THE UNIVERSITY *of* EDINBURGH

Compiler-Driven Data Layout Transformations for Network Applications

Damon Fenacci

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2012

Abstract

This work approaches the little studied topic of compiler optimisations directed to network applications.

It starts by investigating if there exist any fundamental differences between application domains that justify the development and tuning of *domain-specific* compiler optimisations. It shows an automated approach that is capable of identifying domain-specific workload characterisations and presenting them in a readily interpretable format based on decision trees. The generated workload profiles summarise key resource utilisation issues and enable compiler engineers to address the highlighted bottlenecks.

By applying this methodology to data intensive network infrastructure application it shows that *data* organisation is the key obstacle to overcome in order to achieve high performance.

It therefore proposes and evaluates three specialised *data transformations* (*structure splitting*, *array regrouping*, and *software caching*) against the industrial EEMBC networking benchmarks and *real-world* data sets. It also demonstrates on one hand that speedups of up to 2.62 can be achieved, but on the other that no single solution performs equally well across different network traffic scenarios.

Hence, to address this issue, an *adaptive* software caching scheme for high frequency route lookup operations is introduced and its effectiveness evaluated one more time against EEMBC networking benchmarks and real-world data sets achieving speedups of up to 3.30 and 2.27. The results clearly demonstrate that adaptive data organisation schemes are necessary to ensure optimal performance under varying network loads.

Finally this research addresses another issue introduced by data transformations such as array regrouping and software caching, i.e. the need for *static analysis* to allow efficient resource allocation. This thesis proposes a static code analyser that allows the automatic resource analysis of source code containing lists and tree structures. The tool applies a combination of amortised analysis and separation logic methodology to real code and is able to evaluate type and resource usage of existing data structures, which can be used to compute global resource consumption values for full data intensive network applications.

Acknowledgements

First of all I would like to thank my supervisor, Björn Franke for his precious, wise and very enlightening advice during my PhD. I also want to thank him for always encouraging me when I felt stuck and my research wasn't producing the results we were hoping. I'd also like to thank Mino and Kenneth for the very constructive collaboration I had with them and the incredible amount of knowledge I gained from them. Furthermore I'm very thankful to my colleagues and friends of the CArD and other Informatics groups, in particular Matteo for the endless technical and non-technical discussions we had as well as George, Chronis, Sofia, Nikolas, Marcela, Alberto, Zheng, Christophe and Salman. A lot of gratitude also goes to Mahesh and Mike for regularly revising my research and giving me useful hints, to David for supervising our work for the RESA project and to Vijay and Wim for accepting to be my examiners. Finally I would like to thank Dwynwen for putting up with me during the past three years and for proof reading this thesis, my family and Erin in particular for being able to see me only once in a while and all my friends in Edinburgh for making my years there the best ones of my life.

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.

(*Damon Fenacci*)

Related Publications

The contents of this thesis are partly based on the following published, peer-reviewed papers:

- Damon Fenacci, Björn Franke and John Thompson, Automatic Identification of Tuning Opportunities for Domain-Specific Compilers Using Decision Trees for Data Mining, *Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, June 2010
- Damon Fenacci and Björn Franke, Empirical evaluation of data transformations for network infrastructure applications, *International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS X)*, July 2010
- Giacomo Bernardi, Matt Calder, Damon Fenacci, Alex Macmillan and Mahesh Marina, *Stix: A Goal-Oriented Distributed Management System for Large-Scale Broadband Wireless Access Networks, International Conference on Mobile Computing and Networking (MobiCom)*, September 2010
- Kenneth McKenzie and Damon Fenacci, Static Resource Analysis for Java Bytecode Using Amortisation and Separation Logic, *6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode)*, March 2011
- Giacomo Bernardi, Damon Fenacci, Mahesh Marina and Dimitrios Pezaros, Large-Scale Broadband Quality Assessment Using Distributed Monitoring, IFIP/TC6 Networking 2012, May 2012.
- Damon Fenacci and Björn Franke, Adaptive Software Caching for Network Packet Processing Applications, under review at the Journal of Information Technology & Software Engineering, OMICS Publishing Group

Table of Contents

Chapter 1

Introduction

Network appliances are ubiquitous in our everyday life. Contrary to the situation of just a decade ago, network applications are not only running on high-end hardware such as Internet core routers but we find them also more and more in a multitude of small devices deployed in our houses, cars and offices. Some of them are even worn by us, e.g mobile phones, mp3 players etc. All of them are connected to some network for a multitude of reasons, e.g. to provide Internet connectivity, download music or video, share photos, allow phone calls etc. Their connectivity is limited to the personal level or, for home devices, to a small group of people that a family or a small office can represent. From their origins on they have been conceived to handle network traffic respecting the constraints of such a small group. This fact and the market pressure to keep costs low have pushed manufacturing companies to rely on off-the-shelf platforms and processors for their hardware and generic open source software for their applications and device drivers, including the one that handles network traffic.

Internet traffic has been growing at exponential rates since the mid-nineties. Since the year 2000 this growth has been between 50% and 70% in the U.S. alone and it is estimated that by the year 2015 Internet traffic will reach a total of 10^{21} bytes, i.e. 50 times more than it was in 2006 as shown in figure 1.1. Furthermore the type of traffic has been constantly changing and if web and P2P represented the majority of traffic in the past we can foresee that in the future applications such as film downloading, IPTV, video conferencing, cloud computing and virtualisation will make up the majority of traffic flowing through networks [Swanson and Gilder, 2008].

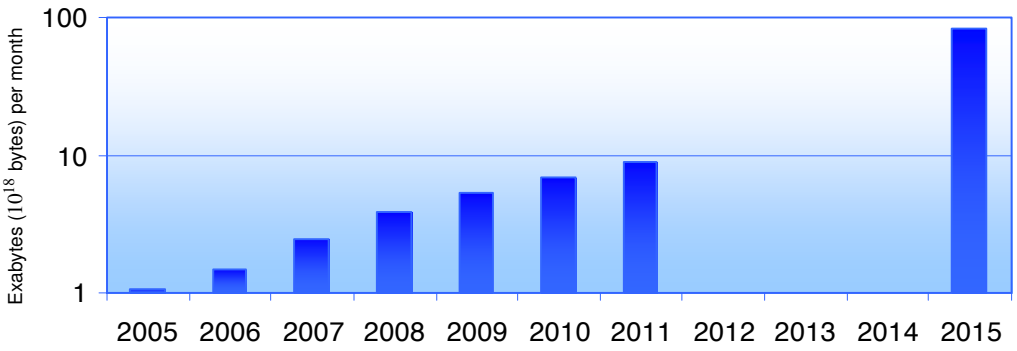


Figure 1.1: Evolution of Internet traffic from 2005 and 2015 projection (source [Swanson and Gilder, 2008]).

This Internet traffic trend has led in the past to the creation of highly specialised hardware architectures for high-end network devices known as network processors. These platforms consist of heterogeneous hardware that handle all the high-performance tasks that a network application requires. In the last decade companies such as NETRONOME, EZCHIP, XILINX and LSI introduced their own network processor (NP) architecture along with proprietary software and development environment. Although their architectures differ, they all have a number of common features. First of all they all share a highly parallel architecture. Components responsible for processing network packets usually consist of very simple RISC processors called *microengines*. These are usually employed in a parallel, pipelined or more often in a mixed way to process a high number of network packets. Next to the microengines, NPs often include a multitude of specialised hardware components and coprocessors that handle performance-intensive tasks such as error correction, deep packet inspection and traffic shaping, and a general purpose processor to perform non-time-critical tasks such as routing table buildup and shortest path algorithm computation [Shah, 2001]. Figure 1.2 shows such an architecture for the case of the NETRONOME NFP-32XX architecture.

