msg->fd, msg) ;
  } else {
    fxn(msg->buffers,msg->start_indx,msg->end_indx) ;
  }
}

void syslink_fxn(buffer* buffers)
{
  foreach processor p in (A9-1, A9-2, M3, DSP)
    pthread_create(thread_desc(p),compute,message(p)) ;

  foreach processor p in (A9-1, A9-2, M3, DSP)
    pthread_join(thread_desc(p)) ;
}

int main()
{
  buffer* buffers = alloc_memory_for_arrays() ;
  compute_start_and_end_indices() ;
  syslink_fxn(buffers) ;
  return 0 ;
}
```

Listing 5.1: Matrix multiplication using Syslink

The threads corresponding to M3 and DSP make remote procedure calls (RPCs) with the function *rpmsg_remote_compute*. This will invoke the stub function *remote_compute_stub* on those processors.

Each of these steps has to be repeated for each parallel loop that is encountered in a program. Furthermore, this programming complexity is not always worthwhile. Applying the same approach to a different program `floydwarshall` gives a speedup of nearly 2x when using OpenMP but a significant slowdown when using Syslink due to excessive data movement overhead as shown in Figure 5.1d.

Our approach is to use the SPMD model of computation to partition data and computation across the different cores. Given the original OpenMP program in Figure 5.1a, our compiler first generates the code as shown in Figure 5.1b which is then mapped down to the Syslink implementation. The resulting performance is shown in Figure 5.1c and Figure 5.1d where it outperforms both approaches. We, therefore, maintain the ease of programming in OpenMP and harness the resources of the heterogeneous platform

5.3 Target Architecture and Runtime

For our experiments, we use the PandaBoard[12]. PandaBoard is a low-cost single-board computer. It has the 4th generation Open Multimedia Applications Platform (OMAP) SoC.

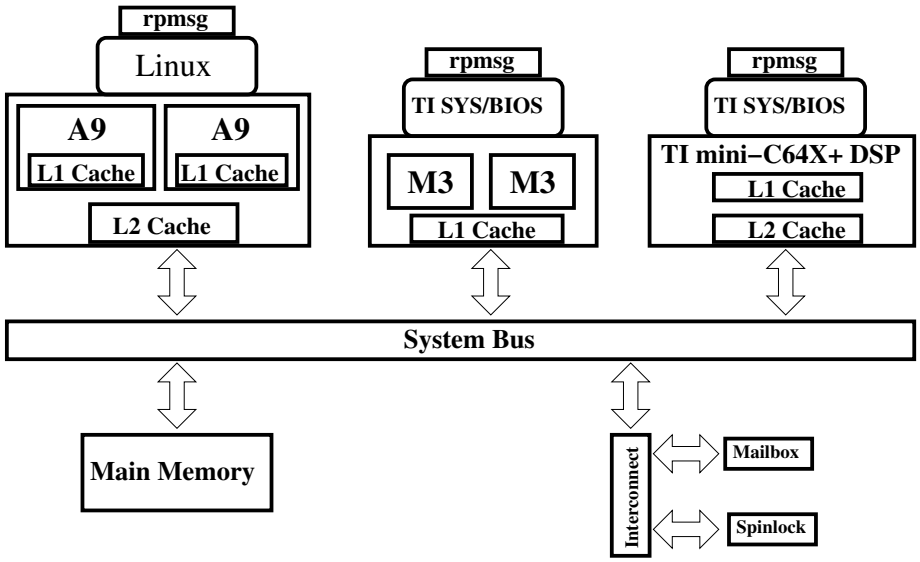


Figure 5.2: OMAP4

Hardware Specifically we use the OMAP4430[7] MPSoC. OMAP4 has 2 ARM A9s, 2 ARM M3s, 1 mini-C64X+ DSP and 1 Imagination GPU besides other accel-

erators. The GPU is not programmable and hence we do not include the GPU as part of this study. The A9s are meant for general purpose computing, the M3s as micro-controllers, and the DSP to make the Image and Video sub-system programmable. The A9s have their own L1 cache of size 32KB but share the L2 cache of size 1MB. The M3s share their L1 cache of size 32KB. The DSP has two levels of cache with L1 cache size of 32KB and L2 cache size of 128 KB. All these processors have their own separate address spaces. The A9 runs at a frequency of 1GHz, M3 at a frequency of 200MHz, and the DSP at 466MHz. It has an LPDDR2 memory running at 400MHz. There are also Hardware Mailboxes and Spinlocks.

Runtime These processors also have different operating systems running on them. The A9s have Ubuntu Linux running on them in Symmetric Multiprocessing (SMP) fashion. The M3s and DSP have a proprietary RTOS from TI called SYS/BIOS as their OS. Again the M3s run the SYS/BIOS in SMP fashion. By default, the programs run only on the A9 processors. To run tasks on the remote processors (M3s and DSP) an Inter Processor Communication Protocol called Syslink is provided. The latest version of Syslink is called Remote Processor Messaging (RPMsg)[1]. This is based on a Remote Procedure Call (RPC) mechanism. RPMsg is built using shared memory and Hardware Mailboxes are used for notification of messages. RPMsg is built into the Linux kernel and SYS/BIOS for providing Inter-processor Communication. Hardware Spinlocks are used for synchronisation between the various processors.

Figure 5.2 shows the relevant portions of the hardware and software present in the OMAP4 SoC.

5.4 Programming models

In this section, we describe three models, OpenMP, Syslink and SPMD that can be used for programming the OMAP MPSoC. OpenMP is a high-level approach that is currently restricted to the dual A9. The low-level Syslink model allows access to all accelerators at significant programming complexity. The SPMD model allows explicit consideration of memory layout but requires compiler support to match heterogeneous architectures.

5.4.1 OpenMP

OpenMP is a widely used high-level shared memory pragma based programming language. The popularity of OpenMP is due to its ease of use and wide availability. OpenMP consists of a set of compiler directives in the form of comment like pragmas. These are used to mark parallel loops and tasks. There are also directives for

specifying scheduling of work, declaration of both shared and private variables and critical sections. OpenMP supports the *fork-join* programming model. In this model, a program starts as a single master thread. When an OpenMP parallel construct is reached, a team of threads is *fork*ed. These threads co-cooperatively perform the work in parallel. At the end of the construct, all the threads *join* and only the master thread continues. Rather than creating and destroying threads at the start and end of parallel regions, most implementations have a pool of threads available which are used and synchronised as needed.

While popular, OpenMP has some issues that make it ill-suited to MPSoC architectures. Firstly, it assumes a single coherent address space and a single operating system. For this reason, it is only available on the dual A9s of the OMAP architecture. More subtly, it is a control-centric parallelisation approach rather than data-centric. Scheduling of parallel loops does not take into consideration data movement impact on other processors and other parts of the program. This has the side effect of increasing synchronisation and memory coherence overhead. To overcome this, programmers have to add affinity scheduling pragmas and no barrier directives.

5.4.2 Syslink Model

Syslink is a low-level API based model which uses pthreads and the IPC mechanism provided by TI to perform computation on all the processors. For each parallel loop in a program, threads are created using the pthread library to be run on each processor. Each thread is also given a message which contains the address of the global arrays and the start and end indices of the computation it needs to perform. These start and end indices are computed at runtime. The threads corresponding to M3 and DSP make RPC calls which invokes a stub function on those processors. If the stub is invoked for the DSP, then the function to compute is called. If the stub is invoked by the M3, then two Tasks are created, each of which calls the function to compute. The threads corresponding to the two A9 cores will directly call the function to compute.

The Syslink Model suffers the overhead of thread creation and RPC calls for each parallel loop. Also, since the computation partition is decided arbitrarily at runtime, there is no guarantee about the dependencies among these partitions, and hence, cache flushes have to be inserted before and after every parallel loop. Additionally, manual conversion of programs from OpenMP to Syslink Model is complex and tedious as is evident from the Syslink Matrix Multiplication example in Listing 5.1.

In Chapter 4, the inter-processor communication mechanism is called Syslink/RPMsg.

In this Chapter, Syslink Model refers to a programming model which uses the inter-processor communication mechanism Syslink/RPMsg without any optimisation.

5.4.3 SPMD

The Single Program Multiple Data (SPMD) model of parallelism is well known and used in a variety of settings. For e.g., Unified Parallel C (UPC)[27] is an extension of the C language which uses the SPMD model. In SPMD, parallel tasks run the same program but operate on independent sections of an array. It is a data-centric approach in which data is first partitioned and scheduled to processors. Computation is then partitioned and scheduled in accordance with this data allocation. Nearly all schemes use an owner-compute or local-write rule [97] where the code executed on a processor is restricted to write to just local data.

Unlike the *fork-join* model, there is no master thread. At program start itself, all the threads start running in parallel. The number of threads that correspond to the number of processing elements is fixed throughout the execution of a program. Synchronisation is not associated with the beginning or end of parallel loops but with whether or not there is a cross-processor data dependence i.e., data written on one processor is read on a different processor. Barriers are inserted to honour such dependences.

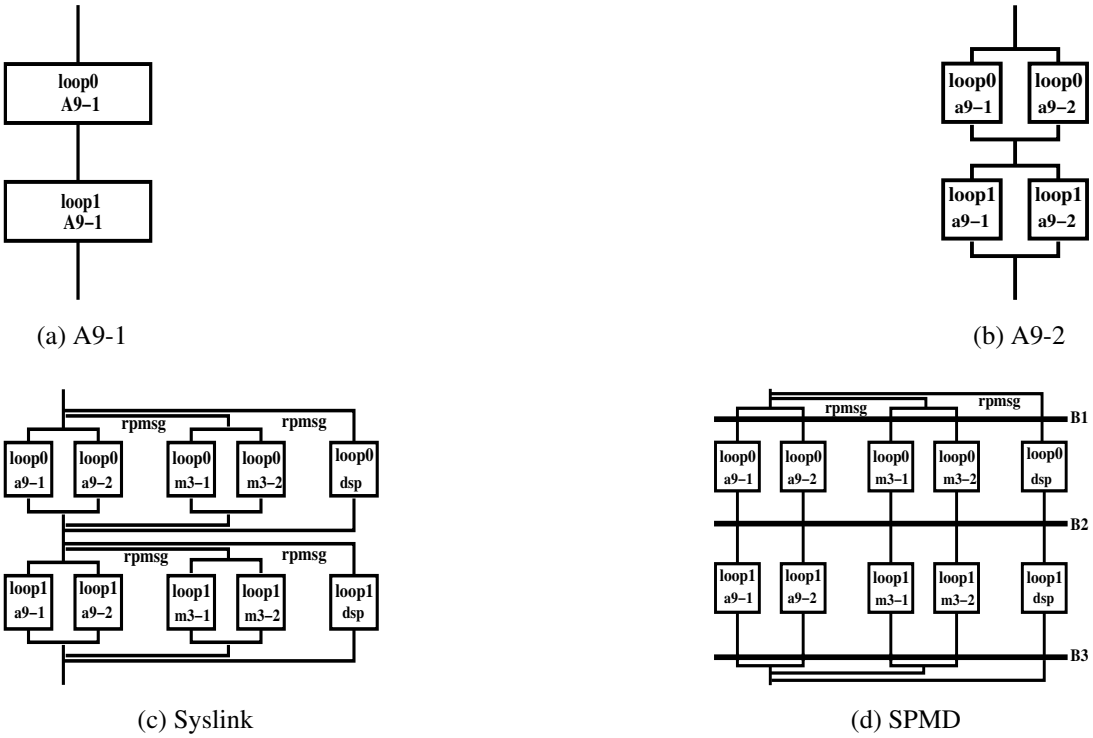
The key benefit of SPMD for MPSoCs is that data layout is known. It forms a natural bridging model between high-level OpenMP and low-level Syslink.

5.4.4 Example

Figure 5.3a shows a sequential program with two loops, loop0 and loop1. This program runs on one core of the A9. Figure 5.3b shows an OpenMP program which runs on two A9s. As can be seen, the parallel loops are split into two and runs on both cores, while the sequential part runs only on one core. Figure 5.3c shows the Syslink Model. In this, the parallel loops run on all the cores of the processors, while the sequential portion runs on only one core of the A9. Figure 5.3d shows the SPMD Model. In this the parallel and sequential portions run on all the cores. Synchronisation is done using barriers. Barriers are placed at the start and the end of the program and points of cross processor dependencies.

5.5 SPMD as a bridging model

Our goal is to both allow programmers to continue to use OpenMP as a high level programming language and exploit the heterogeneous accelerators. In effect, we want to port OpenMP to MPSoCs. We achieve this by using SPMD as a bridging model;



(a) A9-1 (b) A9-2

(c) Syslink (d) SPMD

Figure 5.3: OpenMP and SPMD

first we map OpenMP to SPMD and then map SPMD to MPSoC.

5.5.1 Mapping OpenMP to SPMD

The SPMD model assumes that data is first partitioned across the processors and then local computation is derived using the local-write rule. A good data partition will be one that exposes parallelism without introducing excessive overheads. In our approach we, therefore, interpret OpenMP parallel loop pragmas as defining which dimensions of an array may be executed in parallel on a loop by loop basis. This information is collected for the whole program and used to determine a global data partitioning. The key point is that OpenMP pragmas are used for analysis to drive to global data partitioning rather than as directives on how to schedule loops. Once the data partitioning is determined, the exact amount to be scheduled to each processor must be calculated before generating the local code for each processor.

5.5.2 Mapping SPMD to MPSoC

Once we have determined the local programs for each processor, these must be allocated, compiled by the host compiler and executed. The details of how this is achieved are described in Section 5.6.5.The SPMD model assumes a single coherent address

space. We achieve this by allocating memory in a single contiguous Linux memory space and map these to the M3 and DSP memory spaces as described in Section 5.7. Coherence need only be enforced at synchronisation points. We, therefore, flush cached data to main memory at each barrier synchronisation point. As this is an expensive operation, we try to minimise the number of synchronisation points and flush only data that is out-of date.

5.6 Compiler Algorithm

This section describes the different compiler stages involved in mapping OpenMP to MPSoCs. The overall algorithm is shown in Algorithm 1.

Algorithm 1 CompileforMPSoC

1: Determine data partition dimension.
2: Determine data bounds per processor.
3: Determine Computation partition.
4: Determine barrier synchronisation points.
5: Determine data to be flushed.
6: **for all** processor p: $p \in A9, M3, DSP$ **do**
7: Generate node code for each processor p.
8: **end for**

5.6.1 Data partition

As the SPMD model is data-centric, determining the overall data partitioning is a critical global decision. We consider all arrays are aligned to a common index domain [73], and it is necessary to determine which indices should be partitioned. Due to the relatively low number of cores within an MPSoC, we currently restrict partitioning to just one index or dimension. Using the OpenMP pragmas, we consider which dimensions can be executed in parallel for the most computationally expensive sections of the program. Such sections are determined statically based on loop nest depth. If there are more than one dimension which can be executed in parallel, then we choose the index which causes the maximum number of elements to work in parallel.

False Sharing Partitioning of arrays has to avoid false sharing at the cache line level. By default, C language defines arrays to be stored in row-wise format. As column elements are not consecutively stored, if we partition by columns, we might inadvertently write incorrect data to other elements in the cache line. To avoid this problem we only partition on the first dimension i.e., rows. In cases where the best

index to partition on is the 2nd or higher dimension, we simply apply global index reordering on all arrays, so that partitioned dimension is innermost.

5.6.2 Data bounds per processor

In the second step, we determine the amount of elements from the selected dimension to allocate to each processor. We use the *iter* partitioning method presented in Chapter 4.4 for this purpose. It uses an iterative profiling based method. The program is run with a reduced data set on each of the processors, and the profile data is collected. From this profile data, the throughput of each processor is calculated and the data partition assigned is proportional to its throughput.

5.6.3 Computation Partitioning

After data partitioning, the work scheduled to each processor has to be determined. The local write rule, where each thread writes only to local data, places a constraint on the loop bounds. Figure 5.1b shows an example of such loop bounds. The local write rule ensures that there are no remote-writes, and all output dependences are within a thread by definition. This information is profitably used in determining synchronisation placement.

Scalars

While arrays are partitioned across the processor space, scalars need different treatment. By default, all scalar variables are privatised. This means, all reference to scalars are local and do not incur any cross-processor dependences.

Reductions

Reductions require special treatment. Each processor performs a reduction on its local data and writes the result to a local element of a shared reduction array. After synchronisation, each thread accesses the remote elements of the shared reduction array and performs a final reduction locally.