If NPs have been constantly kept up to date with the Internet traffic increase by regularly doubling the number of their cores and adapting their special hardware, until recently no such performances were needed for top-home or mid-range devices. These devices don't include any specific hardware structure or component to process the fast paced network operations needed for the new type of network applications such as IPTV, streaming, etc. Instead they rely on off-the-shelf hardware to process increas-

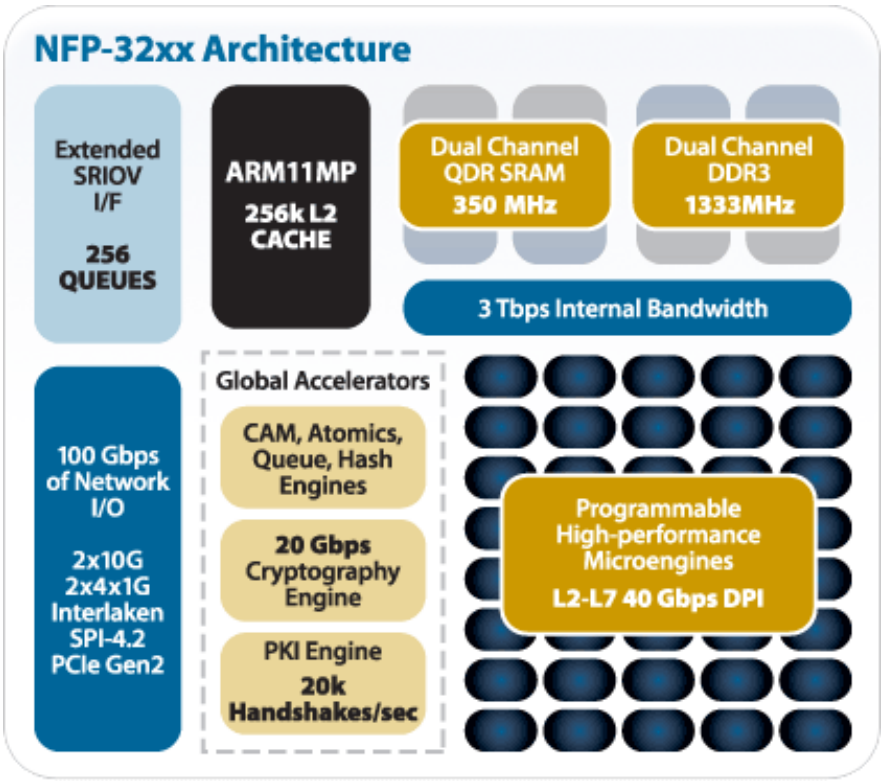


Figure 1.2: Architecture of the NETRONOME NFP-32XX network processor.

ingly big amounts of data with even more complicated network algorithms. In many cases network applications are therefore starting to become the bottleneck for these systems, which have difficulties coping with the amount of networking-related work they have to perform. To address these issues at a software level new code and data transformations have to be applied either to source code or compiler optimisations.

This represents the core of this research. This thesis investigates network applications and how their implementation and compilation can be manipulated to make them run more efficiently regardless of what particular architecture they are running on. It first shows how application domains differ in their workload by demonstrating how data mining techniques such as *decision trees* and *clustering* can be used in an innovative way to determine what the differences among domains are and what issues a software and/or compiler developer has to address to target each domain. Using the results of this analysis we discovered that the major bottleneck for the network application domain lies in the suboptimal usage of the data cache. This knowledge leads us to present three different data transformations (*structure splitting*, *array regrouping* and

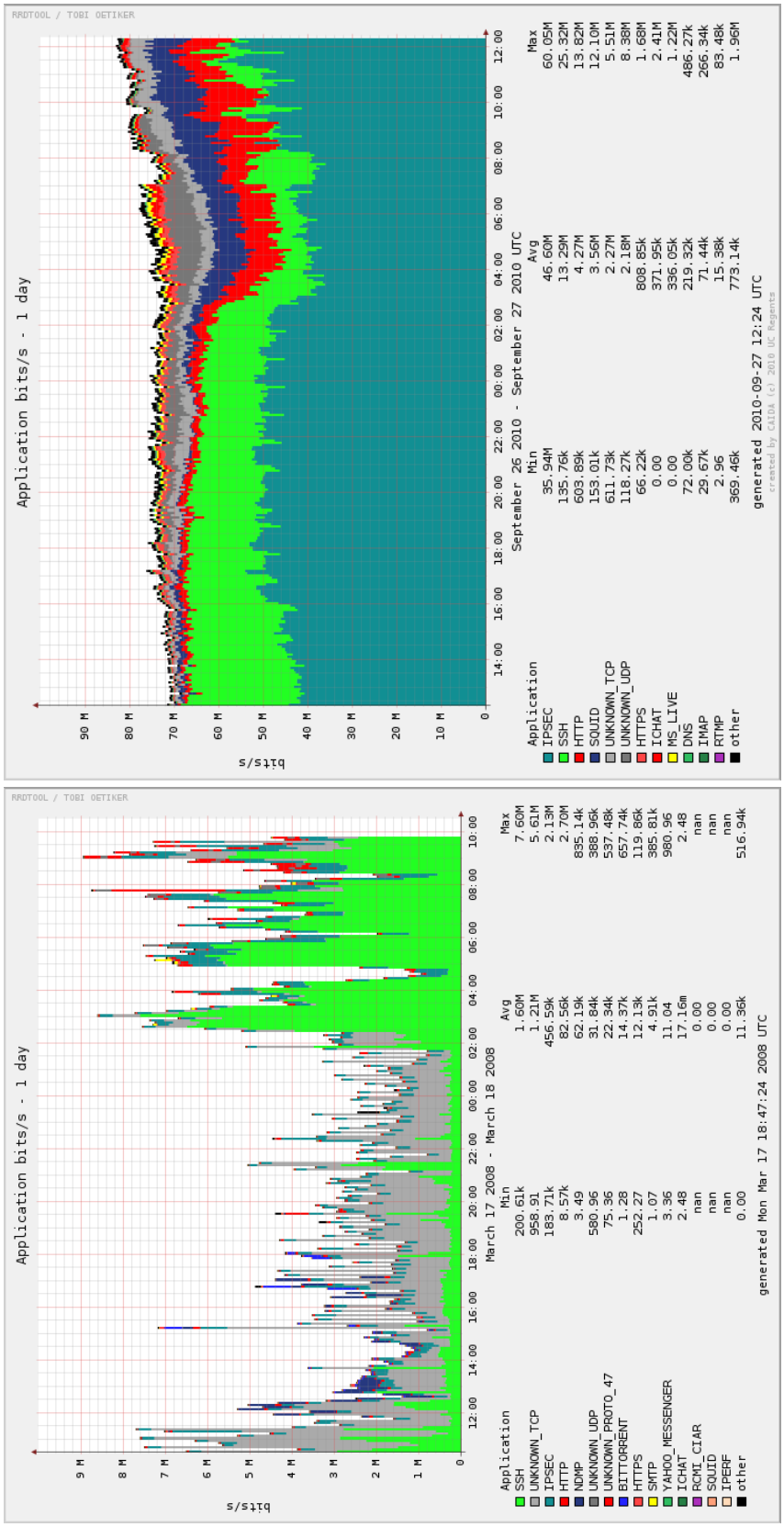


Figure 1.3: Variation in traffic volume during 1 day and type in 2 different CAIDA traffic exchange facilities: SD-NAP and AMPATH [CAIDA, 2011].

software caching) applicable at source or compiler level that tackle bad data cache usage.

These optimisations are specific to the network domain in that they not only improve their performance but they are also only applicable to this set of applications.

To widen our understanding of the data transformation behaviour we used a number of different traffic and topology data sources coming from different sized networks. Traffic can vary substantially from one network to the other and depending on time as figure 1.3 shows for two different routers belonging to The Cooperative Association for Internet Data Analysis (CAIDA). The graphs show the variation of traffic amount (represented by the height of the plotted area) and type (represented by the different colours in the plotted area) at these two routers during 24 hours. They demonstrate that the traffic that flows through them is very different in amount and type and that, even within the same router, there are significant traffic changes during the day. Using this and a few other traffic sources for out tests gave us a thorough view of the behaviour of the data transformations under different network conditions and showed us that, in order to achieve the best possible results, we need to adapt the newly devised optimisations to the particular network environment, to which they are subject.

It also proved to us that utilising profiling approaches to adapt compiler optimisations as done in generic domain-agnostic compilation doesn't achieve the desired continuous performance increase. In fact, in profiling approaches the adaptation takes place only using data gathered at profile time, but this isn't necessarily representative of the behaviour of network applications at all time since traffic conditions can vary dramatically.

Therefore in this research we introduce an online adaptation technique that we apply to software caching. The adaptation is performed constantly depending on the amount of changes in the behaviour of the network application due to varying traffic patterns. This allows the software cache to be exploited in the best way possible according to the current network traffic.

At the same time this research takes a look at static analysis techniques to gather information about network applications without these being executed and profiled or dynamically adapted. Static analysis plays an important role in making the network

specific data transformations we introduced effective allowing the discovery of upper bounds for the size of data structure and revealing how many resources in terms of memory usage and execution time are needed.

1.1 Motivation

Although network applications and network processors have been the focus of numerous publications especially at the beginning of last decade, relatively little has been explored to target generic code and data transformations for networking software. Of the literature that touches upon it, most of it relies either on architectures specifically designed for particular types of network applications [Wagner and Leupers, 2002, Li et al., 2002, Gopalan and Chiueh, 2002, Kind et al., 2002, Zhuang, 2006, Zhuang and Pande, 2004, Liu et al., 2008, Mudigonda et al., 2008], i.e. making use of specific network processor components, or proposing ad-hoc hardware extensions to speedup parts of them [Wang and Kaeli, 2003, Huggahalli et al., 2005, Cascón et al., 2011, Abid et al., 2007]. Very little studies approach the issue of compiler optimisations for generic platforms (e.g. [Degermark et al., 1997]) and it mostly only do so from a theoretical point of view.

Generic, i.e. not domain-specific compiler optimisations have been the focus of research for decades (e.g. in [Muchnick, 1997, Alfred et al., 1986]) and, more recently, profile driven optimisations have been used successfully in many areas [Hagog and Tice, 2005, Hubicka, 2005, Gupta et al., 2002, Chen et al., 1993]. GCC, for instance, has a whole set of such profile-guided optimisations. To take advantage of them, the compilation has to occur in two steps: a first one where profiling information is gathered and a second one where this information is used to adjust optimisations. Although this technique has proven effective for many domains, we discovered that it is not particularly suited for network applications as the vast majority of networks show great variability in the data handled depending on location and time. This means that it is not possible to find representative data to drive profiling beforehand since this largely depends on the network and the time the profiling step has been performed.

1.2 Goals

For the work in this thesis we have set ourselves the following goals:

1. analysis of application workloads to bring to light the bottlenecks specific to each application domain
2. design of novel compiler optimisations to target network applications that take into account the issues discovered in point 1
3. adaptation of the compiler optimisations found in point 2 to changing network conditions to be able to always get the maximum possible performance improvement in all environments
4. exploration of static code analysis tools to allow the effective implementation of the compiler optimisations introduced in point 2

1.3 Contributions

Among the contributions of this research are:

1. the investigation of novel techniques using decision trees for characterising embedded applications from different application domains,
2. the comparison with a clustering based workload characterisation methodology,
3. the demonstration that the obtained workload characterisation can be successfully used to pinpoint performance bottlenecks and to eventually identify domain-specific compiler transformations,
4. conclusive evidence that poor data cache utilisation limits the performance of networking infrastructure applications,
5. the introduction of three data transformations particularly suitable for networking applications (*structure splitting*, *array regrouping* and *software caching*)
6. an extensive evaluation of the effectiveness of the proposed data transformations against the industry standard EEMBC benchmarks, two embedded RISC plat-

forms (INTEL STRONGARM and ARC 750D) and *real-world* network traffic data,

7. the evaluation of two different software caching sampling policies (*periodic sampling* and *continuous sampling*) that aim to minimise the overheads for providing adaptivity to software caching,

8. the empirical evaluation of the adaptive software caching policies against the EEMBC networking benchmarks and the aforementioned embedded RISC architectures and traffic data.

9. the demonstration by means of the RESA static analysis tool that static code analysers can help implement data transformations for network applications effectively.

1.4 Overview

The remainder of this thesis is organised in two different parts:

1. Part I considers network applications that handle data with very tight timing constraints. Applications belonging to this category include the ones performing (IP) address lookup, (IP) packet header error checking, packet store and forward, NAT, deep packet inspection etc. In chapter 4 we look into a new way of characterising applications to point out bottlenecks specific to a particular application domain. We then focus on data intensive applications still giving a short overview on the remaining, control plane, network applications. Based on the results obtained, in chapter 5 we devise three different compiler optimisations directed towards network applications and evaluate them against real network traffic and topology data. This points us to the necessity of making the data transformations adaptive to the changing network environment which we analyse in chapter 6.

2. This is followed by part II where we investigate static analysis approaches to discover the resource utilisation of data structures that are used in network applications (e.g. linked lists, trees, etc.) and essential for the implementation of the

data transformations proposed above. We then show a tool that is able to gather resource information about their sizes and access times.

Chapter 2

Related Work

2.1 Introduction

In this chapter we discuss related work for the subjects of workload characterisation and machine learning techniques, compilation strategies for network related applications and adaptive approaches to compilation. As we will show, for each of these aspects previous research failed to provide the answers we are looking for. For instance in the past workload characterisation has never covered methodologies to discover domain specific performance bottlenecks. Besides, architecture-independent compiler optimisation techniques for network applications and strategies to adapt network software to network traffic changes have been studied very rarely.

2.2 Data Plane

Workload Characterisation

There exists a large body of publications on workload characterisation [Downey and Feitelson, 1999, Ding and Newman, 2000, Yan, 2004], much of which focuses on desktop [Lin et al., 2008], high-performance [Skinner and Kramer, 2005] or database workloads [Yu et al., 1992]. In this research we are more concerned about embedded applications originating from different domains of embedded systems. Most relevant is the

existing workload characterisation [Poovey, 2007] provided by the Embedded Microprocessor Benchmark Consortium (EEMBC). This characterisation has been carried out to enable designers to select the most relevant benchmarks from the EEMBC suite in terms of program similarity and to infer expected performance figures for their own applications or platforms. We, in contrast, are interested in how embedded application domains differ from each other and what features can be exploited for prospective domain-specific compiler development. In [Hoste and Eeckhout, 2008] a characterisation methodology for general-purpose and domain-specific benchmark suites is presented. We build on this work and follow a similar clustering approach, however, there exist a number of important differences. First, in our characterisation we explicitly include platform-specific features (as opposed to micro-architecture-independent characteristics) coming from a number of different embedded architectures as we are interested in specific compiler/architecture interactions. Secondly, [Hoste and Eeckhout, 2008] focuses on phase-level characterisation of complex applications with a distinct phase behaviour, whereas we have chosen smaller, but domain-specific and compute-intensive benchmarks representative of deeply embedded systems running a single, fixed application. Finally, we develop a human interpretable characterisation supporting the compiler or software engineer whereas [Hoste and Eeckhout, 2008] seeks to identify the unique and diverse behaviours of different benchmark suites. Similarly, [Hoste and Eeckhout, 2007] and [Joshi et al., 2006] aim to derive and measure micro-architecture-independent program features for clustering. Eventually, each cluster is expressed by a single ”most representative” application. Both papers use principal component analysis (PCA) for dimension reduction. This technique is useful to extract the vector components in order of variance magnitude by expressing them as orthogonal vectors that are linear transformations of the original ones. For this reason PCA can be used to effectively reduce the number of dimensions to the ones that are most relevant by discarding the vectors with lower variance but makes it harder to trace and understand the causes for a certain behaviour in the domain space of the original features. [Hoste and Eeckhout, 2007] also uses Kiviat diagrams for the representation of the results as does [Poovey, 2007]. Although recognising the merits of this technique it has the major weakness of relying on human interpretation for the extraction of evidence and we found it not to be suitable if we aim at an objective methodology with proven mathematical foundations. [Joshi et al., 2006] introduces K-means

for clustering and in [Hoste et al., 2006] program similarity is exploited to facilitate performance prediction. Clustering is not what we aim for since it doesn't take into account predefined domains. Nevertheless we use this technique for part of our workload characterisation in combination with other methodologies, e.g. decision trees. SVM techniques have been used in various publications to discriminate between applications for different purposes (e.g. multicore mapping [Wang and O'Boyle, 2008]). Unfortunately, like clustering, we don't see them as ideal for our purpose since, although they take into account given categories, the classification takes place using a kernel function which, in the simplest of cases is a linear combination of features and, as for PCA makes it hard to trace the causes of a certain behaviour. Finally also Bayes techniques have been used for e.g. workload classification [Joshi et al., 2006, Li et al., 2008, Hoste and Eeckhout, 2006], power management [Jung and Pedram, 2010] and traffic classification [Erman et al., 2006, Bessa Maia and Holanda Filho, 2009]. They have the advantage of being simple and can perform surprisingly well even when features are presumed independent. Nevertheless, they usually do not perform as well as decision trees [Caruana and Niculescu-Mizil, 2006] and, more importantly, we believe that they are not as immediate as decision trees in presenting the classification to the compiler developer. Additionally for Bayesian techniques to be effective, dimension reductions techniques like PCA have to be used, which further hinders the understandability of the results. Finally code coverage and input variability for a number of embedded applications is studied in [Hunter and Hwu, 2002]. This work aims to support compiler and architecture research by identifying weaknesses such as superfluous functions and insufficient code coverage of existing benchmarks.

Compiler Optimisation Techniques

Classification using decision trees has been employed for the exploration of cache hierarchies for commercial application workloads [Elakkumanan et al., 2005], generation of compiler optimisation heuristics [Monsifrot et al., 2002], branch prediction [Calder et al., 1997], dynamic power management [Chung et al., 1999], and packet classification in network processors [Liu et al., 2006]. In all these works decision trees have been employed as classifiers, i.e. a tree has been built in order to systematically assign every software application to a predefined group with similar characteristics. As we

will extensively explain in chapter 4, we propose instead to build decision trees and look at them as a whole to figure out features that differentiate application domains. K-means clustering is discussed in [Phansalkar et al., 2005] and [Joshi et al., 2006] for benchmark categorisation. Similarly to these articles, we make use of this clustering technique to group benchmarks together but our only purpose is to enhance our decision trees data mining strategy to take similarities into account.