Synchronisation

Whenever there is a cross-processor dependence data, synchronisation is necessary. Ideally, we wish to place the minimal number of synchronisations to cover all dependences. We first construct a cross processor dependence graph from the standard dependence graph. All output dependences and scalar dependences are removed as they are guaranteed to be local. Next, any read access that has the same subscript on the partitioned index as its corresponding write access is guaranteed to refer to local

data due to global data partitioning. Any dependences associated with such a read can also be eliminated. Given this reduced cross-processor data dependence, we place barriers to cut all dependences based on the algorithm given in [111], which places synchronisation directives at the highest lexical level that covers the dependence.

5.6.4 Cache Flush

Placing synchronisation directives ensure the correctness of a program in cache coherent systems. But in systems where caches are not hardware coherent, the compiler is responsible for inserting cache flushes around barriers. These flushes are necessary so that, if a memory location is updated by a processor and read later by another processor, then the latter should read the updated value. In a trivial translation, flushes are inserted for all the memories, which correspond to the global arrays, before and after every parallel loop. But this is very conservative and can lead to slowdowns. We only need to flush the memories corresponding to arrays that were updated and which are involved in cross processor dependencies. We use the dependence analysis mentioned in Section 5.6.3 for barriers to finding these memories and the locations in the program to insert the flushes. An array section analysis is necessary to determine the exact sections of the array to be flushed. Array section analysis is a well-researched topic, and more details can be found in [25, 57, 39].

5.6.5 Code Generation

Once we have the local programs for each processor, these have to be mapped down to the Syslink model. In particular, we need to consider how parallel threads and synchronisation are supported.

Threads

Threads are implemented using pthreads in the A9 cores. We create two threads for the A9, and one each for the M3 and the DSP. *pthread_create*, and *pthread_join* is used for thread creation and joining. The thread for M3 makes an RPC call to get the work done on the remote processors. In M3, the threads are created using Sysbios tasks and events. *Task_create* and *Event_post*, *Event_pend* are used for thread creation and joining in the M3 processor. We run only a single thread on the DSP. The final code executed has the same structure as the Syslink code shown in Listing 5.1, except that the *fxn* function is replaced with the generated SPMD code, which is arbitrarily long containing synchronisations and cache flushes.

Synchronisation

Synchronisation is implemented using barriers. Inter-processor barrier synchronisation is implemented using shared memory and hardware spinlocks. Synchronisation is also needed among threads running on the same processor but different cores as in the A9 and M3. Local synchronisation in A9 is performed using *pthread_barrier* while on the M3 it is performed using barriers created using semaphores and memory.

5.7 Shared Memory on an MPSoC

One of our key requirements is having a single address space.

In our scheme, Memory is allocated in a contiguous area from the Linux kernel. On the A9 side, this area is memory mapped to the user-space. We obtain the physical address corresponding to this area which is sent to the M3 and DSP using rpmsg. The memory map table of M3 and DSP, for the range of addresses corresponding to this contiguous area, is set up such that the physical address is equal to the virtual address. Hence, the physical address on the A9 side can be directly used on the M3 and DSP.

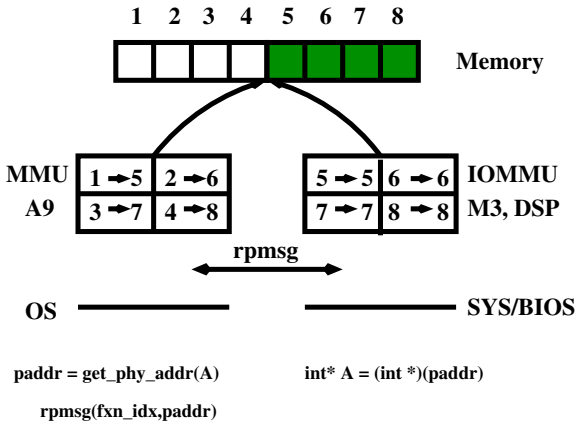


Figure 5.4: Our Model

On the A9, shared array declaration of arrays is replaced with memory allocation from the contiguous memory. System calls like malloc are used for this allocation. On the M3 and DSP, these allocations are not needed. The addresses of these global arrays and the start and end indices which denote the partition corresponding to the remote processors are sent by rpmsg. We replace the declaration of these arrays with a type-casting from the physical addresses received through rpmsg.

This approach is shown in Figure 5.4. The Figure shows a memory consisting of 8 locations. The coloured/darkened area is the contiguous area reserved for our purpose. The Memory Management Unit (MMU) of A9 shows the translation from virtual to

Benchmark	Short Name	Par. Loops
Matrix Multiplication	mxm	1
Dotproduct	dotp	2
Edgedetect	edge	4
Histogram	hist	3
Doitgen	dgen	3
Regdetect	regd	4
FloydWarshall	flwl	1

Table 5.1: Benchmarks

physical address for these locations, e.g., 1→5, 2→6. Whereas the MMU of M3 and DSP shows a 1:1 mapping for these addresses, e.g., 5→5, 6→6. The MMU of the DSP and M3 are programmed when the corresponding firmware is loaded. Let us assume that the virtual address of the array A is 1. By making a system call, the physical address is found to be 5. We make an RPMsg call to perform work on the remote processor. The index of the function that we want to call as well as the physical address of the array A(5) is passed as an argument. At the remote processor, the physical address is typecast to the array A and the remote processor can now work on the array A.

5.8 Compiler Implementation

In this section, we explain in brief our compilation Framework. Figure 5.5 shows the framework. We use data-parallel OpenMP programs as input. The input program is fed to a src-to-src translator. A source to source translation is performed to convert it into an SPMD type program. Then we partition the code for all the processors and code is generated for each of the processors. The generated code contains barriers for synchronisation and rpmsg calls to get work done on the remote processors. Then we use specific compilers to make the binaries for all these processors.

5.8.1 Source to Source Translation

We use Clang/LLVM to do source to source translation of OpenMP data-parallel programs to the SPMD form. We use the OpenMP Clang frontend [11] open sourced

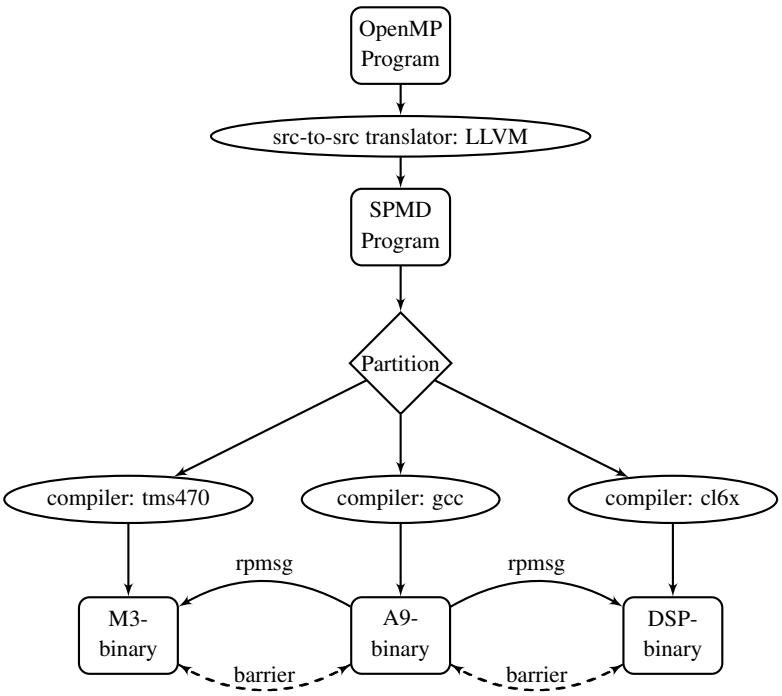


Figure 5.5: Compiler Flow

by Intel to perform the source to source translation. This tool annotates the clang AST with OpenMP pragmas. Once the AST is constructed and annotated by this tool we traverse the AST and perform the required translation. Callbacks are inserted for the OpenMP Pragmas and these callback functions get called when the pragmas are reached during traversal. We use the Rewriter framework in Clang/LLVM for source to source translation. The simplest way to translate is whenever we see an OpenMP parallel for pragma, we replace the start and end indices of the loop associated with the pragma with two variables *start_indx*, *end_indx*. When this replacement is done the *start_indx* and *end_indx* have to be propagated to the function parameters as well as to the sites where this function is called.

Compilation

Table 5.2 shows all the details of the platform used. We use the gcc compiler to compile for the A9 processor on the pandaboard. The board runs the Ubuntu Linux OS. For M3 and DSP, the programs are cross-compiled remotely using proprietary TI compilers tms470 and cl6x compiler to generate firmware. These compilers come as part of the TI Code Composer Studio (CCS). These firmware are then copied to the pandaboard and put in /lib/firmware. When some kernel modules are inserted, this firmware is loaded onto the remote processors.

Optimisations

We use the highest optimisation levels as default for all the processors. This is *O2* for the A9, and *O3* for the DSP and M3 processors. Many of the benchmarks that we have tried can potentially benefit from memory access optimisations, but that is not the focus of our work.

Platforms like the DSP are extremely sensitive to memory aliasing and dependence issues, hence, if there are more than one parameters to a function which are arrays, then we should qualify them with *restrict* if the parameters are not expected to alias. Use of *restrict* also can give marginal benefits on the A9 and M3 processors. There are also benefits for the DSP when using the pragma *MUST_ITERATE* which gives bounds for the loops.

Processors like the M3 have limited cache. There can be thrashing when two threads run simultaneously and fetch and operate on a lot of data from the memory. So sometimes it is better just to have one thread working on the data.

Processor	Cortex A9	Cortex M3	C64X+ DSP
Vendor	ARM	ARM	TI
Clock	1 GHz	200 MHz	466 MHz
L1 Cache	32 KB	32 KB	32 KB
L2 Cache	1 MB	-	128 KB
Memory	LPDDR2 400 MHz		
OS	Ubuntu 12.04-3	SMP/BIOS	SYS/BIOS
Compiler	gcc 4.6.3	CCS tms470	CCS cl6x

Table 5.2: Platform details

RPMsg

RPMsg is a way of performing Inter-processor communication in the TI OMAP system. RPMsg is implemented in the Linux kernel on the A9 side and sysbios on the M3 and DSP side. User applications in the A9 side make system calls to invoke RPC calls to get work done on the remote processors. An RPC call contains the id of the function to be invoked on the remote processor, the physical addresses of the arrays, and the location of the partition via *start_indx* and *end_indx*. The RPMsg messages are passed using the virtio framework. This framework has circular linkedlists for sending and receiving messages. After sending a message, the recipient processor is notified by

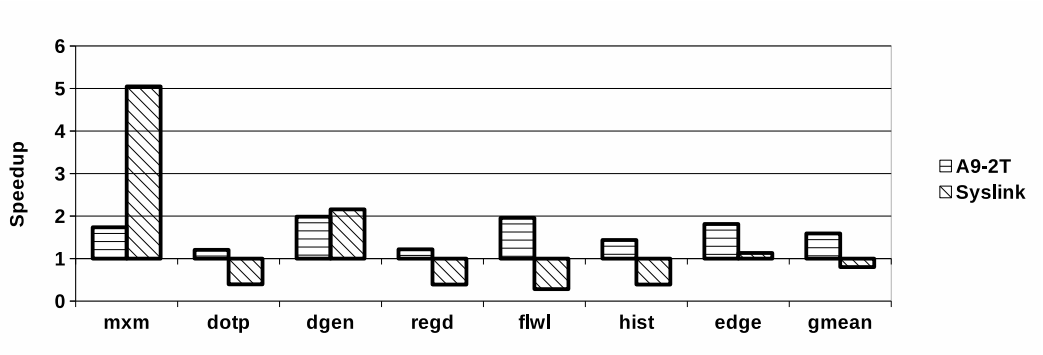


Figure 5.6: Results for Syslink compared to A9-2T. Results are normalised to sequential execution on A9. The Syslink implementation gives significant improvement for mxm, but performs poorly for dotp, regd, flwl and hist. While A9-2T has an average speedup of 1.59x, Syslink has a slowdown of 0.8x.

sending a message to the mailbox. On receiving a notification of a message, a server like thread is called which checks the message. This server invokes a thread to run the function whose id is contained in the RPC call.

5.9 Setup

In this section, we describe the experimental setup. We briefly describe the benchmarks that we use and the method for measuring runtime in this section.

5.9.1 Benchmarks

We use a few data-parallel benchmarks from DSPstone[113], UTDSP[70] and polybench[92] for experiments. These benchmarks are parallelised by annotating them with OpenMP pragmas. From the DSPstone suite, we have Matrix Multiplication and Dotproduct. Dotproduct has a reduction. From UTDSP we have Edgedetect and Histogram. From Polybench we have Doitgen, Regdetect and FloydWarshall. Doitgen contains some matrix operations, and Regdetect is a regularity detection algorithm for 2D images. Barriers are needed for synchronisation in all benchmarks except Matrix Multiplication and Doitgen. Table 5.1 lists all the benchmarks, the short names used for these in the Chapter and the number of parallel loops in these benchmarks.

5.9.2 Measurement

System calls form part of the Syslink framework. We have to include the runtime of these calls in our runtime measurement. Hence, we need an API function which measures the system time as well as user time. For these reasons, we use the *gettimeofday*

API function. This function is called before forking off the threads and after the threads join. The difference in time between these two calls is the execution time. We take the measurement 10 times and report the median value as the result.

5.10 Experiments

This section investigates the performance of OpenMP, Syslink and our approach and includes an investigation of the impact of barrier synchronisation and cache flushing. This is followed by an evaluation of the energy used by the best 2 schemes and the resulting Energy-Delay-Delay, EDD, product. Finally, we perform a simple limit study to see whether further communication optimisation can improve performance.

5.10.1 Syslink and OpenMP Model

Figure 5.6 shows the results for the Syslink Model compared to the OpenMP default model (A9-2T) that just uses the two A9 cores. On the y-axis, we have the Speedup over sequential execution on the A9. Syslink achieves better speedup than A9-2T for the *mxm* and *dgen* benchmarks. In *edge*, the Syslink model is worse than running on A9 with two threads but better than sequential execution. For other benchmarks viz. *dotp*, *regd*, *flwl*, and *hist*, the Syslink model is worse compared to running sequentially on the A9. On average, while A9-2T had a speedup of 1.59x, the Syslink model has a slowdown of 0.8x. In trying to improve the performance by using the other processors, we have ended up degrading in performance in five out of the seven benchmarks.

5.10.2 Our SPMD approach

In this section, we evaluate our SPMD based approach. In particular, we examine the performance achieved using a naive ”flush all data” at a barrier approach vs. a smart ”flush only data written to”. Figure 5.7 shows the speedups achieved with and without flush optimisations. *F-ALL* is the default scheme where all data is flushed at a barrier. It should be noted that *F-ALL* is not the same as Syslink. The primary difference is that *F-ALL* runs in SPMD style whereas Syslink runs in fork-join style. *F-OPT* refers to code with flushes optimised. As can be seen, with naive cache flushing our scheme achieves poor performance. In only two benchmarks, `mxm` and `dgen` do we see any speedup as the amount of data flushed is small relative to the total computation. In all other cases, it leads to slow-down, 0.78x and performs on average worse than the default OpenMP A9-2T scheme. Applying optimised flushing has a significant impact on program performance. In all cases except `mxm`, it significantly improves over the naive scheme. Furthermore, it outperforms the OpenMP A9-2T approach in all cases

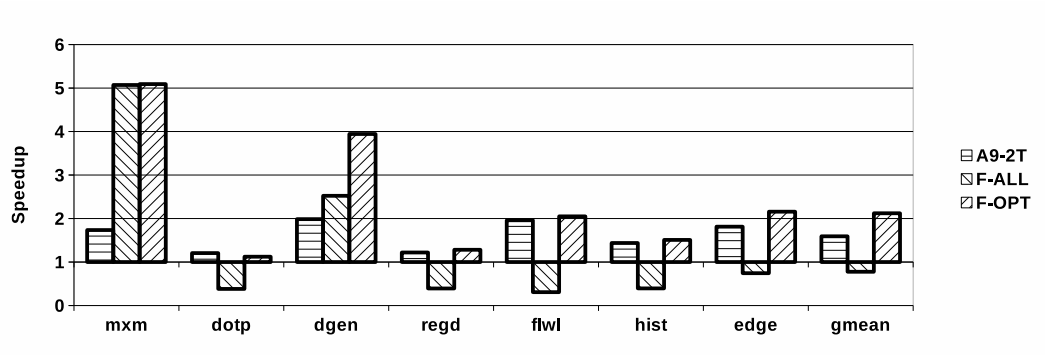


Figure 5.7: Results for our SPMD based compiler model with and without Flush Optimisations. Results are normalised to sequential execution on the A9. Without flush optimisations, the SPMD model is slower than A9-2T. With flush optimisations, it is faster than A9-2T and achieves an average speedup of 2.12x.

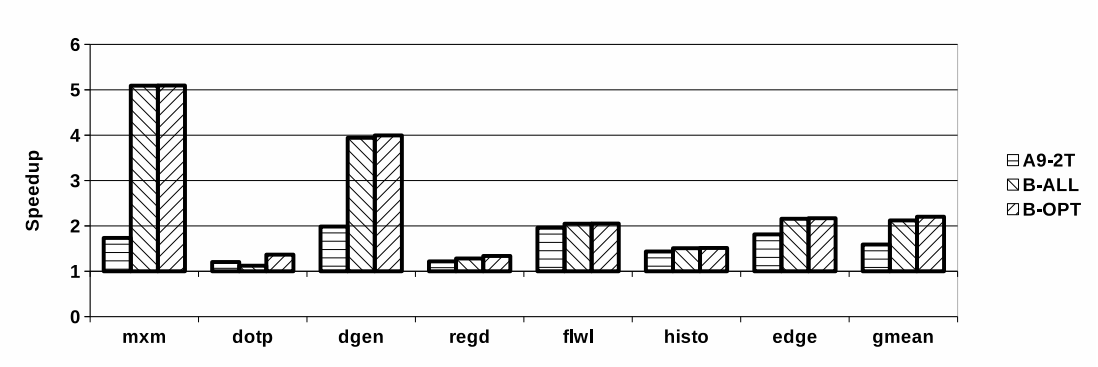


Figure 5.8: Results for SPMD Model with and without Barrier Optimisations. Results are normalised to sequential execution on the A9. Without barrier optimisations, only *dotp* is slower than A9-2T. With barrier optimisations, a speedup of 2.20x is achieved which is not significantly higher than without Barrier Optimisations.

except `dotp` giving an average speedup of 2.12x.

5.10.3 SPMD Barrier optimisations

In this section, we compare the impact of smart barrier placement within our model. Figure 5.8 shows the results with and without barrier optimisations. *B-ALL* refers to a naive scheme where barriers are placed whenever we encounter an OpenMP parallel pragma. *B-OPT* refers to the case where barriers are placed solely based on cross-processor dependence analysis. In most cases we see little improvement in overheads in smart placement. This is because there is only a maximum of five threads which are synchronising at the barrier. Only in the case of `dotp` benchmark do we see an improvement. Removing the barrier allows better load balancing and reduces execution time. On average, the SPMD programs with barrier optimisation achieves a speedup

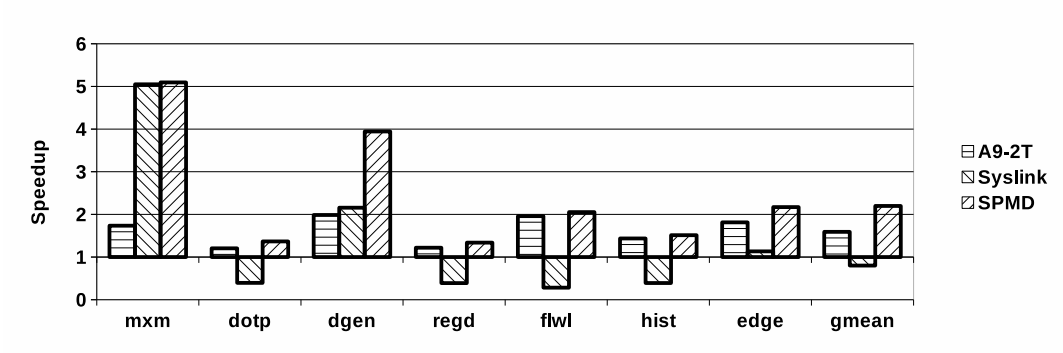


Figure 5.9: Results for SPMD Model compared with Syslink and A9-2T. Results are normalised to sequential execution on the A9. On average, the SPMD model delivers a speedup of 2.20x, Syslink delivers a slowdown of 0.8x, and A9-2T delivers a speedup of 1.59x.

of 2.20x compared to 2.12x without optimisations.

5.10.4 Comparison Summary

In this section, we summarise our results so far, comparing the optimised SPMD model with A9-2T and Syslink. Figure 5.9 shows the results for the SPMD and Syslink model. As can be seen, the Syslink mode is worse compared to SPMD model in all benchmarks. The runtime is comparable in only one benchmark, ie *mxm*. The improvements in performance over Syslink comes primarily due to compiler directed optimal cache flushing. Across all benchmarks, the SPMD model is better than *A9-2T*. On average, the SPMD model gives a speedup of 2.20x, Syslink a slowdown of 0.8x, and A9-2T a speedup of 1.59x. In other words, the SPMD model gives a speedup of 2.75x over Syslink and 1.38x over A9-2T.

5.10.5 Energy and EDD

One of the benefits that a heterogeneous multicore system offers is power- performance trade offs. In the target architecture, the M3 and DSP are low power processors. Allocating work to the M3 and DSP not only improves performance but also improves energy efficiency. In this section, we quantify the energy and hence EDD benefits. We compare against the best performing OpenMP approach. We did not evaluate Syslink as it gives, on average, a slowdown and hence would have excessive energy consumption. We measure Power by monitoring the current flowing into each of the processors and the memory and multiplying it by the supply voltage. We use the method described by Jos[115]. Energy is measured by measuring the instantaneous power over the entire duration of a program execution using an Oscilloscope.

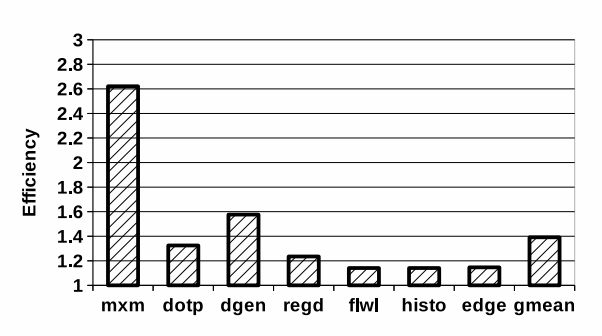


Figure 5.10: Energy of SPMD normalised to A9-2T. On average, SPMD model achieves an energy efficiency of 1.39x over A9-2T. The largest improvement is for *mxm*

Figure 5.10 shows the energy efficiency of the SPMD model compared to A9-2T. We get the maximum efficiency improvement in *mxm* of around 2.62x. This improvement is because around 60% of the work is now done by the M3 and DSP processors for *mxm* in the SPMD mode. This is followed by *dgen*, *dotp* and *regd*. We achieve an improvement of around 1.14 for *flwl*, *histo* and *edge*. In general, the improvement in efficiency is proportional to the amount of work that is done by the low power M3 and DSP processors. On average, with the SPMD mode we get an improvement in energy of 1.4x when compared to A9-2T.

Finally, Figure 5.11 shows the fraction of EDD of SPMD when compared to A9-2T. For *flwl* and *histo*, SPMD has around 0.8x the EDD compared to A9-2T, for *regd* around 0.67x, for *edge* and *dotp* around 0.6x, for *dgen* around 0.16x and *mxm* just around 0.04x. On average, the SPMD mode needs only 0.37x the EDD of A9-2T.

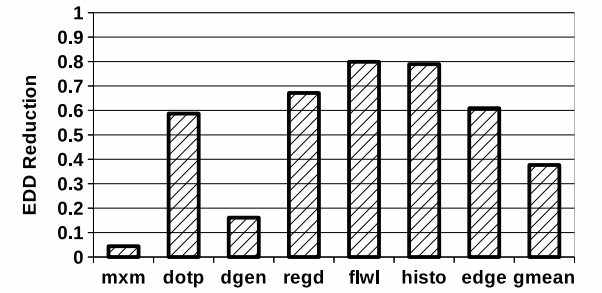


Figure 5.11: SPMD EDD normalised to EDD of A9-2T. On average, the SPMD Mode needs only 0.37x the EDD of A9-2T.

5.10.6 Limit Study

In this section, we report the results of a study on the performance limit possible with SPMD. Here, we remove all the barriers and flushes from the SPMD program and compare the performance with the optimised SPMD program. It should be noted that this performance will not be achievable since removing all the barriers and cache flushes will result in incorrect results. Figure 5.12 shows the results of this study. The maximum possible performance improvement is in *flwl* of 1.13x. In benchmarks like *mxm* and *dgen*, there is no possible improvement. On average, the possible improvement is 1.05x. From this, we can conclude that there is little room for improvement in further

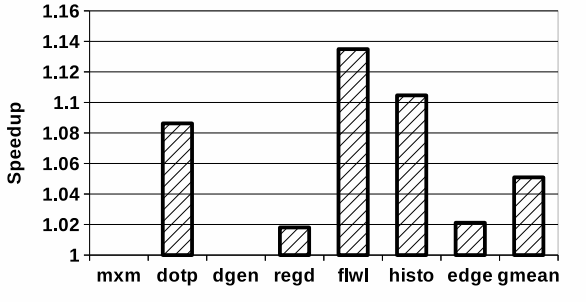


Figure 5.12: Performance Limit: Runtime of SPMD without barriers and flushes normalised to optimised SPMD. On average there is a possibility of improving the performance by 1.05x only. The maximum possible improvement is for *flwl*

barrier/flush implementation. To achieve further performance improvement, we should concentrate on optimisations of individual processors.

5.11 Summary

We have presented an LLVM-based compiler approach for compiling data-parallel OpenMP programs to heterogeneous embedded systems. We have shown that by using the SPMD model of compilation we can obtain better performance than the existing Syslink model. Smart insertion of cache flushes is important for achieving good performance. On average, our approach achieves a 2.75x speedup over Syslink. Furthermore, it delivers a speedup of 1.38x and an improved energy efficiency of 1.4x over using the 2 A9 cores alone. All these improvements are achieved without the user having to make any code changes.

The previous and current Chapters demonstrated the advantages of partitioning data-parallel programs for heterogeneous processors and described a compiler framework for automation. One of the big challenges of the future is Dark Silicon where

all the cores cannot be active at the same time due to power budgets. The next Chapter explores code generation for heterogeneous processors in the presence of a power budget.

Chapter 6

Power Constrained Code Generation

This Chapter presents a compiler based method for code generation for heterogeneous processors in the presence of a power budget.

Section 6.1 introduces the work in this Chapter. Section 6.2 presents the motivation for the work. Section 6.3 presents our approach and Section 6.4 discusses the details of code generation. Section 6.5 describes the power constrained model. Section 6.6 describes the experimental setup and Section 6.7 presents experimental results. Finally, Section 6.8 concludes the Chapter.

6.1 Introduction

Power is a first class constraint in modern processor design. It is the driving force behind the shift to multi-cores in today's computing systems. Ever-higher clock frequency scaling is unsustainable due to power-density and thermal limitations [105]. Parallel programming on multiple cores has the potential for increased performance without increased power. Power is also the reason for increased processor heterogeneity. Dark silicon [45] suggests that since we will be unable to simultaneously power-up all the cores on a chip, cores should be specialised to best utilise the available power.

Simple heterogeneous GPU based systems are now common-place. In this Chapter, we are interested in investigating more challenging heterogeneous platforms that are likely to become more prevalent in the future. In particular, we examine the TI OMAP4430 [6] and Samsung Exynos 5422 [3] platforms which contain heterogeneous CPUs, heterogeneous GPUs and DSP cores. The non-CPU processors in these platforms are generally used for offloading for power-efficiency/performance. These processors have different micro-architecture, memory hierarchy and/or Operating Systems. There is no single programming language, library or runtime to program these platforms. Hence, these are highly challenging to program but offer the potential for

excellent power and performance.

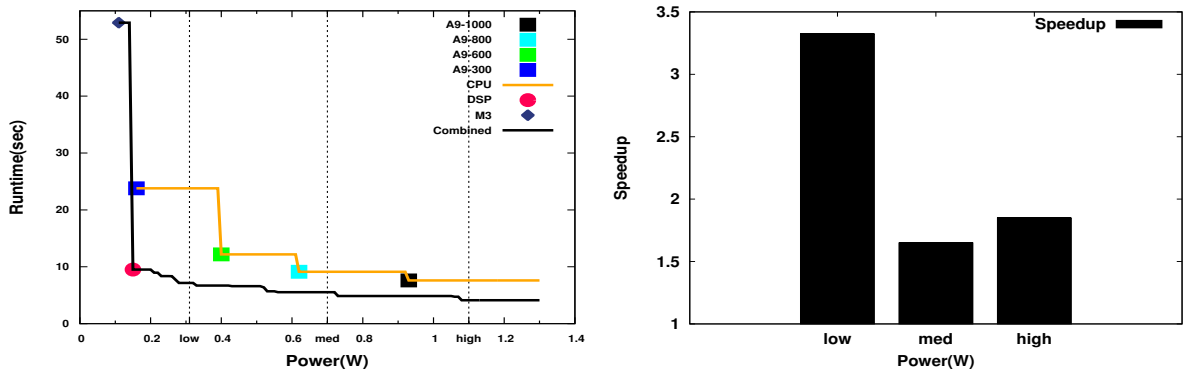
Given that, it is frequently not possible to run all cores at their maximum clock rate for long periods, runtime power-management is needed to keep within the thermal constraints[105]. Such power management is the responsibility of the operating system or hardware which uses *DVFS* or power gating to reduce power at the expense of performance.

Such approaches work well with homogeneous multi-cores or single-ISA heterogeneous cores as there is no need to modify the binaries. However, if, due to dark silicon [45] we have more diverse cores each with their own ISA and specialised behaviour, there is no straightforward approach to using the right cores based on program suitability and available power budget. It is not possible to migrate code compiled for a DSP ISA to a CPU ISA. Even if it were possible, given that some cores will be more power-efficient for certain applications and not others, it is not clear which core(s) to migrate to. In future, we will see an ever-increasing number of cores available for a newly scheduled application, but only a subset may be used due to a limited power budget. Here we need to select from the cores those that give the best performance and allocate the appropriate amount of work to each.

Previous approaches have focused on CPU/GPU power-based allocation of work rather than examining more general heterogeneous multi-cores. In [19] they build a model to select where to run a program, either on a CPU or a GPU based on power. Work presented in [121] uses an online search of the kernel partition parameter space. Although it extends work on kernel partitioning to consider power, this is at the expense of additional execution every time a kernel is run.

This Chapter presents a compiler-based approach to runtime power-management for heterogeneous cores. Given an externally provided power budget, it generates heterogeneous, partitioned code that attempts to give the best performance within that budget. At runtime, it selects parameters determining the workload on each core. We consider a number of different selection policies that combine program and system information.

We applied this approach to parallel OpenMP benchmarks on the OMAP 4430 and Exynos 5422 platforms. We evaluated a number of parameter selection policies and compared against an *idealised* perfect *DVFS* scheme. Such a scheme always picks the best voltage and frequency that ensures the program stays just within the power budget. It represents the best that could be realistically achieved by DVFS approach and is a strong baseline to compare against. We performed a partitioning Oracle study which



(a) Pareto curve. Best runtime available for power budget

(b) Low, Mid, High power Speedup

Figure 6.1: OMAP Power/Performance pareto: Using all the processors gives better pareto points than using the CPU only with *DVFS*. This is particularly true with a low power budget where there is significant 3.3x speedup for Doitgen

shows that the best parameter selection gives an average 1.55X and 1.30X speedup across all programs and power budgets. Determining the right parameters is however challenging and we show that a *Naive* scheme that ignores program-structure and just uses average power values, in fact, performs worse than the idealised *DVFS*.

Using our *Profile-directed* analytic model we achieve on average a 1.37X and 1.15X speedup averaged across power budgets, showing that power constrained heterogeneous code generation is an effective technique for managing future heterogeneous systems.

The rest of the Chapter is organised as follows: the next section presents a simple example demonstrating that partitioning for heterogeneous processors gives better performance for a power budget than relying just on DVFS. This is followed by a brief overview of our approach, code generation and the various power modelling policies proposed. Next, we present the experimental setup which is followed by the experimental results, related work and conclusion.