Adaptive compilation [Cooper et al., 2005] largely focuses on machine learning for improved phase ordering in an optimising compiler [Agakov et al., 2006] or tuning of optimisation heuristics [Monsifrot et al., 2002] or compiler settings [Cavazos et al., 2007] [Hoste and Eekhout, 2008]. While this improves the performance of existing compilers it does not help identify the code generation and optimisation deficits that demand genuinely new transformations rather than the tweaking of existing ones. In contrast, in this research we apply data mining techniques to extract human-interpretable information, supporting the compiler engineer in developing new, *domain-specific* compiler optimisations.

Compilation for network processing units (NPUs)

Although this research focuses on compilation for generic embedded platform we try to take advantage of findings originating from specific network architectures.

Compilation for NPUs has been an active research area for more than a decade [Ramakrishna and Jamadagni, 2003, Ding and Liu, 2005, Dai et al., 2005, Zhuang, 2006, Kim et al., 2002, Wagner and Leupers, 2002, Wagner and Leupers, 2001] and and a comprehensive summary of the programming challenges in network processor deployment are documented in [Kulkarni et al., 2003].

The design and implementation of C compilers for NPUs is subject of [Wagner and Leupers, 2001] and [Kim et al., 2002]. In [Wagner and Leupers, 2002] a code generation technique targeting NPU-specific bit-packing instructions is presented.

Memory Utilisation Improvement Techniques

Cache design for NPUs has been repeatedly discussed in the computer architecture community, e.g. in [Gopalan and Chiueh, 2002, Liu et al., 2008, Xu et al., 2009, Chiueh and Pradhan, 2000]. With this work we share the view that data access in an NPU form a performance bottleneck, however, we propose a more flexible software solution orthogonal to existing hardware approaches. We don't restrict ourselves to any specific NPU but instead try to solve the poor data access performance in a broader way by using generic embedded architecture that ultimately also form the core of NPUs.

Reference affinity in data transformations has been investigated in [Zhong et al., 2004, Hundt et al., 2006, Hagog and Tice, 2005]. These works are primarily concerned with general-purpose workloads rather than networking applications. We clearly agree that the transformations proposed here can be applied to network processors but we aim to find other ones that are specifically targeting network applications and, because of this, show a greater impact.

For the specific case of route lookup applications (which will be the focus of chapter 6) [Nie et al., 2005] and [Degermark et al., 1997] propose partially new ways of organising prefixes to perform faster route lookups in many situations. Their approach focuses on replacing the routing trie with a completely new structure not taking into consideration the possibility of having both being used simultaneously or dynamically in any way. In chapter 6 we aim instead to have the old and new data structure working together and the latter being modified when needed to adapt to changing traffic. Additionally their results are mainly based on theoretical reasoning and do not take into account real architectures of existing network processors, which is the approach we take.

2.3 Static Analysis

In order to be able to establish the resource constraints needed to run network applications and apply optimisations as effectively as possible we looked into static analysis approaches. Our technique uses the theoretical work on combining separation logic

with amortised analysis first proposed in [Atkey, 2010] and demonstrates its applicability to actual applications written in a high-level programming language.

A number of other resource-inference techniques have been proposed and implemented, for Java and also for other languages but none of them have actually used amortised analysis and separation logic to extract resource usage constraints effectively. For instance the COSTA system of Albert et al. [Albert et al., 2007, Albert et al., 2008] converts JVM bytecode into a collection of "cost equations" which are then solved to obtain a symbolic (and possibly non-linear) bound for resource usage.

The SPEED project of Gulwani et al. [Gulwani et al., 2009, Gulwani and Zuleger, 2010] uses a number of techniques to obtain (non-linear and symbolic) bounds for the number of times a program location is visited, and has been applied to the complexity analysis of C# programs.

The Hume project [University of St Andrew/Heriot-Watt University,] instead defines an ad-hoc functional programming language aimed at extracting time and space execution costs for applications with running on devices with limited resources such as embedded systems.

Type-theoretic methods for resource usage inference are described in [Portillo et al., 2003] (inference of symbolic bounds for recursive functional programs) and [Chin et al., 2005] (heap usage for object-oriented programs).

Our technique makes use of annotations to pass extra necessary information to the analyser. A number of other representations have appeared in the literature to solve the same problem. Many of these ([League et al., 2001], [Beringer et al., 2003], and [Albert et al., 2008] for example) utilise the technique of introducing new variables to represent values on the Java VM operand stack. The Soot framework (see [Vallée-Rai et al., 1999] for example) contains a number of intermediate representations for Java bytecode and has been applied to many problems in optimisation and analysis. Another framework for bytecode analysis is Julia, which is described in [Spoto, 2005]. Finally, another OCaml representation for Java class files is described in [Hubert et al., 2010]; this has much in common with our representation.

Chapter 3

Background Infrastructure

3.1 Introduction

In this chapter we give a brief overview of the hardware and software infrastructure choices we have made to run experiments throughout our research. These include benchmark programs, hardware platforms, compilers, data sources and various tools.

3.2 Benchmarks

The evaluation of the data plane workload characterisation, data transformations and adaptation has been performed using either the whole set or part of the EEMBC benchmark suite [The Embedded Microprocessor Benchmark Consortium, 2008]. This set of benchmarks include 43 programs divided into 5 domains according to their use as summarised in table 3.1.

Benchmark Suite	Description
Automotive/Industrial	Tests the performance of processors in automotive, industrial, and general-purpose applications
Consumer	Approximates the processor's performance of such end products as PDAs, mobile phones, MP3 players, digital cameras, TV set-top boxes, and in-car entertainment systems
Networking	Approximates the performance of processors tasked with moving packets in networking applications
Office Automation	Approximates the tasks performed by processors in printers, plotters, and other office automation systems that handle text and image processing tasks
Telecommunications	Approximates the performance of processors in modem, xDSL and related fixed-telecom applications

Table 3.1: EEMBC benchmark suite

3.3 Target Platforms

Data Plane

Throughout this thesis the evaluation of data plane applications has been performed against three embedded RISC processor cores: INTEL STRONGARM, ARC 750D and MPC7410. Details of the processors and their memory system, their simulators and the software development tool chains are shown in table 3.2. These platforms are representative of many proprietary RISC cores of which more complex NPUs are composed. The INTEL STRONGARM is utilised in the first generation of INTEL's IXP network processor line and the ARC 750D can be found in many home networking devices.

To be able to perform quick prototyping of the cache behaviour of these platforms without having to run benchmark programs on simulators, a combination of the PIN code instrumentation tool [Luk et al., 2005], to trace memory accesses of networking programs, and the DINERO IV cache simulator [Edler and Hill, 1999], to evaluate cache misses, have been used.

Platform	STRONGARM	ARC 750D	MPC7410
Processor Core			
• Pipeline	5-stage	7-stage (interlocked)	Sep. int., FP & vector pipelines
• Execution Order	In-Order	In-Order	Out-Of-Order 8 FUs, dual issue
• FP Support	No (SW)	Yes	Yes
• Branch Pred.	No	Yes	Yes
Memory System			
• Level 1 Cache	16k(I)/8k(D)	8k(I)/8k(D)	32k(I)/32k(D) (8)
• Level 2 Cache	-/-	-/-	1024k
• MMU	Yes	Yes	Yes
• Main Memory	Simple RAM	Simple RAM	SDRAM/MPC106
• Bus	native	native	60x bus
Simulation			
Simulator	SimIt-ARM	ARC simulator	Freescale SimG4
• Options	FP library	Default	Default
• I/O	Emulated	Emulated	Native
SW Dev. Tools			
• Compiler	GCC 3.3.2	GCC 4.2.1	GCC 4.0.1
• Compiler Opt.	-O3	-O3	-O3

Table 3.2: Details of the INTEL STRONGARM SA-1110, ARC 750D and FREESCALE MPC7410 platforms.

Static Analyser

The static analyser presented in part II has been developed in OCaml, an ML-derived language with an object-oriented extension. The static analyser can take any Java bytecode as input and produces resource utilisation constraints for all of its methods. The choice of a functional language is due on one side to the easy bytecode parsing support that these types of languages offer and on the other to the availability of libraries implemented for previous projects.

3.4 Network Data Sources

To empirically evaluate data plane code transformations we rely on the EEMBC benchmark set, which is widely recognised as one of the most authoritative. Several bench-

Source	Time	Description
CAIDA	1 day/month in 2009	1 min traffic traces from the *equinix-chicago* router at different times in the morning
	31/03/2009 – 12:28	IP Prefix to autonomous system map (routing table of *equinix-chicago* backbone router)
	13/09/2009	Topology of the IPv4 autonomous systems [Hyun et al., 2009]
UoE	12/01/2010 every hour	1 min traffic traces from the edge router of the Informatics School
	18/09/2009	Routing tables of the edge router of the Informatics School
	16/02/2010	Topology of the network of the Informatics School

Table 3.3: New sources of input data for EEMBC networking applications.

mark programs require input data to be fed to them and EEMBC provides one or more of them and, for some benchmarks, it envisages different scores depending on the data source. Unfortunately the source of data used by networking benchmark applications proved inappropriate in several cases (see chapters 5 and 6). New data sets have therefore been introduced to complement them, whose sources are on one side the The Cooperative Association for Internet Data Analysis and on the other the IT support team of the Informatics School of the University of Edinburgh. The first institution provided traffic, routing and topology data as seen by their *equinix-chicago* backbone router [Walsworth et al., 2009]. The second group supplied similar type of data extracted from an edge router of the Informatics School network of the UoE (details in table 3.3).