6.2 Motivation

When an application is to be scheduled, it must not exceed the maximum available power. The standard way to manage varying power availability is to change frequency and voltage. This has the effect of reducing power consumption at the expense of slowing the processor and hence the application. To illustrate the effect of frequency/-voltage scaling, consider Figure 6.1a. It shows the power-runtime trade-off points for the parallel OpenMP `Doitgen` benchmark from `Polybench` on the OMAP4 platform.

The details of OMAP architecture are given in Table 6.1.

The OMAP4 has three processor types that can be programmed: a *DVFS*-capable ARM Cortes 2xA9 CPU, an ARM Cortex 2xM3 and a TI DSP. It also has a GPU, but this is not programmable with current, publicly available APIs.

The OpenMP implementation utilises the 2 A9 cores which run at frequencies varying from 300MHz to 1GHz. The 4 frequency levels form a power/ performance Pareto curve highlighted by the yellow line. Given a power budget, the frequency can be scaled to trade off performance for power savings.

If the other processors are available, and we can generate code for them, it is possible to improve performance across the power spectrum. The power and performance of the DSP are denoted by the red circle. It provides an excellent power/runtime trade-off. The M3 denoted by the blue diamond provides the lowest power at severely increased runtime. In isolation, they do not provide the flexibility of the *DVFS* on the A9s. However, we can generate partitioned code for each core and share the work across the processors. If we can determine the ideal amount of much work for each core at the appropriate frequency/voltage scale, we obtain the new power/performance Pareto curve shown in black.

Power/Performance

It is clear that across the power spectrum, correctly utilising the cores gives a considerable performance improvement on both platforms. If we pick three representative power budgets of low (0.3W), medium (0.7W) and high (1.1W), we obtain the speedups shown in Figure 6.1b where we obtain a speedup of 3.32x for low, 1.65x for medium and 1.85x for high over using the default Cortex cores alone with ideal *DVFS*.

We have shown that using heterogeneous processors gives significant improvement over idealised DVFS alone for a given power budget. To attain this performance improvement, there are two key challenges to overcome: (i) generating partitioned code for such a heterogeneous system and (ii) determining the correct amount of work to allocate to each core. The next section describes our overall approach.

6.3 Our Approach

We assume that we are provided with a power-budget at runtime and must generate and execute code that is as fast as possible but does not exceed this power budget. We achieve this by a 2-stage process: (i) compile-time generation of parametrised heterogeneous code; (ii) runtime selection of code parameters.

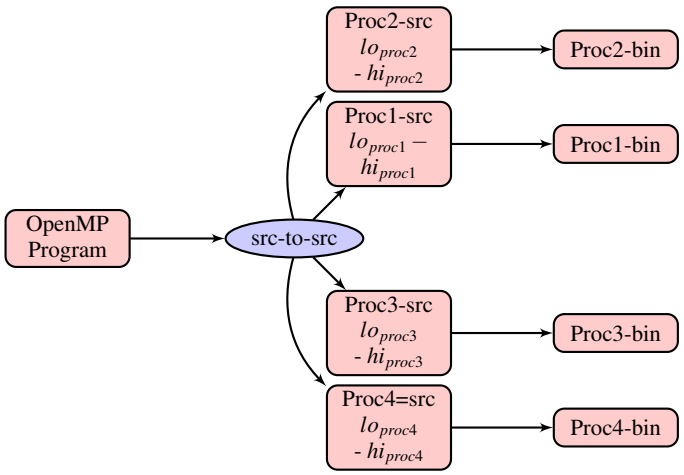


Figure 6.2: Compiler Flow. We apply a source to source transformation that partitions the original OpenMP program based on a hybrid SPMD model. It outputs a program for each of the heterogeneous processors which are then compiled by the local node compiler to produce an executable

6.3.1 Compile-time

In this work, we restrict our attention to a subset of OpenMP programs with pragmas denoting parallel loops. Our goal is to take such high-level platform independent programs and generate code that runs on the heterogeneous cores. Partitioning and OpenCL generation is performed using an LLVM-based source to source compiler [30][51]. We then rely on the local compiler for each core to generate the local binaries. This is shown in Figure 6.2. The local programs are parametrised by parameters **lo** and **hi** which denote the partition and hence the amount of data/work allocated to each processor. These values are set at runtime.

6.3.2 Runtime

Once we have the parametrised programs for each core, we determine the parameter values so as to give the best performance within a power budget. In our approach, as shown in Figure 6.3, we have a runtime library which is provided with four inputs. i) a power budget, ii) the cores available and possibly iii) static information on the SPMD programs, iv) the individual runtime and power of the program on each of the processors. The runtime decides the DVFS setting and the **lo** and **hi** values for these processors based on a policy. When the binary for the program starts executing, it makes calls to the runtime which assigns the partition values and also sets the DVFS value of the CPU. In section 6.5.3 we examine policies that use some or all of these

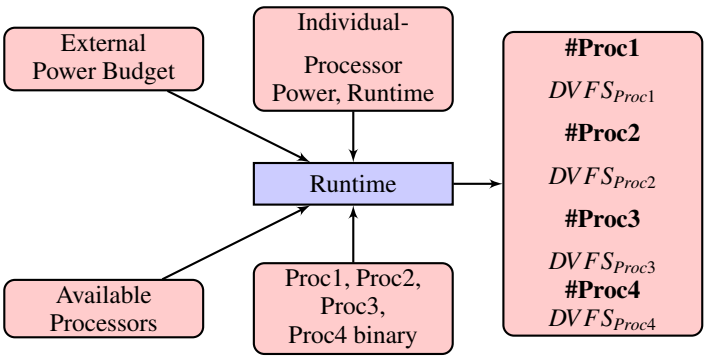


Figure 6.3: Runtime: At runtime we determine the amount of data and work allocated to each of the processors as well as the DVFS level it should run at. This decision is based on the processors available and the available budget; both externally determined. Depending on the policy, we may also have static or profile information on the program

	OMAP 4430			**Exynos 5422**			
Name	Proc1	Proc2	Proc3	Proc1	Proc2	Proc3	Proc4
Processor	Arm A9	Arm M3 μC	TI DSP	Arm A15	Arm A7	Mali-T628 GPU	Mali-T628 GPU
Cores	2	2	1	4	4	4	2
Threads	1	1	1	1	1	-	-
Clock	1000MHz	200MHz	466MHz	2000MHz	1400MHz	600MHz	600MHz
DVFS Levels	4	1	1	19	13	6	
Memory	LPDDR2			LPDDR3			
OS	Android	SMP/BIOS	SYS/BIOS	Linux	Linux	-	-

Table 6.1: Target MPSoC details

inputs.

The next section describes how we partition the code to generate parametrised code based on a hybrid SPMD model.

6.4 Code generation

In this section, we describe the steps required in code generation. We use a hybrid SPMD model targeting simple OpenMP programs. Depending on the platform, we insert memory flushes to manage memory coherence.

6.4.1 OpenMP

OpenMP is a large pragma-based, shared-memory language that can describe various types of parallelism (e.g., loop, task) and the scheduling of that parallelism to processors. We consider only parallel loop pragmas denoting which loop iterators may be executed concurrently to infer data parallelism and to determine how to partition data and loop iterations across the cores. We ignore any scheduling information as our

runtime policies determine this.

Data Parallelism

We restrict our attention to data parallel programs. These are programs where individual elements of a data structure can be independently computed. They are normally found in array based programs with corresponding parallel loops. Such programs are frequently found in applications and can easily scale. We use information from the OpenMP pragmas and data dependence analysis to determine those array index dimensions that may be exploited in parallel.

We are not concerned in how the data parallelism is determined. This can either be performed by the programmer who inserts a parallel pragma or determined by automatic parallelisation. Instead, we focus on how this potential parallelism can be mapped to a heterogeneous system

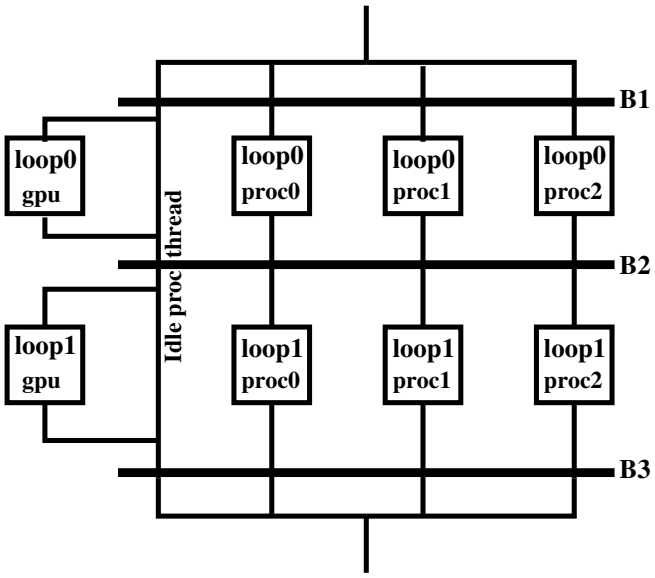


Figure 6.4: Hybrid SPMD. Each processor has an allocated thread allocated which coordinates with other processors via barrier synchronisation. Also, processor 0 has an additional thread that explicitly manages the GPU

6.4.2 Hybrid SPMD

The SPMD (single program, multiple data) model of computation is well-known[43]. Independent tasks execute the same program but operate on independent sections of an array. The array is partitioned across processors such that each processor works on a separate array section synchronising via barriers whenever there is a cross-processor dependence. This model is useful for heterogeneous systems as the same code can

be used for each core. Also, it can be easily mapped to systems that do not have a hardware-based cache-coherent single address space. As we know what data is allocated to each core, we can generate code to manage it.

GPUs add complexity as they are programmed using OpenCL which operates in a master-slave manner, whereas SPMD has cooperative execution and synchronisation. OpenCL barriers can only synchronise with OpenCL threads and cannot be used for inter-processor synchronisation. We use a hybrid programming model (see figure 6.4) where the non-GPU cores run in SPMD and we run an additional thread on the CPU. The job of this thread is to make calls to the GPU in master-slave mode. As can be seen in figure 6.4, the calls to the GPU happen after the idle threads hit the shared barrier and waits until the call returns from the GPU before calling the barrier.

Local Program

Once the data is partitioned, we need to determine the local code to be executed by each core. Each core is restricted to writing to its local data, placing a constraint on the local code's loop bounds. Naive synchronisation would place barriers around each parallel loops. However, in our implementation, synchronisation is only needed whenever there is cross-core dependence. We use the *pthread* library to run threads on the cores.

For OpenCL code generation, we use a modified version of [51] which converts `omp` parallel loops, i.e. loops that are annotated with `omp for` or `omp for reduction`, into `ocl` kernels. Each parallel `omp` loop is translated to a separate kernel using the `ocl` APIs where each iterator is replaced by a global work-item ID. As the Exynos has shared memory, we do not need to copy data to and from the GPU.

6.4.3 Memory Management

The single address space assumed by the SPMD model needs further treatment in heterogeneous cores before emitting code.

Coherence

Barriers ensure the correctness of an SPMD program in cache coherent systems. In systems where caches are not coherent, i.e. OMAP4 and the GPU of Exynos, the compiler is responsible, for inserting cache flushes. These are necessary when a memory location is updated by a core and read later by another. We use dependence analysis to determine the cache lines to flush at barrier synchronisation.

Shared memory across cores

Ensuring shared memory on heterogeneous cores is platform specific. On the OMAP4, memory is allocated in a contiguous area from the Linux kernel. On the A9 side, this area is mapped to the user-space. The physical address corresponding to this area is sent to the M3 and DSP. The memory map table of M3 and DSP for the range of addresses corresponding to this contiguous area is set up such that the physical address is equal to the virtual address ensuring a single address space across cores. On the Exynos, all devices have access to shared memory, and no further support is needed. Memory for the arrays is allocated using the OpenCL function clCreateBuffer. The flag *CL_MEM_ALLOC_HOST_PTR* is used to create buffers that are host accessible. This memory can be mapped and used by the A15/A7 processors.

6.5 Power Constrained Model

In this section, we describe the overall objective of minimising time subject to power. We simplify this into 2-stages: core selection and then partitioning. We exploit power invariance across partitions to simplify the core configurations to consider. We then describe three selection and partitioning policies.

6.5.1 Objective

We wish to minimise parallel execution time, T, subject to a power budget, P_b i.e

$$\min T | P \leq P_b$$

where P is the power used in its execution. Assuming we have n cores, parallel execution time is dominated by the slowest core

$$T = max(T_1 \ldots, T_n)$$

where T_i denotes the execution time on available core, i, while power is the sum of each component: core, memory, interconnect used i.e.

$$P = \sum_{i=1}^{n} (P_i)$$

where P_i denotes the power of each available component. Time is minimised when all processors selected for execution finish at the same time with no load imbalance

$$T_1 = T_2 = \cdots = T_n$$

The time taken for each core i will depend on the program p and the amount of work allocated to it. If the total amount of work is w, then $\sum_i w_i = w$ where w_i is the amount of work per core. As we are dealing with data parallel programs, the amount of work is a simple function of the data size, N.

The objective, then of any policy is to find the set of cores, C, that fit in the power budget and determine the amount of work w_i per core to minimise time.

6.5.2 Selecting a configuration

From the information in Table 6.1, we can see that there are several different ways in which a parallel program can execute on the OMAP/Exynos platform. For e.g., the Exynos platform has 4 BIG A15s with 19 DVFS levels, 4 LITTLE A7s with 13 DVFS levels, 2 Mali GPUs with 7 DVFS levels. This platform has around 100,000 configurations and these configurations have different power consumption values.

Our first task is to find the best configuration (set of cores+DVFS values), that meets the power budget, on which a parallel program can execute.

Partitioning

Once the configuration has been chosen, we have to divide the work among the processors in this configuration. Since the processors are heterogeneous, the work allocated to each processor will be different. Also, partitioning is likely to vary across programs. The various methods used for partitioning are described in the next section.

6.5.3 Policies

This section describes the policies for power constrained code generation used in this work. The policies vary in the amount of information required to make a decision. We present three policies: *Naive*, *Static* and *Profile-directed*.

Intuition

Before describing the policies in detail, the rationale behind the policies is given in this section. Given a program and a power budget, a policy chooses a hardware configuration and a partitioning. This is a difficult problem firstly due to the availability of a lot of hardware configurations and secondly because different programs consume a different amount of power and take a different amount of runtime. To begin with, in the *Naive* policy, we use the average values of the given set of benchmarks. This policy is naive in the sense that it only uses average information and does not use any program specific information. We can see later on in the Results section that using this policy leads to mediocre results. For the next policy, we add some additional program spe-

Policy	Configuration	Partitioning
Naive	Average Power	Average
Static	Average Power	Average + properties
Profile-directed	Individual Power	Throughput + properties
Oracle	Best	Best

Table 6.2: Policies: Configuration and Partitioning methods

cific information. In the *Static* policy, we use the average values of the benchmarks for computing the power consumption of a configuration, but use program specific static information for partitioning. Though the *Static* policy outperforms the *Naive* policy it still gives poor results. To improve the results, we use program specific information to determine the hardware configuration and partitioning in the *Profile-directed* policy and this gives us acceptable results. Table 6.2 summarises the method of selecting the configuration and partitioning for the policies discussed.

Naive

This scheme uses no static or dynamic analysis. By definition, it is applicable to all programs and is the partitioning equivalent of *DVFS* in that it requires no compiler or program knowledge. Any compiler-based scheme must outperform this approach to be worth considering.

The *Naive* scheme is based on average power values and average throughput for a number of benchmarks. It assumes the power used by a core, P_i is the average power seen on m programs p_j:

$$P_i = \sum_{j=1}^{m} P_i(p_j)/m$$

The relative throughput or rate, R_i of a core i is the average time for the slowest core s versus core i averaged over m programs i.e.

$$R_i = \sum_{j=1}^{m} T_s(p_j) / \sum_{j=1}^{m} T_i(p_j)$$

The *Naive* policy searches each of the configurations based on its power approximation to find the configuration which does not exceed the power budget P_b:

$$min\ T(C_k) | P(C_k) \leq P_b$$

i.e. find the configuration that minimises the runtime and satisfies the power budget.

Once the configuration has been selected, we partition the data, N_i for each core i based on the approximate throughput rates

$$N_i = \frac{NR_i}{\sum_{j=1}^{n} R_j}$$

Profile-directed

In this approach, we require the execution time and power of the target program on each of the cores. The cost of the profile runs can be amortised over future uses at any power level and data size.

Rather than estimating the power for each core, we have the actual value P_i. Similarly as we have times, we can more accurately estimate the work rate:

$$R_i = T_s/T_i$$

and determine the core configuration and data partition as before. Also, we use static analysis to alter work allocation based on the observation that synchronisation and cache flushing is more expensive on a slower processor. We redistribute work from the slower processor to the other faster cores. This gives the new work allocation

$$N_i = \frac{NR_i}{\sum_{j=1}^{n} R_j} + k_1(N_{barriers})$$

where $N_{barriers}$ is the normalised number of barriers and cache flushes. The parameter k_1 defines whether work is added or taken away from a core.

$k_1 = R_i/\sum_{j=1}^{n} R_j \times K$ if $R_i > R_s$, in other words, add extra work to the fastest core proportional to its relative speed and remove this amount of work evenly from the remaining $n-1$ cores, i.e. $k_1 = -(R_i/\sum_{j=1}^{n} R_j \times K)/(n-1)$

K represents the relative additional cost of barriers and flushes. and is found empirically and in our experiments is 10 for the OMAP and 8 for Exynos. i.e. flushes and barriers are 10 (8) times more expensive on the slower cores.

Static only policy

While sequential profile runs may be acceptable in some settings where a particular job is to be executed many times over the life of a device, there are occasions when profiling runs are too expensive. This policy tries to improve on the *Naive* policy by using just static analysis without profiling runs.

Like the *Naive* scheme, it uses average power over micro-benchmarks to determine the power of each core:

$$P_i = \sum_{j=1}^{m} P_i(p_j)/m$$

Similarly, it bases the partitioning of work based on average throughput or rate,

$$R_i = \sum_{j=1}^{m} T_s(p_j) / \sum_{j=1}^{m} T_i(p_j)$$

but also, the number of barrier and cache flushes are used to modify work allocation similar to the *Profile-directed* scheme i.e.:

$$N_i = \frac{NR_i}{\sum_{j=1}^{n} R_j} + k_1(N_{barriers})$$

We compare our two proposed polices: *Profile-directed* and *Static* analysis against *DVFS* and the *Naive* partitioning scheme in Section 6.7. We also compare against an *Oracle* - an exhaustive evaluation of all possible processors and partitions plus two other policy variations.

6.6 Setup

In this section, we describe the setup for our experiments: the benchmarks and platform used; how power is measured and the policies evaluated.

6.6.1 Benchmarks

We use data-parallel benchmarks from embedded suites. They are OpenMP versions of benchmarks discussed in Section 2.7. These benchmarks are from the `DSPstone`[113], `UTDSP`[70] and `Polybench`[92] benchmark suites.

6.6.2 Platforms

We target the TI OMAP4430 and Samsung Exynos 5422 MPSoCs in this Chapter. For our experiments we use the OMAP4430 found on a pandaboard and the Exynos 5422 found on an Odroid XU3 board. These MPSoCs are typical heterogeneous multi-core containing different ISA cores with varying power/performance trade-offs. The details of these platforms are presented in Table 6.1. Power measurement on the Exynos is easy as it has accurate hardware counters. On the OMAP it is found by measuring the current consumed by the processor and multiplying by the supply voltage. The OMAP system is mounted on the Panda board which is driven by a supply voltage of 4.2V. We can accurately measure individual power lines for each processor and memory.

6.6.3 Policies evaluated

Section 5 described the main selection and partitioning policies evaluated in this Chapter i.e. *Naive, Profile-directed* and *Static*. For the purpose of our evaluation we consider other possible schemes.

Idealised DVFS

This is the idealised baseline with which we compare all other schemes. Neither platform actually performs DVFS and instead runs at maximum power. To evaluate how DVFS would behave, we assume ideal behaviour i.e., it chooses the voltage and frequency level that gives the maximum performance without exceeding the power budget. OpenMP runs by default on all A15/A7s on Exynos and the 2xA9 cores on OMAP. Idealised DVFS selects the core frequency that meets the power budget. Only a single run of the program is needed for this approach. For the OMAP we assume the OS can lower the voltage/frequency of the A9 CPUs to meet the power budget. On the Exynos platform, there are A15 and A7 cores which can be separately scaled. For this platform, we assume the OS changes the voltage/frequency of each processor in lock-step until there are no more levels left on the A7 to scale. Then on, only the A15 is scaled.

Oracle

On the OMAP, this policy is an exhaustive search of all possible core selection and work partitions which find the configuration that gives the minimum time within a power budget. It gives an upper-bound on performance. On the OMAP we can exhaustively enumerate all options. However on the Exynos, there are simply too many configurations so we sample 3000 points from the space.

Core-type

This policy does not consider partitioning but chooses either the big A15 or LITTLE A7 cores on the Exynos or the 2xA9, 2xM3 or DSP on OMAP as the core type to run an application. It is intended to mimic a big.LITTLE-like runtime scheduler that can select the different heterogeneous cores that satisfy a power budget without altering the code via partitioning. It assumes that a binary for each core type is available.

Table 6.3 lists the policies evaluated and the work required.

Policy	Exynos Work	OMAP Work
Oracle	6881280	16384
Profile Guided	40	6
Core-type	40	6
DVFS	1	1
Static	0	0
Naive	0	0

Table 6.3: Policies: Overhead/Work required

6.6.4 Methodology

For the purpose of this experiment, we assume all the cores on both platforms are available at the time of runtime scheduling but the amount of power available varies. This evaluates all policies over the widest possible configuration space. The selected configuration+partitioning of a policy may exceed the power budget. We assume that hardware or the operating system will step in and downgrade the configuration to a lower one which meets the power budget.

6.7 Results

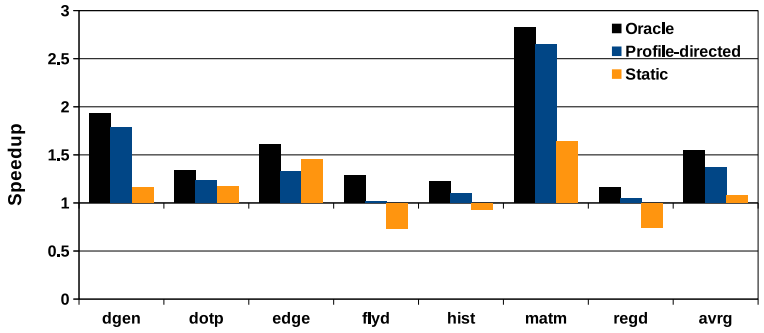
In this section, we first evaluate our two proposed policies *Profile-directed* and *Static* against a *DVFS* baseline and an *Oracle*. Next we evaluate them against alternative policies *Naive* and *Core-type*. This is followed by a summary of all schemes.

6.7.1 Comparison against *DVFS* and Oracle

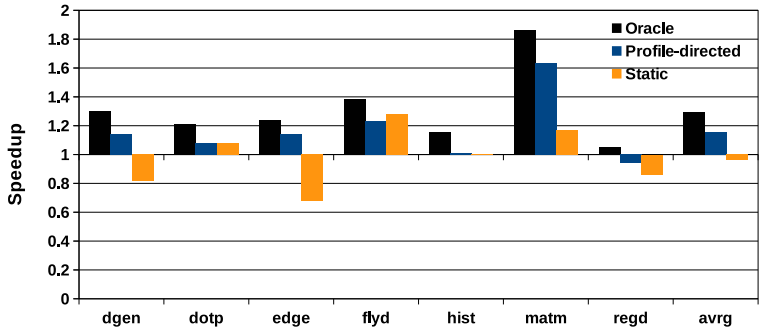
Figure 6.5 compares the speedups obtained for the *Profile-directed* and *Static* policies relative to the *DVFS* baseline on both platforms. We also show the upper-bound of available performance i.e. *Oracle*. Speedup is shown on a per-program basis and is averaged across the entire power range.

OMAP

Figure 6.5a presents the results for OMAP. The *Oracle* policy gives on average a speedup of 1.55 across the benchmarks. The *Profile-directed* policy gives a speedup of 1.37 or 67% of the *Oracle* policy. The *Static* policy performs significantly worse. It achieves just a speedup of 1.07 or only 12% of the *Oracle*. With only static information available, the *Static* policy makes poor partitioning decisions. The amount of improvement varies across the benchmarks. There is significant improvement available on `matm`, 2.8x, which the *Profile-directed* scheme can largely find. For two benchmarks, `flyd` and `regd`, where there is less improvement available, the *Profile-directed* scheme is only able to obtain low speedups while the *Static* scheme slows down relative to the



(a) OMAP: Oracle, Profile-directed and Static Policies gives speedups of 1.55, 1.37 and 1.07 respectively over the DVFS policy. Profile-directed policy provides 67% of the speedup of Oracle, but the Static policy is only able to achieve 12%.



(b) Exynos: Oracle, Profile-directed and Static Policies gives speedups of 1.30, 1.15 and 0.97 respectively over the DVFS policy. Profile-directed policy provides 52% of the speedup of Oracle. The static policy is unable to achieve a speedup on average.

Figure 6.5: Speedups of Oracle, Profile-directed and Static Policies.

DVFS baseline. In one case, `edge`, the *Static* approach outperforms *Profile-directed*. This is due to a suboptimal core selection rather than partitioning decision.

Exynos

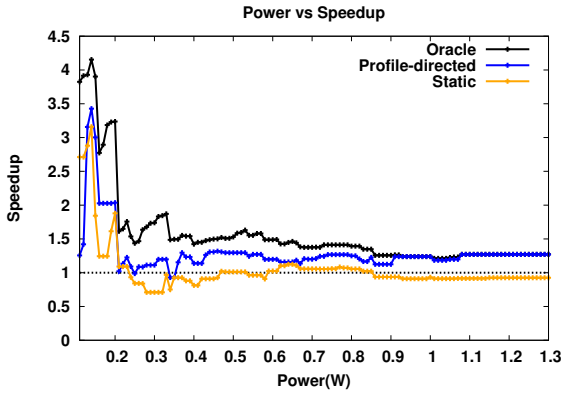
Figure 6.5b presents the results for Exynos. The *Oracle* policy achieves on average a speedup of 1.30 across the benchmarks. The *Profile-directed* policy achieves a speedup of 1.15 which is around 52% of the speedup obtained by the Oracle. The other policies are unable to achieve a speedup. The *Static* policy is only able to obtain a slowdown of 0.97. Like with OMAP, there is significant improvement available for `matm`, 1.86x, which the *Profile-directed* scheme can largely find. `regd` is the only benchmark where the *Profile-directed* policy is slower than *DVFS*. This happens because the best configuration is without the GPU and the best partitioning is surprisingly one which assigns a larger partition to the LITTLE A7 cores than the A15 cores. Single thread performance for this benchmark is better for the BIG A15 core and hence *Profile-directed* policy assigns a larger partition size to the A15. The *Static* policy achieves speedups only for `dotp`, `flyd` and `matmul`. The *Static* policy achieves a higher speedup than *Profile-directed* policy for the *flyd* benchmark. This happens due to poor partitioning decision by the *Profile-directed* policy caused due to the difference in sequential and parallel behaviour of the *flyd* benchmark.

Analysis across the power range

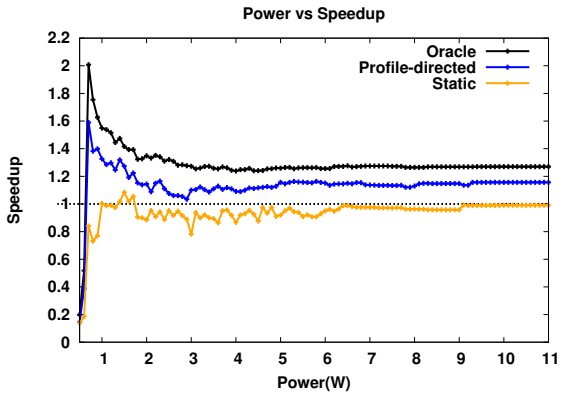
Figure 6.6a and Figure 6.6b shows the speedup of the three policies obtained over the entire power range. Each line denotes speedup averaged across all benchmarks. All the policies (except *Static* for Exynos) obtain high speedups in the lower ranges of power over the DVFS policy. This is because low-power GPU/M3 and DSP processors perform well in this power range. For OMAP, at the higher power ranges the Profile_directed policy is competitive with the *Oracle* policy. Here the most powerful configuration is always chosen, and usage of profile information provides good partitioning decisions. For Exynos, the *Profile_directed* policy achieves speedups but is not able to achieve the speedups of the *Oracle* policy. For both platforms, the *Static* policy consistently under performs when compared to the *Oracle*.

6.7.2 Comparison to Naive and Core selection

While our *Profile-directed* scheme performs better than *Oracle*, it may be the case that simpler heuristics perform well. Here we evaluate against the two other policies, *Naive* and *Core-type*, which are schemes not requiring any program knowledge.

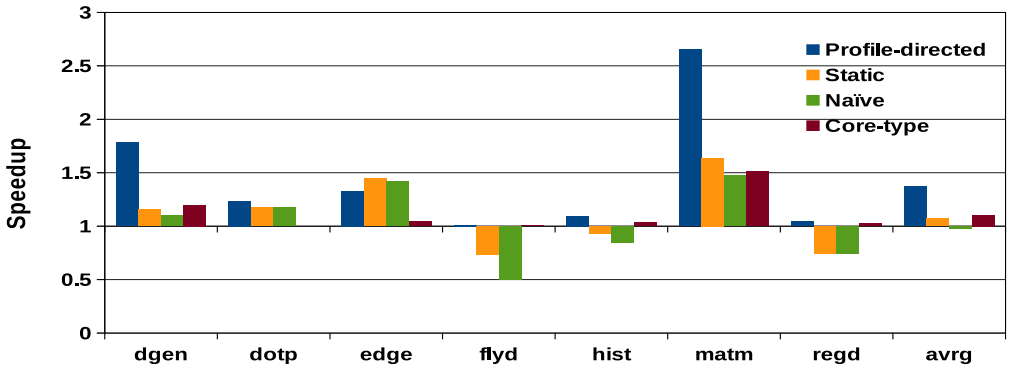


(a) OMAP4: High speedups are obtained in the low power range. *Profile-directed* policy matches the speedup of *Oracle* in the higher power ranges. The *Static* policy is consistently poor.

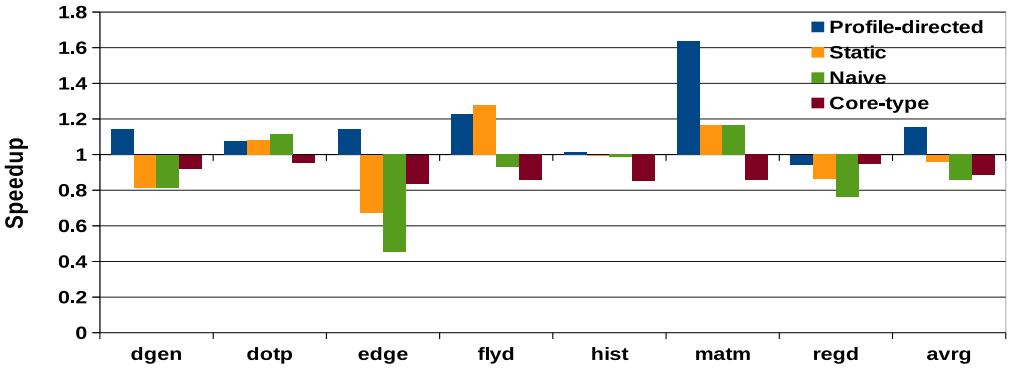


(b) Exynos: High speedups are obtained in the low power range. The speedups obtained by *Profile-directed* policy is consistently lower than that obtained by the *Oracle*. The *Static* policy performs poorly and is only able to provide a speedup at the highest power levels.

Figure 6.6: Whole power range speedup comparison of *Profile-directed*, *Oracle* and *Static* policy.