Different parts of the data have been used to assess the performance of data transformations and adaptation strategies. More details are provided in specific chapters.

3.5 Data Mining Tools

The workload characterisation of data plane network applications in this research has been prototyped using the RapidMiner open source data mining tool [Mierswa et al., 2006]. RapidMiner is a machine learning environment that allows the quick evaluation of data mining processes by building them visually connecting data mining algorithm

basic blocks together and visualising results in a spectrum of different ways. This tool has been used to devise decision trees combined with feature selection as well as to apply the K-Nearest Neighbour clustering algorithm and cross validation on workload features.

Part I

Data Plane

Introduction and Overview

Data plane applications are the subset of network applications that include programs with very tight timing constraints. Applications that are a common part of router software such as IP address lookup, IP packet header error checking, IP packet store and forward, NAT address translation and various deep packet inspection protocols are just a few examples of them. These applications usually require powerful hardware in order to be able to process network traffic. For instance modern network processing units are able to process packets at rates above 100-200Gbps that, with IP packets of average length, means handling up to 400 million packets per second. In other words, applications such as IP address lookup have to be able to perform one lookup in less than 2.39 nanoseconds. Handling this massive amount of data puts a strain on the underlying hardware and software but, if during the last decade plenty of research has been carried out for the hardware side and new and more powerful architectures for high-end network processing have constantly been developed, this has surprisingly not happened for the software side, as pointed out in chapter 2. In particular, very little attention has been given to compiler optimisations and code and data transformations for network software.

To maximise the impact of any code optimisation attempt for network applications and other software domains it is especially important to detect their performance bottlenecks since not all parts of the code under scrutiny show the same timing and resource usage behaviour. In order to understand and point out the bottlenecks specific to data plane networking applications, first in chapter 4 a study of their workload is conducted on a set of applications coming from a wide variety of domains. For this purpose EEMBC benchmark applications are used since they are widely recognised as being representative of software running on today's embedded systems. Characteristics

that distinguish networking applications are extracted using specific data mining techniques, i.e. *decision trees* and *clustering*. These point to *data layout* as being the major issue for data plane application performance bottleneck.

To tackle this problem in chapter 5 three different data manipulation techniques are therefore proposed: *structure splitting*, *array regrouping* and *software caching*. These transformations prove to be extremely effective in most cases but the latter two can also fail to deliver the expected performance improvements when they are exposed to specific network environment and traffic.

Hence a strategy to allow them to adapt to the current condition is proposed in chapter 6 and assessed for *software caching*. This procedure envisages adaptation not only for data organisation but also for the internal parameters of the caching solution, which are also modified at runtime to answer current traffic patterns.

Chapter 4

Workload Characterisation

Embedded processors are used in a vast range of different application domains. This is reflected, for example, in the provision of domain-specific benchmarks offered by EEMBC, covering embedded application domains such as automotive, consumer, digital entertainment, Java, networking, office automation and telecoms. For some of these domains, hardware vendors have developed specialised embedded processors such as digital signal processors, multimedia processors and network processors that contribute to more efficient solutions at a lower price due to their domain specialisation. However, the same is not true for the compiler technology targeting embedded applications. In fact, compiler research in this area is driven by the development of new hardware and has largely focused on targeting specific architectural features such as dual memory banks [Sipkova, 2003], zero-overhead loop buffers [Uh et al., 1999] and irregular data paths [Lee and Chen, 2007]. More recently, adaptive techniques [Cooper and Waterman, 2003] have found their way into optimising compilers, but their application is restricted to the tuning of heuristics [Monsifrot et al., 2002, Stephenson et al., 2003] and selecting the order of code transformation sequences [Agakov et al., 2006]. So far, little advantage has been taken of domain-specific workload characteristics to further specialise existing compilers to a particular embedded domain. We believe that this is due to two reasons: (a) the lack of work in characterising different embedded application domains and (b) the lack of knowledge transfer between the workload characterisation and compiler development communities.

In this chapter we investigate a data mining approach using *human-interpretable* de-

cision trees to explore the characteristics of a broad range of embedded applications. We seek to identify and characterise the specific traits of different embedded domains in such a way that they can be exploited for the development of domain-specific optimising compilers, targeting data intensive network applications in particular. We also aim at differentiating these to control plane network applications, which don't show the same data hungry behaviour.

4.1 Motivation

An earlier workload characterisation of embedded applications [Poovey, 2007] from different domains has looked into the resource utilisation of processor components such as the arithmetic-logic unit (ALU), load/store unit (LSU), multiplier and divisors (MULTDIV), shifters (SHIFT) and the branch unit (BRANCH). The author uses an array of Kiviat diagrams to compare different application domains. Kiviat diagrams consist of a sequence of spokes put at equal angular distance representing, in this case, processor components. The length of the spoke is proportional to the usage of the component it represents. Lines are drawn connecting the values of each spoke creating a star shaped surface (see figure 4.1 for a few examples that we will use later in this chapter). In principle, by observing them, characteristics such as dominant components and outliers can be found and comparison between domains can be conducted. Unfortunately, the origin of the results of Poovey's study is not clear, it look as if results have been averaged over a number of platforms and no further information is provided. We have therefore repeated the experiments with a greater level of detail and show results for three embedded architectures (details in Table 3.2) and the five application domains defined by EEMBC (Automotive, Consumer, Networking, Office, Telecom) using Kiviat diagrams similar to those in [Poovey, 2007]. The diagrams are shown in figure 4.1.

A comparison across architectures reveals significant differences in the utilisation of functional resources. For example, telecom applications show a relatively high proportion of SHIFT operations on the INTEL STRONGARM platform, however, no such effect can be observed on the FREESCALE MPC7410 and ARC 750D processors. At the same time the BRANCH unit on the ARC processor shows a higher utilisa-