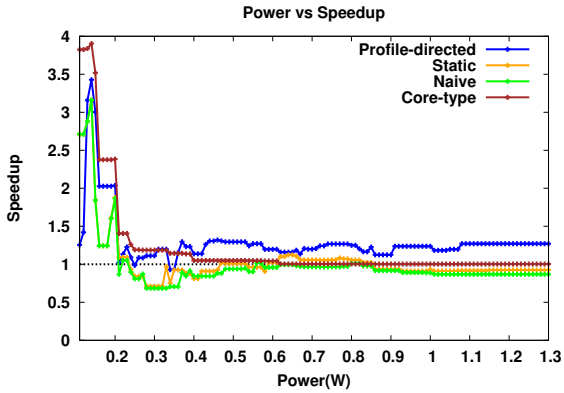
(a) OMAP4: On average the *Profile-directed* policy is the best policy. The *Core-type* policy can achieve small speedups in all the benchmarks and is slightly better than the *Static* policy and provides a speedup of 1.1 on average. The *Naive* policy is the worst and gives a slowdown even-though it achieves a speedup in four of the benchmarks.



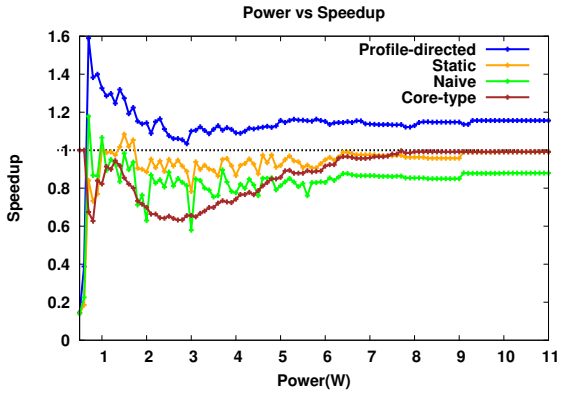
(b) Exynos: Only the *Profile-directed* policy can achieve a speedup. The *Naive* policy can achieve speedups in two benchmarks, but on average it only achieves a slowdown. The *Core-type* policy is not able to achieve a speedup over *DVFS* in any of the benchmarks. On average the *Core-type* policy is slightly better than the *Static* policy.

Figure 6.7: Speedup of *Profile-directed*, *Static*, *Core-type* and *Naive* policy over *DVFS*.

Figure 6.7a compares the speedups obtained for the *Profile-directed*, *Static*, *Naive* and *Core-type* policy for the OMAP platform. On average the *Profile-directed* policy remains the best policy. The *Naive* policy performs poorly, particularly on the `flyd`, `hist` and `regd` benchmarks and gives a slowdown. The *Core-type* policy is slightly better than the *Static* policy and provides a speedup of 1.1. The *Naive* policy performs poorly, particularly on the `flyd`, `hist` and `regd` benchmarks and gives a slowdown. Thus, while the *Oracle* shows there is significant performance available when using core selection and partitioning, if this is done blindly without program knowledge, then performance is worse than relying on *DVFS* alone. The *Core-type* scheme shows that using heterogeneity (DSP and M3) to meet power budgets is better than *DVFS* for runtime performance.



(a) OMAP4: The *Core-type* policy performs well in the low power range. The *Naive* policy performs poorly and sometimes does not even have half the speed of the DVFS policy.



(b) Exynos: The *Core-type, Static* policies are unable to achieve speedups and is only able to match *DVFS* at the higher power levels. The *Naive* policy performs poorly in the entire power range.

Figure 6.8: Whole power range comparison of *Core-type* and *Naive* policy.

Figure 6.7b compares the speedups obtained for the *Profile-directed*, *Static*, *Naive* and *Core-type* policy for the Exynos platform. On average the *Profile-directed* policy remains the best policy. Other policies *Static*, *Naive* and *Core-type* are unable to obtain a speedup. Only the *Profile-directed* policy can achieve a speedup. The *Naive* policy can achieve speedups in two benchmarks, but on average it only achieves a slowdown. The *Core-type* policy is not able to achieve a speedup over *DVFS* in any of the benchmarks. On average the *Core-type* policy is slightly better than the *Static* policy

Analysis across the power range

Figure 6.8a and Figure 6.8b shows the speedup of *Static*, *Profile-directed*, *Naive* and *Core-type* policy across the entire power range. For OMAP, the *Core-type* policy performs well in the low power range, as it can select the M3 and DSP, unlike DVFS. However, it is slightly worse than the *Static* policy in the middle power ranges. At high power values, it performs better than the *Static* policy. For both platforms, except in the low power range, the *Naive* policy performs poorly. This is due to poor partitioning decisions. For Exynos, all policies work better at higher power values. At higher power values, all the cores can be activated, and there is no loss of performance due to poor configuration selection.

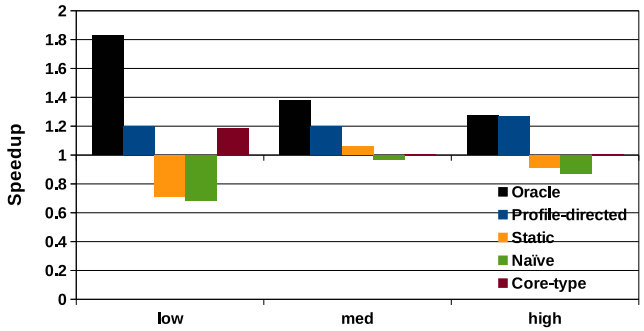
6.7.3 Speedup Summary

To summarise the interaction between power range, performance and policy, we selected 3 power values, low, middle and high (from Figure 6.1a)) and summarised the performance of each policy across the benchmarks. Figure 6.9 shows the average speedup obtained.

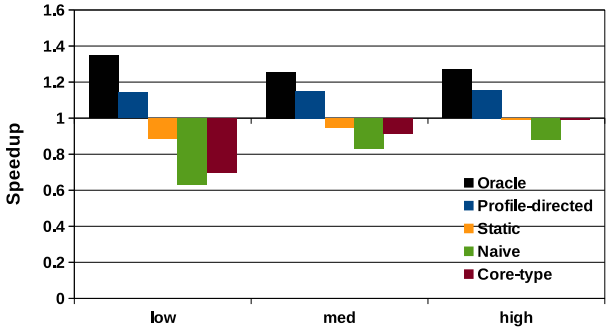
OMAP

We can see that the *Oracle* provides high speedup at low power, 1.8x, but other policies are not able to match this. As we have seen before, at high power values the *Profile-directed* policy is competitive with the *Oracle*. The *Static* policy performs best at the middle power value. The *Naive* policy is poor in all settings reinforcing the need to consider program information in partitioning decisions. At low power values the *Core-type* policy can make use of the low-power M3 and DSP processors and hence obtains good speedups. But at higher and middle power values it performs similarly to *DVFS* and loses any advantage.

Exynos



(a) OMAP4: The *Profile-directed* policy achieves similar speedups at all power levels. The *Static* policy achieves moderate speedups in the middle power level. The *Core-type* policy performs well at low power levels, but badly at medium and high power levels.



(b) Exynos: The *Profile-directed* policy achieves similar speedups at all power levels. The *Static, Core-type* policies performs poorly but can match *DVFS* at middle and high power levels.

Figure 6.9: Speedups at low, mid and high power levels

For Exynos, the low, middle and high power values are 2, 6 and 10 W. From the results for the *Oracle* policy, we can see that the largest speedups are available at low power. The *Profile-directed* policy performs uniformly across all three power levels. The other policies perform better at the higher power levels. Choosing the hardware configuration becomes easy at the higher power levels. Since the power required for all the cores is available, all policies will pickup all the cores with their highest DVFS values. Hence, there is not much chance for Over/Undershooting. The lack of speedup at high power values is due to poor partitioning decisions of the *Naive* and *Static* policies. The *Core-type* policy meets the power budget by varying the type and number of A15/A7 cores. At low power values, the power hungry A15 cores cannot be used. Hence, this policy meets the budget by varying the number of the small A7 cores. This limits its ability to exploit the parallelism in the benchmarks and hence is not competitive with other policies. At the highest power levels, the *Core-type* policy can activate all the cores and achieve the performance of the *DVFS* policy.

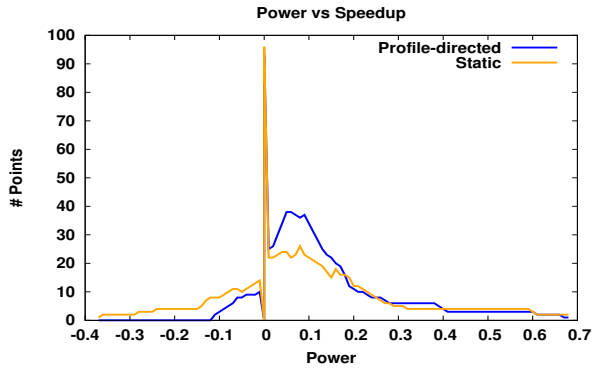
6.7.4 Over/Under Shooting

A good policy should try to find the fastest configuration that meets the budget. Selecting configurations which do not use all the available power (undershooting) or which exceeds the budget (overshooting) will lead to sub-optimal performance.

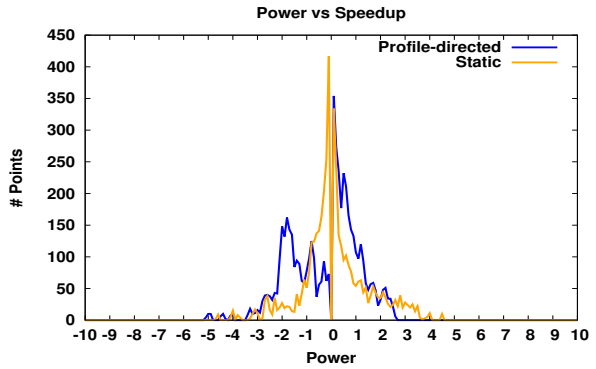
Undershooting: If a policy selects configurations that are much lower than the power budget then the system is under-performing, since the system could have used the extra power available to choose a configuration that is faster. Hence, when a policy is undershooting, it is choosing configurations that are slower than the best possible ones and this leads to a loss of speedup.

Overshooting: The power budget imposes a hard limit on the power consumption. If a policy selects configurations that exceed the power budget, this configuration cannot be allowed to run. The OS intervenes and selects a lower power configuration by using DVFS. The OS intervention is a runtime overhead, and the lower power configuration (which is very likely non-optimal) is slower than the best possible configuration. These two factors lead to a loss of speedup.

Figure 6.10a and Figure 6.10b shows the summary of undershooting and overshooting for the *Static* and *Profile-directed* policies. In these Figures, the x-axis shows the range of overshooting/undershooting. The x-axis value is positive for overshooting and negative for undershooting. A good policy will have most of its values around zero and a bad policy will have more values away from zero. On the y-axis, we have the



(a) OMAP: The *Static/Naive* policy has a higher degree of undershooting compared to the *Profile-directed* policy.



(b) Exynos: The over and undershooting profiles look similar but the extent of overshooting is more for the *Static/Naive* policy.

Figure 6.10: Overshooting vs. Undershooting.

number of points which exhibited that particular value of overshooting or undershooting. For both platforms, we can see that most of the values are clustered around zero. But the *Static* policy has higher undershooting and overshooting for the OMAP and Exynos platforms respectively. This is one reason for the poor performance of *Static* policy. For e.g., the *Static* policy has the worst undershooting value of -0.35 while *Profile-directed* policy has the worst value of -0.12 for the OMAP platform.

6.8 Summary

Processor platforms of the future will be heterogeneous, composed of different kinds of processors and accelerators like the MPSoCs of today. While these platforms offer excellent power and performance trade-offs, they will be very difficult to program. Due to dark silicon effects, all the processors on a platform cannot be turned ON at the same time. Hence, we will have to choose a set of processors based on a power budget.

This Chapter has presented a compiler-based approach for improving program performance on heterogeneous multi-cores in the presence of a power budget. Our approach is based on determining the best hardware configuration from the cores available and partitioning the work across processors based on a data parallel code generation scheme. We evaluate a number of policies and show that our *Profile-directed* approach obtains a speedup of 1.37X and 1.15X over traditional *DVFS*.

This chapter ends the final contribution of the thesis. Next Chapter will conclude and present ideas for future work.

Chapter 7

Conclusion

Future processor platforms are likely to contain many cores. Due to power limitation, all these cores cannot be switched ON at the same time. Hence multicore designs will be heterogeneous with cores specialised to better utilise the available power. Prior research has shown that heterogeneous processors are better than homogeneous ones when considering power-performance trade-offs. Recent research has demonstrated the advantages of multiple ISA heterogeneity over single ISA. This thesis has investigated issues in mapping data-parallel programs to multiple ISA heterogeneous processor systems. Results of a detailed design space exploration were presented in Chapter 4. The results confirm that partitioning programs for multiple ISA heterogeneous processors is beneficial. Chapter 5 presented a compiler approach for automating the partitioning of data-parallel programs. Chapter 6 presented a compiler based method to generate code that meets power budgets for multiple ISA heterogeneous systems.

This chapter has the following structure. Section 7.1 presents the contributions of this thesis. A critical analysis of the work in this thesis is presented in Section 7.2. Finally, Section 7.3 presents ideas for future work.

7.1 Contributions

7.1.1 Partitioning

A design space exploration in the partitioning of data-parallel programs for heterogeneous processors is presented in Chapter 4. To perform the design space exploration, a framework for running programs in parallel on the OMAP4 framework is presented. The Chapter also describes a method for measuring the energy consumption of programs. A simple method for partitioning programs for runtime and energy is presented. The partitioning approach is within 10% of the best partitioning schemes across all optimisation criteria. On average, a 2.2x speedup and a 1.45X energy improvement are

achieved over sequential execution on the ARM A9 CPU.

7.1.2 Compiler Framework

A compiler based approach for mapping data-parallel OpenMP programs to heterogeneous embedded systems is presented in Chapter 5. OpenMP source programs are taken as input and using a source to source translator (based on Clang), the programs are converted to SPMD format. The target used is OMAP4 by Texas Instruments. Texas Instruments provides a library for programming heterogeneous processors called Syslink/RPMsg. It is shown that using the SPMD model of compilation offers better performance than the existing Syslink model. On average, the SPMD approach achieves a 2.75x speedup over Syslink. Experiments show that smart insertion of cache flushes is important for achieving good performance. A speedup of 1.38x and an improved energy efficiency of 1.4x are achieved over using the CPU cores alone. All these improvements are achieved without the user having to make any code changes.

7.1.3 Power Budget

Dark silicon effects are predicted to be a major issue in the future. This will cause processors to have power budgets. Typically, the role of meeting power budgets is left to the Operating System or hardware by using Dynamic Voltage and Frequency (DVFS) scaling. Heterogeneity presents a new dimension for meeting power budgets. Chapter 6 presents a compiler-based approach for achieving good performance on heterogeneous multi-cores in the presence of a power budget. The approach is based on determining the best hardware configuration from the cores available and partitioning the work across processors based on a data parallel code generation scheme. Experiments are performed on two platforms, OMAP4 and Exynos. Some policies are evaluated, and it is shown that the *Profile-directed* approach obtains a speedup of 1.37X and 1.15X over traditional *DVFS*.

7.2 Critical Analysis

This section presents a critical analysis of the work presented in this thesis. Section 7.2.1 discusses issues that are common to the entire work presented in this thesis. Section 7.2.2, Section 7.2.3 and Section 7.2.4 presents a critical analysis and drawbacks of the work in Chapter 4, Chapter 5 and Chapter 6.

7.2.1 Common Issues

All the benchmarks used for experiments in this thesis are simple, regular data-parallel benchmarks. While the approach described works for regular array based programs, it is unlikely to work for real world programs with dynamic memory allocation and pointers. This thesis focuses on data-parallel programs and excludes other forms of parallelism like stream and task based parallelism. The work presented in this thesis supports only a single benchmark at a time. If more than one benchmark is active at the same time, then, the computed partitions are not balanced anymore, and this can lead to poor results. The partitioning work presented requires knowledge of the hardware since the partitioning values are determined by running the program on each of the processors. The SIMD units on the ARM processors and the vector units of the GPUs are not used. Data-parallel benchmarks can be vectorised and hence it might perform better on the ARM Neon vector unit and the vector GPU. The SIMD units are likely to consume more power since they use the processor pipeline, but will end up using lesser energy since the code gets executed faster. Vectorised code is unlikely to use more power since the GPU executes scalar code on one of the units in the vector. Considering the vector units would have opened a new dimension for the design space exploration and would have led to a more holistic study.

7.2.2 Partitioning

The *iter* method for partitioning used in this work (Chapter 4) does not take advantage of any static information. It runs the benchmark on each processor to compute the partitioning. This task has to be repeated for each benchmark and also might be necessary if data size changes. Using static features in addition to this method can reduce the need to perform these runs for each benchmark and each platform. Oracle values are found by exhaustive search at a coarse level granularity. For e.g., if the size of the array to be partitioned is 1024, then the array is divided into 16 units of 64 size, and runtime and energy values are computed for each partition value at this granularity. This process of finding Oracle value is not feasible when the number of processors increases or at higher granularities. Though heterogeneity is considered for energy measurement, DVFS values are not considered. The framework is very OMAP specific. It is not clear whether the method can extend to other platforms. While the SPMD model is easily portable to other platforms, the approach for accessing memory from remote processors without copying is platform specific. A single partitioning is used for all the loops in a benchmark. The loops themselves can have different characteristics and hence

have different performance. This means loop specific partitioning values are possibly better.

7.2.3 Compiler Framework

The method described (in Chapter 5) uses a source to source translation. Sometimes, all the optimisation opportunities are not available at the early stages of the compiler. Better optimisation and performance can be achieved if the translation is performed at a lower level in the compiler. The SPMD based approach described in this work does not directly extend to GPUs. This is subsequently handled in Chapter 6. Only row wise partitioning is supported. By default, programming languages like C support row major form. Hence, for supporting row partitioning, the only necessary condition is that arrays should be located in physically contiguous locations. Cache flushing is simplified in this case since partitions are located in contiguous locations. If the parallelism demands a column based partitioning, then successive columns are not in contiguous locations and cache flushing can be costly. The data size of the benchmarks is not varied. It is not clear whether the results hold true when data size changes, particularly the degradations observed for Syslink.

7.2.4 Power Budget

Though the *Profile-directed* method presented in Chapter 6 performs better than DVFS; it is not competitive with Oracle. For e.g., Only 67% of the performance of the Oracle has been achieved for OMAP, and only 52% of the performance of the Oracle has been achieved for Exynos. The Oracle values are computed by using a local search of neighbouring points of all the other methods. This method has all the disadvantages of using local searches, like getting stuck at local minima. The implementation considers changes in DVFS values to be instantaneous, but in reality, this is not so. A good implementation should factor in the delay due to DVFS.

7.3 Future work

Future work is summarised in this section.

7.3.1 Scale

Only a limited scale work has been performed in this thesis. The current benchmarks used are small and they are mostly data-parallel. It will be interesting to see whether similar techniques can be used for complex programs. Future work can also consider mapping other forms of parallelism such as tasks and streams to heterogeneous plat-

forms. It will also be interesting to see this framework extended to other platforms like Snapdragon from Qualcomm, Tegra from Nvidia, etc.

Task Parallelism

Task parallelism can only be supported by having a runtime for scheduling tasks to the heterogeneous processors. Task-parallel systems should have support for creating and specifying dependence between tasks, and the ability to schedule task graphs to processors. OpenMP version 4.0 supports task parallelism, but the existing runtimes are for homogeneous systems. The runtime system will have to be extended to support heterogeneous systems. The following additional capabilities are required. 1) In homogeneous systems, the runtime can schedule a task to any of the free cores or processors. In a heterogeneous system, the runtime has to decide which processor can execute the given task the fastest or at the highest energy efficiency. Such a decision system can be created by using machine learning techniques. 2) The lack of cache coherency in some of the heterogeneous systems requires that in addition to the sequencing of dependent tasks if the dependence is caused due to arrays then the caches have to be flushed and invalidated. Techniques presented in this thesis can be extended to handle cache flushes and invalidations.

7.3.2 Partitioning

The partitioning methods presented are either static or dynamic. Static methods suffer from performance issues and are not able to handle load imbalance issues due to other workloads running or due to inherent load imbalance issues. Dynamic or profile based methods suffer overhead at runtime. It remains to be seen whether machine learning techniques can be used to improve the static methods. One requirement for machine learning is a large number of benchmarks. Only a few data-parallel benchmarks were available. Since the DSP and M3 processors on OMAP did not support floating point in hardware, partitioning experiments used only integer benchmarks. Future platforms are unlikely to have these drawbacks, and more benchmarks can be used on these platforms to build a machine learning model.

Handling Load Imbalance

Some programs have inherent load imbalance issues. Load imbalance can also arise due to other workloads running on the system. These kinds of programs are handled poorly by the work described in this thesis. This problem can be solved by finely partitioning tasks and mapping to each processor with work-stealing queues. In these

kinds of systems, if due to load imbalance a processor is idle while another processor is loaded then the idle processor can steal tasks from the loaded processor's queues. However, work-stealing queues are difficult to implement for the kind of systems that are targeted in this thesis. This is primarily due to three reasons. i) Lack of cache coherence. ii) Heterogeneity and presence of multiple binaries. iii) Multiple address spaces. If caches are not coherent, the process of probing and deqeueing the task from another processor's queue will be runtime heavy. Since the processors are heterogeneous, there exist different binaries for each processor. For enqeueing the stolen task, a different binary version of the task might have to be fetched from memory. Since the address spaces of processors are different, the addresses of the arrays in the task might have to be converted to enable the task to work on the new processor. Systems like HSA[98] already support cache-coherent heterogeneous systems with single address space. This is likely to be the direction that the industry is going to take, and hence, problems (i) and (iii) will be automatically solved. Problem (iii) can be solved by using a common language interface like OpenCL.

7.3.3 Oracle Computation

Oracle values are currently obtained by performing an exhaustive design space exploration. Heterogeneous processors like OMAP4 have limited hardware configurations and hence it is possible to perform an exhaustive design space exploration. New MPSoCs and heterogeneous processors have a lot of hardware configurations (number of cores and DVFS values). As the number of hardware configurations increase, it becomes difficult to compute the Oracle (best obtainable) values. A faster method is needed to compute the Oracle values.

7.3.4 Multiple Programs

The work in this thesis has focused on partitioning a single program to run across all the processors. However, in most systems, multiple programs are active at the same time. It will be interesting to see how multiple programs are handled. This is difficult because partition values cannot be changed at runtime and when other programs are running, the partition values are no longer optimal. It will also be interesting to see how power budgets are met when there is more than one application running.

7.3.5 Mobile Devices

MPSoCs are widely used in mobile devices. Mobile devices have a lot of interactive applications and few background applications. It will be interesting to concentrate on

these kinds of workloads and study how the energy usage can be optimised. Other interesting cases would be to decide the hardware configuration based on the power supply. For e.g., if using battery power then use low power hardware configuration, if connected to the supply then use high power hardware configuration etc.

7.4 Summary

Future general purpose multicore processors will be heterogeneous. This presents a lot of challenges as well as opportunities for creating smart software to map parallelism and workloads to the best processor capable of execution. This thesis explored the partition of data-parallel programs for runtime and energy. It was shown that partitioning data-parallel programs is beneficial for heterogeneous processors. A compiler framework for automatic partitioning was also presented. The thesis also demonstrated that compiler techniques can be used to meet a power budget by choosing a hardware configuration and partitioning data-parallel programs for that configuration.

The Chapter presented a summary of the contributions as well as a critical analysis of the contributions. Ideas for future work were also presented. This Chapter concludes the thesis.

Appendix A

Experiments with Snapdragon Platform

During this study, partitioning experiments were performed on other platforms as well. In this Chapter, details of the experiments using the Snapdragon Platform are discussed.

A.1 Snapdragon

The Dragonboard development kit based on the Snapdragon 800 processor (APQ 8074) was used to perform partitioning experiments.

A.1.1 Hardware

Snapdragon has four Krait CPU cores, a Hexagon DSP and an Adreno GPU. The Krait 400 CPU has 4 cores and a maximum clock frequency of 2.7 GHz. The Hexagon QDSP6V5A DSP has 3 threads and a maximum clock frequency of 600MHz. The Adreno 330 GPU has 128 shader cores and runs at 450 MHz.

A.1.2 Runtime

The Dragonboard runs the Android Operating System on the CPU. Qualcomm's RTOS runs on the Hexagon DSP. Since the processors have their own Operating Systems and address spaces, a mechanism is needed to pass around pointers so that all the processors can work on the same arrays. Contiguous physical memory is allocated using the ION memory allocator on Android. This memory is then memory mapped to the CPU virtual address space. The Hexagon DSP is programmed through an SDK provided by Qualcomm. This SDK provides an IDL (Interface Description Language) for specifying the parameters to functions that are to be executed on the DSP. This IDL provides a provision for specifying the parameter as *in*, *out* or *inout*. The runtime inserts functions to flush the caches depending on the type of the parameters. For e.g., if the parameter is *inout*, then caches for that parameter are flushed before and after the

function is executed. For the GPU, buffers are created from the host memory pointer using the *CL_MEM_USE_HOST_PTR* and *CL_MEM_EXT_HOST_PTR_QCOM* flags in clCreateBuffer function.

A.1.3 Programming Model

A Hybrid SPMD model as described in Chapter 6 is used for running code in parallel across all the processors. pthreads are used on the CPU; custom threads are used on the Hexagon DSP, and OpenCL is used for the CPU.

A.2 Experiments

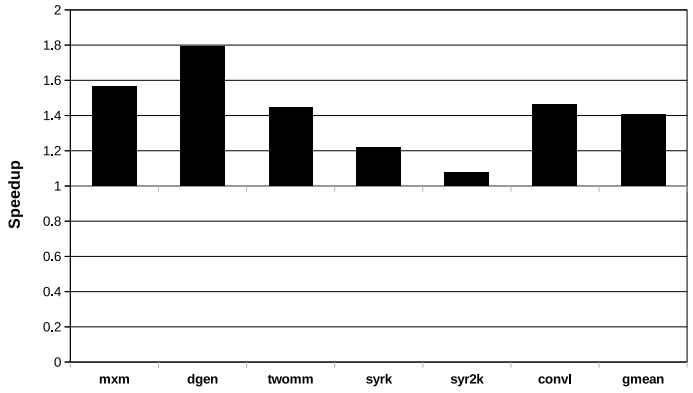


Figure A.1: Speedup over execution on the CPU with 4 threads

Partitioning experiments were performed for six benchmarks taken from the polybench benchmark suite. As Figure A.1 shows, on average, these benchmarks achieve a speedup of 1.4 over parallel execution on the CPU. These benchmarks are purely data-parallel without any synchronisation. Benchmarks like *flwl* and *hist* (not shown here) etc. with synchronisation did not perform well on this platform, and it was not able to achieve significant speedups. Speedups could not be achieved due to the lack of fine control over cache flushing and also in the mismatch in performance of the CPU and DSP/GPU. The A15 based CPU performs exceptionally well on these benchmarks and hence it makes it less desirable to partition the programs for heterogeneous execution. Power could not be measured on this platform since the board did not offer such a feature. It was also not possible to measure power using the Oscilloscope method described in Chapter 4 since the resistors were in an inaccessible position.

A.3 Summary

Partitioning experiments demonstrate that speedups can be obtained on the Snapdragon platform for pure data-parallel programs. For benchmarks with inter-processor dependence, speedups could not be attained.

Appendix B

Bug fix in Linux Kernel

It was not straightforward to get all the processors running on the OMAP4 board. There was a bug in the Linux Kernel, which needed to be fixed. Programs were crashing with a SIGBUS error (”Command terminated by signal 7”). The crash was subsequently isolated to the case where more than one threads are running on the CPU, and the remote processors are also used. When all the processors have to work on the same buffer, memory (GEM buffer) was allocated in the graphics memory region. After allocating this memory, the memory has to be mapped to the CPU's virtual memory. The mapping and creation of page table entries happen on demand. Consider the case when there are two threads working on the CPU. When the first thread accesses a location, a fault is generated, and the OS tries to map that page. Before this request gets completed the second thread also accesses the location, and a fault is generated, and the OS tries to map that page. Now the OS realises that the mapping is already being performed on behalf of the first thread and returns EBUSY. But this EBUSY case was not properly handled in the fault handler code of the OMAP driver and returns a SIGBUS error and the application crashes.

The bug fix listing is given below.

```
Date:    Sun, 20 Oct 2013 12:07:42 -0400 [10/20/2013 05:07:42 PM GMT]
From:    Rob Clark <robdclark@gmail.com
To:      dri-devel@lists.freedesktop.org
Cc:      Kiran <s1064296@sms.ed.ac.uk>
Subject:     [PATCH] drm/omap: EBUSY status handling in omap_gem_fault()

Subsequent threads returning EBUSY from vm_insert_pfn() was not
handled correctly. As a result concurrent access from new threads
to mmapped data caused SIGBUS.

See e79e0fe3

Signed-off-by: Rob Clark <robdclark@gmail.com>
---
```

```
 drivers/gpu/drm/omapdrm/omap_gem.c | 5 +++++
 1 file changed, 5 insertions(+)

diff --git a/drivers/gpu/drm/omapdrm/omap_gem.c b/drivers/gpu/drm/omapdrm/omap_gem.c
index 533f6eb..435c6b1 100644
--- a/drivers/gpu/drm/omapdrm/omap_gem.c
+++ b/drivers/gpu/drm/omapdrm/omap_gem.c
@@ -542,6 +542,11 @@ fail:
         case 0:
         case -ERESTARTSYS:
         case -EINTR:
+        case -EBUSY:
+                /*
+                 * EBUSY is ok: this just means that another thread
+                 * already did the job.
+                 */
                 return VM_FAULT_NOPAGE;
         case -ENOMEM:
                 return VM_FAULT_OOM;
--
1.8.3.1
```

Listing B.1: Kernel bug fix

Bibliography

[1] Rpmsg. http://omappedia.org/wiki/Category:RPMsg, 2012. [Online; accessed 3-Dec-2015].

[2] Cuda. http://www.nvidia.com/object/cuda_home_new.html, 2015. [Online; accessed 3-Dec-2015].

[3] Exynos soc, samsung. http://www.samsung.com/semiconductor/minisite/Exynos/w/, 2015. [Online; accessed 3-Dec-2015].

[4] The llvm compiler infrastructure. http://llvm.org, 2015. [Online; accessed 3-Dec-2015].

[5] Mare. http://developer.qualcomm.com/mare, 2015. [Online; accessed 16-Dec-2015].

[6] Omap soc, ti. http://www.ti.com/lsds/ti/omap-applications-processors/technologies.page, 2015. [Online; accessed 3-Dec-2015].

[7] Omap4430. http://www.ti.com/product/OMAP4430, 2015. [Online; accessed 3-Dec-2015].

[8] Ompss programming model. https://pm.bsc.es/ompss, 2015. [Online; accessed 3-Dec-2015].

[9] Openacc. http://openacc-standard.org, 2015. [Online; accessed 16-Dec-2015].

[10] Opencl 2.0. http://www.khronos.org/opencl/, 2015. [Online; accessed 3-Dec-2015].

[11] Openmp clang compiler. http://clang-omp.github.io/, 2015. [Online; accessed 3-Dec-2015].

[12] Pandaboard. http://pandaboard.org, 2015. [Online; accessed 3-Dec-2015].

[13] Snapdragon soc, qualcomm. http://www.qualcomm.com/snapdragon, 2015. [Online; accessed 3-Dec-2015].

[14] Tegra soc, nvidia. http://www.nvidia.com/object/tegra.html, 2015. [Online; accessed 3-Dec-2015].

[15] Xilinx ultrascale mpsoc. http://www.xilinx.com/products/technology/ultrascale-mpsoc.html, 2015. [Online; accessed 16-Dec-2015].

[16] David Abdurachmanov, Peter Elmer, Giulio Eulisse, Robert Knight, Tapio Niemi, Jukka K Nurminen, Filip Nyback, Gonalo Pestana, Zhonghong Ou, and Kashif Khan. Techniques and tools for measuring energy efficiency of scientific software applications. *Journal of Physics: Conference Series*, 608(1):012032, 2015.

[17] Anant Agarwal, David A. Kranz, and Venkat Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 6(9):943–962, September 1995.

[18] ARM. Arm big.little. http://www.arm.com/products/processors/technologies/biglittleprocessing.php, 2015. [Online; accessed 3-Dec-2015].

[19] P.E. Bailey, D.K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B.R. de Supinski. Adaptive configuration selection for power-constrained heterogeneous systems. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 371–380, Sept 2014.