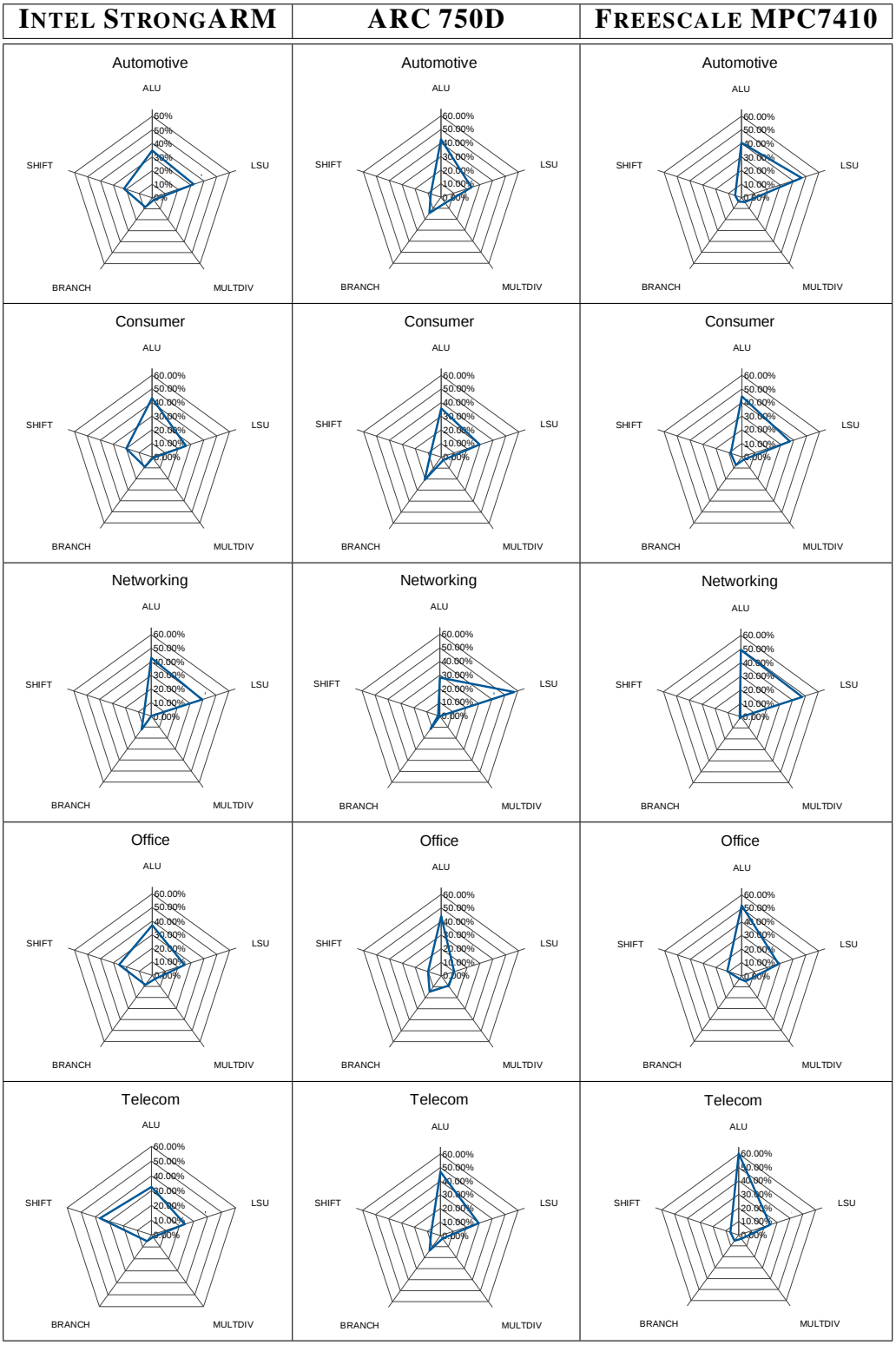


Figure 4.1: Kiviat diagrams showing the resource utilisation for the 5 different EEMBC benchmark domains (*automotive*, *consumer*, *networking*, *office* and *telecom*) and 3 embedded platforms (INTEL STRONGARM, ARC 750D and FREESCALE MPC7410).

Platform	STRONGARM SA-1110	ARC 750D	FREESCALE MPC7410
Instruction Set	all instructions	all instructions	all instructions
Branches	-	Branch Pred. Hits/Misses	Branches Taken/Pred./Always Taken
Memory			
• Instr. Cache	Reads/Read Misses	Instr. Cache Hits/Misses	L1 Hits/Misses
			L2 Hits/Misses
• Data Cache	Reads/Read Misses	Hits/Misses	L1 Read/Write Hits/Misses
	Writes/Write Misses	Dirty Misses	L2 Read/Write Hits/Misses
• Instr. TLB	Reads/Read Misses		Hits/Misses
• Data TLB	Reads/Read Misses		Read/Write Hits/Misses
• Other Mem.	Pages	Memory Reads/Writes	SDRAM Read/Write Accesses
			SDRAM Page Hits/Misses
Other	Instr. Count	Instr. Count	Instr. Count
	Cycles	Cycles	Cycles

Table 4.1: Overview of the program features gathered by the simulators.

tion than on the Intel and FREESCALE platforms. Interestingly, consumer applications exhibit a higher BRANCH utilisation on the ARC processor, but this not matched on the other two processors. There is no obvious correlation between the BRANCH frequency across architectures. On the other hand, visual inspection of the Kiviat diagrams reveals a number of similarities. The utilisation of the load/store unit (LSU) is significantly higher for networking applications across all three platforms. Furthermore, automotive and office applications on the STRONGARM result in similar patterns, and so do automotive and consumer applications for the FREESCALE processor and automotive, consumer and telecom applications for the ARC core, respectively. The diagrams suggest that certain application domains such as automotive and office show a similar resource utilisation for some processors, whereas other domains such as networking exhibit unique resource utilisation patterns. Furthermore, the data suggests that some of the characteristics differ across platforms (e.g. SHIFT for automotive), but other features are correlated (LSU for networking). However, it remains unclear what the underlying causes are for these different workload patterns. More importantly, it is not obvious if and how specific workload characteristics could be exploited for the tuning of domain-specific compilers.

4.2 Approach

We put forward the hypothesis that if we can successfully construct a *classifier* to distinguish applications from different embedded domains then the classifier *itself* is a characterisation of the corresponding application domains. The possible problem with

Grouped Features
ALU, LSU, Branch, Mult./Div., Shift, FPU
Memory Read/Write, Cache Reads/Writes
Arith., Logic, Move, Sign Ext., Bit Manip.

Table 4.2: Overview of the aggregated feature sets.

this approach is that many classifiers, e.g. artificial neural networks, are not open to human interpretation. Thus, we propose the use of decision trees for data mining that are simple to understand and interpret and have been shown to work well even with small data sets.

We test the above hypothesis and induce decision trees for the industrial EEMBC benchmark suite with its different application domains based on dynamic program features such as the instruction mix and cache utilisation for three popular embedded processors. In a cross-validation experiment we demonstrate that the induced decision trees are capable of accurately predicting the application domain and, hence, can be used to derive a meaningful domain characterisation. We investigated decision trees for characterising embedded applications from different domains, compared them with a clustering based workload characterisation methodology and demonstrated that the obtained workload characterisation can be successfully used to pinpoint performance bottlenecks and to eventually identify domain-specific compiler transformations.

4.3 Methodology

Classification tries to predict an attribute ("label") of a previously unseen data item based on other available attributes ("features") of this item. A popular classification algorithm that is used extensively in data mining and machine learning is the *decision tree*. The decision tree algorithm is easy to use and implement, has low computational complexity and, most importantly, its result is readily available for interpretation by non-statisticians despite its information-theory foundations.

A decision tree classifier is induced from a training data set where both the features and the labels are present. After this initial training period the decision tree can be used as a predictive model, i.e. to predict the label of a new data item where only the fea-

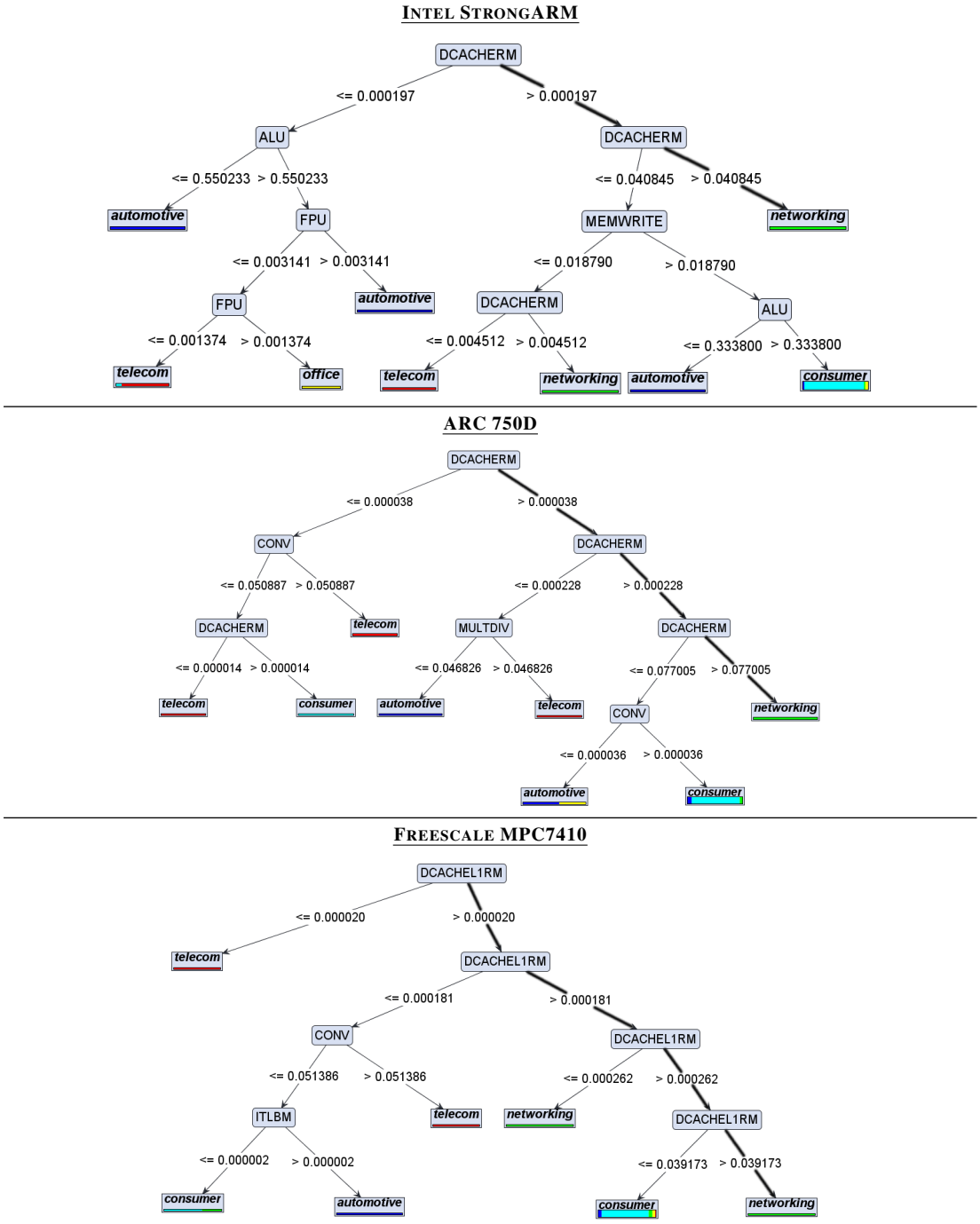


Figure 4.2: Decision trees induced for the EEMBC benchmarks (max. depth 5). Leaves represent application domains and branches represent conjunctions of features that lead to those domains. The highlighted paths refers to the *networking* domain that contains the *IP reassembly* benchmark discussed in section 5.3.2. Labels:

Label	Description	Label	Description
DCACHERM	Data Cache Read Misses	*MEMWRITE*	Memory Writes
CONV	Sign Extensions	*MULTDIV*	Multiplications/Divisions
DCACHEL1RM	Level 1 Data Cache Read Misses	*ITLBM*	Instruction TLB Misses

The thickness of coloured bars in leaves indicate how many benchmarks fall in that leave and colours indicate how many of each group do.

tures, but not the associated label, are given. However, we decided to use decision trees differently. Rather than using them for prediction we use the induced decision tree *directly* for workload characterisation. In our case, the features characterising a program comprise the instruction mix and other micro-architectural counters (see section 4.4.2) and the label corresponds to one of the application domains (e.g. "automotive" or "networking"). The decision tree then establishes a relationship between the observed counters and the membership to a particular domain.

Each internal node of a decision tree corresponds to a conditional expression relating to the value of a feature to a constant threshold. Such nodes are usually labelled with the name of the feature and the particular threshold value. Each branch departing from the node corresponds to possible intervals of the tested value. Leaf nodes, on the other hand, specify the label to be returned if that leaf is reached.

Following the path from the root of the decision tree to one of its leaves then produces a sequence of conditions such as "data cache miss rate greater than 10%" that characterise the items described by the leaf category (e.g. "automotive"). For categories that contain examples that differ significantly from each other, more than one leaf node may be generated (e.g. "automotive 1" and "automotive 2").

To build decision trees we use the standard C4.5 decision tree algorithm introduced by Quinlan [Quinlan, 1993]. During the construction of the tree the algorithm chooses for each node the exact attribute that splits the data most effectively into two subsets by maximising the information gain (difference in entropy).

4.4 Experimental Setup and Evaluation Methodology

In this section we provide a detailed outlook of our experimental setup and evaluation methodology as a complement to chapter 3.

4.4.1 Benchmarks

We use the EEMBC benchmark suite [The Embedded Microprocessor Benchmark Consortium, 2008] for the evaluation of our proposed workload characterisation method-

ology. Each of the 49 benchmarks belongs to one of following five application domains: *automotive*, *consumer*, *networking*, *office* and *telecom* (see chapter 3 for details).

4.4.2 Program Features

Programs are characterised by sets of numerical features collected from simulation runs using instruction set simulators of the three processor platforms considered in this research (see chapter 3). Essentially, the features correspond to counters maintaining instruction and micro-architectural event frequencies. The specific feature types for each architecture are listed in table 4.1. Since the total number of features for each architecture is a big set and each architecture has features that, at least in name, are different than the others we have decided to group the features by functionality (e.g. ALU groups *add*, *sub*, *mul*, *div*..., branch groups *be*, *bl*, *jmp*...etc.) similarly to the approach taken in [Poovey, 2007]. These grouped features are summarised in table 4.2. Contrary to [Poovey, 2007] and [Hoste and Eeckhout, 2007] the features considered in this research are *machine-dependent* as we primarily focus on the identification of platform-and domain-specific workload profiles.

4.4.3 Platforms

We have evaluated our workload characterisation methodology against the three popular embedded RISC architecture families introduced in chapter 3: INTEL STRONGARM SA-1100, FREESCALE MPC7410 and ARC 750D. The platform details (including software development tool chains) are summarised in table 3.2.

4.5 Evaluation Methodology

We have evaluated the accuracy of the decision tree characterisation approach using k-fold cross-validation. This technique partitions the data set into k subsets and, in turn, uses $k-1$ partitions to induce the decision tree that is subsequently evaluated

against the kth data partition. Cross-validation is essential to avoid *over-fitting* and at the same time provides a metric of how well the induced decision tree describes the observed applications and their domain membership. In the experiments we conducted we decided to use the special case of k-fold cross-validation called *leave-one-out cross-validation* in which k is set to the total number of observations [Bishop, 1995].

Following normalisation we use greedy, forward feature selection to automatically eliminate redundant and useless program features. The advantage of feature selection over principal components analysis (PCA), which has been used in e.g. [Hoste and Eeckhout, 2007], is that features are selected based on their significance in describing the observed data rather than just their variation. The selected features are also more amendable to human interpretation than the projected principal components resulting from PCA.

4.6 Results and Interpretation

In this section we present our results and compare the workload characterisations obtained from data mining with decision trees to those obtained from statistical clustering.

4.6.1 Characterisation by Decision Trees

The decision trees induced from the EEMBC benchmarks and features grouped by function are shown in figure 4.2. In addition, table 4.3 summarises the cross-validation results for decision trees built with feature sets (“all instructions”, “instruction groups”, with and without feature selection).

	StrongARM SA-1100	ARC 750D	Freescale MPC7410
All features	82.88%	73.10%	64.12%
All with feature selection	90.62%	85.09%	89.17%
Grouped features	85.62%	70.34%	77.86%
Grouped w. feat. select.	89.11%	85.09%	85.00%

Table 4.3: Cross-validation results for decision tree characterisation.

The most striking result is that the cross-validation accuracy of the decision trees is generally high, especially after feature selection. The accuracy is in the range of 85-90%, indicating that the program features can be used to successfully *predict* the application domain and, hence, the induced decision trees can be trusted as reliable domain characterisations.

The decision trees shown in figure 4.2 reveal a number of interesting facts. First, across the three platforms the single most important feature discriminating application domains is the rate of level 1 data cache misses. Networking applications, in particular, are characterised by their poor D-cache performance and most networking benchmarks can be distinguished from all other application domains based on this single feature alone. In chapter 5 we will first demonstrate how this information can be used to guide the compiler engineer towards a cache-aware code and data transformation specifically targeted at the networking domain that has little significance for a compiler targeting e.g. general-purpose applications. Secondly, automotive applications are spread over several leaf nodes characterised by broadly different feature sets. This indicates that the applications of the automotive domain do not exhibit homogeneous characteristics, but in fact comprise sub-domains with rather heterogeneous properties. For example, on the Intel StrongARM automotive applications either show low data cache miss rates, but high ALU and FPU activity, or a higher rate of D-cache misses and memory write activity, but lower ALU utilisation. This indicates that automotive applications are either compute-intensive kernels with good spatial and temporal locality or less compute-intensive applications with non-local memory access patterns. Third, misclassifications occur most frequently between automotive and consumer applications, indicating that these two domains share some characteristics that make them hard to distinguish. This can be seen in the coloured boxes attached to the leaves of the decision trees that represent the actual domains (as opposed to the predicted ones that are shown as labels of the leaves). Finally, telecom applications show low data cache read miss rates across platforms and at the same time perform only a modest number of ALU and very few, if any, FPU operations.

k	**StrongARM SA-1100**	**ARC 750D**	**Freescale MPC7410**
1	89.73%	84.14%	79.39%
3	88.36%	82.76%	80.92%
3 weighted	88.36%	82.76%	78.63%
5	84.25%	83.45%	77.10%
5 weighted	86.30%	82.76%	78.63%
1 feature selection	87.67%	88.97%	85.50%

Table 4.4: Accuracy of k-NN (with varying k and scaling) calculated using leave-one-out cross-validation for all architectures.

4.6.2 Comparison with Nearest Neighbour

To better understand the accuracy results obtained with decision trees we compare them with accuracy measurements obtained with another popular classification method, i.e. k-nearest neighbour (k-NN). k-NN is one of the simplest classification algorithms: it classifies objects to the class most common among its k nearest objects with "closeness" measured as euclidean distance between points [Bishop, 1995]. Using the most relevant features extracted by the decision tree construction method, we calculate the accuracy of k-NN for $k = 1$, 3 and 5. For 3 and 5 we also use distance as weight for the choice of the category and finally apply feature selection for $k = 1$ as shown in table 4.4.

Results show that the best accuracy with k-NN is obtained using different methodologies and number of neighbours k depending on the architecture taken into account though $k = 1$ and feature selection seem to give the best overall results. More important for us is that the accuracy is comparable to that obtained with decision trees, being globally between 85% and 90%. This shows that the decision tree approach, besides being easily interpretable, also keeps good accuracy compared with other methods.