[20] R. Bakker, M.W. Van Tol, and A.D. Pimentel. Emulating asymmetric mpsocs on the intel scc many-core processor. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 520–527, Feb 2014.

[21] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *ACM/IEEE CONFERENCE ON SUPERCOMPUTING*, page 86. ACM, 2006.

[22] W.L. Bircher and S. Naffziger. Amd soc power management: Improving performance/watt using run-time feedback. In *Custom Integrated Circuits Conference (CICC), 2014 IEEE Proceedings of the*, pages 1–4, Sept 2014.

[23] Franois Bodin and Michael OBoyle. A compiler strategy for shared virtual memories. In BoleslawK. Szymanski and Balaram Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 57–69. Springer US, 1996.

[24] F.A. Bower, D.J. Sorin, and L.P. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *Micro, IEEE*, 28(3):17–25, May 2008.

[25] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 162–175, New York, NY, USA, 1986. ACM.

[26] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 225–236, Washington, DC, USA, 2012. IEEE Computer Society.

[27] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[28] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. Maps: An integrated framework for mpsoc application parallelization. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 754–759, New York, NY, USA, 2008. ACM.

[29] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM.

[30] Kiran Chandramohan and Michael F. P. O'Boyle. A compiler framework for automatically mapping data parallel programs to heterogeneous mpsocs. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, pages 9:1–9:10, New York, NY, USA, 2014. ACM.

[31] Kiran Chandramohan and Michael F.P. O'Boyle. Partitioning data-parallel programs for heterogeneous mpsocs: Time and energy design space exploration. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 73–82, New York, NY, USA, 2014. ACM.

[32] Andrew A. Chien, Allan Snavely, and Mark Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science*, 4:1987 – 1996, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[33] Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, P K. Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, Suchit Subhaschandra, Sabina Grover, Xiaowei Jiang, and Ravi Iyer. Quickia: Exploring heterogeneous architectures on real prototypes. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–8, Washington, DC, USA, 2012. IEEE Computer Society.

[34] Lynn Choi and Pen Chung Yew. A compiler-directed cache coherence scheme with improved intertask locality. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, Supercomputing '94, pages 773–782, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[35] N.K. Choudhary, S.V. Wadhavkar, T.A. Shah, H. Mayukh, J. Gandhi, B.H. Dwiel, S. Navada, H.H. Najaf-abadi, and E. Rotenberg. Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 11–22, June 2011.

[36] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & cap: Adaptive dvfs and thread packing under power caps. In *Proceedings of the 44th*

Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, pages 175–185, New York, NY, USA, 2011. ACM.

[38] Jason Cong and Bo Yuan. Energy-efficient scheduling on heterogeneous multi-core architectures. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 345–350, New York, NY, USA, 2012. ACM.

[39] Keith D. Cooper, Ken Kennedy, and Nathaniel McIntosh. Cross-loop reuse analysis and its application to cache optimizations. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '96, pages 1–19, London, UK, UK, 1997. Springer-Verlag.

[40] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 250–259, New York, NY, USA, 2008. ACM.

[41] M. Daga, Z.S. Tschirhart, and C. Freitag. Exploring parallel programming models for heterogeneous computing systems. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 98–107, Oct 2015.

[42] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[43] Frederica Darema, David A. George, V. Alan Norton, and Gregory F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[44] R.H. Dennard, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, Oct 1974.

[45] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[46] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 319–332, New York, NY, USA, 2011. ACM.

[47] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 347–358, New York, NY, USA, 2010. ACM.

[48] Michel Goraczko, Jie Liu, Dimitrios Lymberopoulos, Slobodan Matic, Bodhi Priyantha, and Feng Zhao. Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 191–196, New York, NY, USA, 2008. ACM.

[49] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 31(2):86–95, March 2011.

[50] K. Gregory and A. Miller. *C++ Amp: Accelerated Massive Parallelism With Microsoft Visual C++*. Microsoft Press, Reading, Massachusetts, 2nd edition, 2012.

[51] D. Grewe, Zheng Wang, and M.F.P. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10, Feb 2013.

[52] Dominik Grewe and Michael F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 286–305, Berlin, Heidelberg, 2011. Springer-Verlag.

[53] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. Navigating heterogeneous processors with market mechanisms. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 95–106, Washington, DC, USA, 2013. IEEE Computer Society.

[54] Apala Guha, Kim Hazelwood, and Mary Soffa. Balancing memory and performance through selective flushing of software code caches. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, pages 1–10, New York, NY, USA, 2010. ACM.

[55] Jianjun Guo, Kui Dai, and Zhiying Wang. A high performance heterogeneous architecture and its optimization design. In *Proceedings of the Second International Conference on High Performance Computing and Communications*, HPCC'06, pages 300–309, Berlin, Heidelberg, 2006. Springer-Verlag.

[56] Jawad Haj-Yihia, Yosi Ben Asher, Efraim Rotem, Ahmad Yasin, and Ran Ginosar. Compiler-directed power management for superscalars. *ACM Trans. Archit. Code Optim.*, 11(4):48:1–48:21, January 2015.

[57] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, Jul 1991.

[58] N. Ioannou, M. Kauschke, M. Gries, and M. Cintra. Phase-based application-driven hierarchical power management on the single-chip cloud computer. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 131–142, Oct 2011.

[59] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[60] Wei Jiang and Gagan Agrawal. Mate-cg: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. In *Proceed-*

ings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS '12, pages 644–655, Washington, DC, USA, 2012. IEEE.

[61] Xiaowei Jiang, A. Mishra, Li Zhao, R. Iyer, Zhen Fang, S. Srinivasan, S. Makineni, P. Brett, and C.R. Das. Access: Smart scheduling for asymmetric cache cmps. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 527–538, Feb 2011.

[62] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005.

[63] Melanie Kambadur and Martha A. Kim. An experimental survey of energy management across the stack. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 329–344, New York, NY, USA, 2014. ACM.

[64] A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, and Cho-Li Wang. Heterogeneous computing: challenges and opportunities. *Computer*, 26(6):18–27, June 1993.

[65] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[66] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, November 2005.

[67] Tapas Kumar Kundu and Kolin Paul. Improving android performance and energy efficiency. In *Proceedings of the 2011 24th International Conference on VLSI Design*, VLSID '11, pages 256–261, Washington, DC, USA, 2011. IEEE Computer Society.

[68] George Kurian, Jason E. Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C. Kimerling, and Anant Agarwal. Atac: A 1000-core cache-coherent processor with on-chip optical network. In *Proceedings of the*

19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pages 477–488, New York, NY, USA, 2010. ACM.

[69] Youngjin Kwon, Changdae Kim, Seungryoul Maeng, and Jaehyuk Huh. Virtualizing performance asymmetric multi-core systems. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 45–56, New York, NY, USA, 2011. ACM.

[70] C.G. Lee and M. Stoodley. Utdsp benchmark suite. In *http://goo.gl/PE5wjg*, 1992.

[71] Jungseob Lee, V. Sathisha, M. Schulte, K. Compton, and Nam Sung Kim. Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 111–120, Oct 2011.

[72] Rainer Leupers and Jeronimo Castrillon. Mpsoc programming using the maps compiler. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 897–902, Piscataway, NJ, USA, 2010. IEEE Press.

[73] J. Li and M. Chen. Index domain alignment: minimizing cost of cross-referencing between distributed arrays. In *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, pages 424–433, Oct 1990.

[74] J. Li and J.F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 77–87, Feb 2006.

[75] Hock-Beng Lim and Pen-Chung Yew. Efficient integration of compiler-directed cache coherence and data prefetching. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 331–340, 2000.

[76] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. Reflex: Using low-power processors in smartphones without knowing them. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 13–24, New York, NY, USA, 2012. ACM.

[77] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 285–300, New York, NY, USA, 2014. ACM.

[78] Zhenying Liu, Barbara Chapman, Yi Wen, Lei Huang, Tien-Hsiung Weng, and Oscar Hernandez. Analyses for the translation of openmp codes into spmd style with array privatization. In *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, WOMPAT'03, pages 26–41, Berlin, Heidelberg, 2003. Springer-Verlag.

[79] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[80] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Ronald Dreslinski, Jr., Thomas F. Wenisch, and Scott Mahlke. Heterogeneous microarchitectures trump voltage scaling for low-power cores. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 237–250, New York, NY, USA, 2014. ACM.

[81] Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. A run-time system for power-constrained HPC applications. In *High Performance Computing - 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*, pages 394–408, 2015.

[82] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 619–630, New York, NY, USA, 2013. ACM.

[83] Simon McIntosh-Smith, Terry Wilson, Amaurys Ávila Ibarra, Jonathan Crisp, and Richard B Sessions. Benchmarking energy efficiency, power costs and carbon emissions on heterogeneous systems. *The Computer Journal*, 55(2):192–205, 2012.

[84] Gordon E Moore. Cramming more components onto integrated circuits. electronics, 38 (8), april 1965. *VLSI Technologies and Architectures*, 2010.

[85] Bradford Nichols, Dick Buttlar, and J Proulx Farrell. Pthread programming: A posix standard for better multiprocessing. *Ed. O'Reilly*, 1996.

[86] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[87] M. F. P. O'Boyle, L. Kervella, and F. Bodin. Synchronization minimization in a spmd execution model. *J. Parallel Distrib. Comput.*, 29(2):196–210, September 1995.

[88] Santiago Pagani, Heba Khdr, Waqaas Munawar, Jian-Jia Chen, Muhammad Shafique, Minming Li, and Jörg Henkel. Tsp: Thermal safe power: Efficient power budgeting for many-core systems in dark silicon. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, pages 10:1–10:10, New York, NY, USA, 2014. ACM.

[89] Indrani Paul, Vignesh Ravi, Srilatha Manne, Manish Arora, and Sudhakar Yalamanchili. Coordinated energy management in heterogeneous processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 59:1–59:12, New York, NY, USA, 2013. ACM.

[90] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151, Sept 2008.

[91] J. Planas, R.M. Badia, E. Ayguade, and J. Labarta. Self-adaptive ompss tasks in heterogeneous environments. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 138–149, May 2013.

[92] Louis-Noel Pouchet. Polybench benchmark. In *http://www.cse.ohio-state.edu/pouchet/software/polybench/*, 1992.

[93] Alok Prakash, Siqi Wang, Alexandru Eugen Irimiea, and Tulika Mitra. Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms. In *Proceedings of the 33rd International Conference on Computer Design*, ICCD '15, 2015.

[94] Bharathwaj Raghunathan and Siddharth Garg. Job arrival rate aware scheduling for asymmetric multi-core servers in the dark silicon era. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, pages 14:1–14:9, New York, NY, USA, 2014. ACM.

[95] Nishkam Ravi, Yi Yang, Tao Bao, and Srimat Chakradhar. Semi-automatic restructuring of offloadable tasks for many-core accelerators. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 12:1–12:12, New York, NY, USA, 2013. ACM.

[96] M. Reshadi and C. Cascaval. Multidimensional dynamic behavior in mobile computing. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 113–115, Nov 2012.

[97] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 69–80, New York, NY, USA, 1989. ACM.

[98] Phil Rogers and AC Fellow. Heterogeneous system architecture overview. In *Hot Chips*, volume 25, 2013.

[99] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, March 2012.

[100] Sumit Semwal. Dma buffer sharing api guide. `http://lwn.net/Articles/489703/`, 2012.

[101] Muhammad Shafique, Siddharth Garg, Tulika Mitra, Sri Parameswaran, and Jörg Henkel. Dark silicon as a challenge for hardware/software co-design: Invited special session paper. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, pages 13:1–13:10, New York, NY, USA, 2014. ACM.

[102] Chenguang Shen, Supriyo Chakraborty, Kasturi Rangan Raghavan, Haksoo Choi, and Mani B. Srivastava. Exploiting processor heterogeneity for energy efficient context inference on mobile phones. In *Proceedings of the Workshop*

on Power-Aware Computing and Systems, HotPower '13, pages 9:1–9:5, New York, NY, USA, 2013. ACM.

[103] Weihua Sheng, Stefan Schürmans, Maximilian Odendahl, Mark Bertsch, Vitaliy Volevach, Rainer Leupers, and Gerd Ascheid. A compiler infrastructure for embedded heterogeneous mpsocs. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 1–10, New York, NY, USA, 2013. ACM.

[104] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid map task scheduling for gpu-based heterogeneous clusters. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 733–740, Washington, DC, USA, 2010. IEEE Computer Society.

[105] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 2–13, New York, NY, USA, 2003. ACM.

[106] Tyler Sondag and Hridesh Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

[107] I. Tartalja and V. Milutinovic. Classifying software-based cache coherence solutions. *Software, IEEE*, 14(3):90–101, May 1997.

[108] E. Tomusk, C. Dubach, and M. O'Boyle. Diversity: A design goal for heterogeneous processors. *Computer Architecture Letters*, PP(99):1–1, 2015.

[109] Erik Tomusk, Christophe Dubach, and Michael O'Boyle. Measuring flexibility in single-isa heterogeneous processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 495–496, New York, NY, USA, 2014. ACM.

[110] Erik Tomusk, Christophe Dubach, and Michael O'boyle. Four metrics to evaluate heterogeneous multicores. *ACM Trans. Archit. Code Optim.*, 12(4):37:1–37:25, November 2015.

[111] Chau-Wen Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 144–155, New York, NY, USA, 1995. ACM.

[112] Y. Turakhia, B. Raghunathan, S. Garg, and D. Marculescu. Hades: Architectural synthesis for heterogeneous dark silicon chip multi-processors. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–7, May 2013.

[113] C. Schlager V. Zivojnovic, J. Martinez and H. Meyr. Dspstone: A dsp-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, 1984.

[114] N. Vallina-Rodriguez and J. Crowcroft. Energy management techniques in modern mobile handsets. *Communications Surveys Tutorials, IEEE*, 15(1):179–198, First 2013.

[115] Jos van Eijndhoven. Power measurement in omap4. `http://goo.gl/TH2Y5R`, 2011.

[116] Ankush Varma, Brinda Ganesh, Mainak Sen, Suchismita Roy Choudhury, Lakshmi Srinivasan, and Bruce Jacob. A control-theoretic approach to dynamic voltage scheduling. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 255–266, New York, NY, USA, 2003. ACM.

[117] Alexander V. Veidenbaum. A compiler-assisted cache coherence solution for multiprcessors. In *International Conference on Parallel Processing, ICPP'86, University Park, PA, USA, August 1986.*, pages 1029–1036, 1986.

[118] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 121–132, Piscataway, NJ, USA, 2014. IEEE Press.

[119] Magudilu Vijayaraj and Thejasvi Magudilu. An empirical power model of a low power mobile platform. 2013.

[120] Guibin Wang and Xiaoguang Ren. Power-efficient work distribution method for cpu-gpu heterogeneous system. In *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pages 122–129, Sept 2010.

[121] Hao Wang, Vijay Sathish, Ripudaman Singh, Michael J. Schulte, and Nam Sung Kim. Workload and power budget partitioning for single-chip heterogeneous processors. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 401–410, New York, NY, USA, 2012. ACM.

[122] Matthew A. Watkins and David H. Albonesi. Remap: A reconfigurable heterogeneous multicore architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 497–508, Washington, DC, USA, 2010. IEEE Computer Society.

[123] V.M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. Measuring energy and power with papi. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 262–268, Sept 2012.

[124] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (mpsoc) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, Oct 2008.

[125] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, and Hong Wang. Pangaea: A tightly-coupled ia32 heterogeneous chip multiprocessor. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 52–61, New York, NY, USA, 2008. ACM.

[126] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 248–259, New York, NY, USA, 2004. ACM.

[127] Qiang Wu, Margaret Martonosi, Douglas W. Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. Dynamic-compiler-driven control for microprocessor energy and performance. *IEEE Micro*, 26(1):119–129, January 2006.

[128] Youfeng Wu, Shiliang Hu, Edson Borin, and Cheng Wang. A hw/sw co-designed heterogeneous multi-core virtual machine for energy-efficient general purpose computing. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 236–245, Washington, DC, USA, 2011. IEEE Computer Society.

[129] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[130] Tomofumi Yuki and Sanjay Rajopadhye. Folklore confirmed: Compiling for speed = compiling for energy. In Calin Cascaval and Pablo Montesinos, editors, *Languages and Compilers for Parallel Computing*, volume 8664 of *Lecture Notes in Computer Science*, pages 169–184. Springer International Publishing, 2014.