4.6.3 Comparison with Statistical Clustering

Statistical clustering [von Luxburg and David, 2005] approaches to workload characterisation have been very popular in the last decades (e.g. [Raatikainen, 1993, Pentakalos et al., 1996]) and more recently have also been used to gain insight into benchmark

variation [Joshi et al., 2006] and to exploit similarity between programs for performance prediction [Hoste et al., 2006]. We compare our decision tree based approach to a recently proposed clustering methodology [Thomson, 2008] and use this, in particular, to study workload similarities *across* application domains and hope to gain additional insights to those domains such as *automotive* and *consumer* where decision trees have a below average classification success rate. Unlike classification, statistical clustering does not rely on known labels, but operates solely on the attributes describing the data items ("unsupervised learning"). Clustering seeks structure in the input data, finding clusters of input points which broadly share similar features. From this it may be possible to classify the input data into sets, according to their proximity to each cluster in the feature space. Juxtaposing the generated clusters with the known application domains (see figure 4.3) enables us to investigate both the spread of typical characteristics within a single application domain and the similarity of characteristics across domains. This information may be combined with the decision trees to identify sub-domains and domains sharing similar behaviour despite their different application contexts. In this research we use k-means clustering for workload characterisation [Bishop, 1995]. This technique is based on Lloyd's algorithm [Lloyd, 1982]. It starts by partitioning the input space into k initial sets and calculates the centroid of each set. It then constructs a new partition by associating each point with the closest centroid, after which the centroids are recalculated and the algorithm repeated until convergence. Inputs for the k-means clustering are the number k of clusters and the initial set. We have chosen k to be the number of leaf nodes generated by the decision tree characterisation and used a random set initialisation.

The known application domains and the generated clusters are combined and contrasted in figure 4.3. The y-axis of the scatter graphs represents clusters, whereas the x-axis corresponds to the application domains as defined by EEMBC benchmark suite. Each dot in the scatter graph represents one particular benchmark program. For any particular column in the diagram we can see how applications from a single domain spread over clusters with different characteristics. Each row, however, corresponds to a single cluster and shows how programs with similar behaviour can be found in different application domain. The bar plots next to the y-axis represent the coordinates of the corresponding cluster centroid.

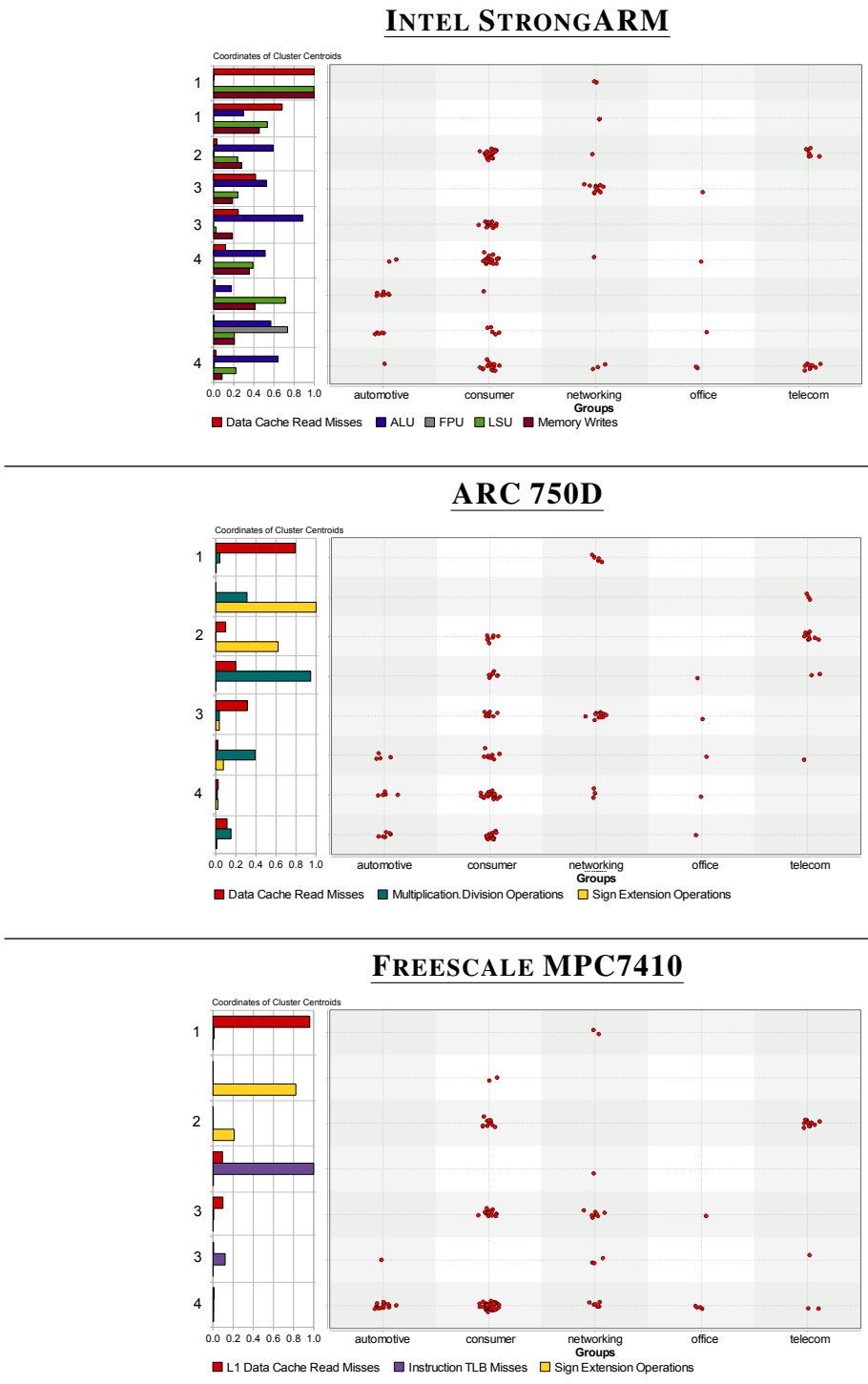


Figure 4.3: This scatter graph shows the relation between clusters obtained with k-means clustering and the EEMBC application domains. Some domains comprise several clusters indicating heterogeneous workloads in those domains. Similarly, some clusters span across domains indicating that similar workload characteristics can be found in different domains. The bar charts represent the coordinates of the cluster centroids. Numbers on the left of the bar plots on the y-axis are used as reference in section 4.6.3, which explains the graphs in detail.

Figure 4.3 reveals that application domains and clusters rarely coincide. For some domains such as consumer and automotive we see a spread over several clusters, whereas the majority of office, telecom and networking applications can be found in a small number of clusters with just a few outliers. Consumer applications typically span four to six clusters with vastly different characteristics. This is because the 88 benchmark runs contained in the consumer domain deal with different kinds of data (e.g. image, video processing, audio coding) and perform different types of algorithms (e.g. conversion, compression etc.). The "automotive" group with its 14 applications spans fewer clusters and these correspond roughly to the leaf nodes identified by the decision tree approach. Most of the applications from the networking and telecom domains fall into two to three distinct clusters that again have been previously discovered and characterised using decision trees. If we look at the same graphs horizontally, we notice that some clusters correspond only to one or two benchmark groups whereas others span (almost) all of them. For instance, clusters labelled "1", which show high data cache miss rates, always correspond to networking applications. Clusters labelled "2", which show lower data cache miss rates and medium ALU/sign extension instruction usage, correspond partly to consumer and partly to telecom. For example, the *Viterbi decoder* benchmark program ("telecom") is present in this cluster for all architectures. Across architectures the clusters are not necessarily homogeneous, e.g. cluster "2" includes *MPEG-2 decoding* ("consumer") on the INTEL STRONGARM, and *bit allocation* ("telecom") on the ARC 750D, however, on the FREESCALE MPC7410 it contains both of them. A subset of networking (in particular *IP packet check*, *packet flow* and *TCP*) and consumer benchmarks (mainly *RGB conversion*) are included in cluster "3". Less correspondence can be found for other clusters. For example, clusters labelled "4" spread over (almost) all groups for all architectures and contain some common benchmark programs but, except for the fact that all architectures show medium-low values for all features taken into account, no stronger statement can be made.

In summary, both decision trees and statistical clustering are capable of extracting defining characteristics of a set of programs. For example, the distinctive D-cache behaviour of networking applications has been picked up by both approaches. Similarly, the existence of sub-domains in the consumer application domain has been detected by both methods. Decision trees are a powerful instrument to identify and represent the differences between application domains. For the three platforms under consid-

eration, clear characterisations of the five embedded application domains defined by the EEMBC benchmark suite have been derived. Decisions trees appear more capable of representing differences across domains. The clusters generated by the statistical clustering technique instead are more abstract as application domain membership is not handled explicitly, but the clusters are represented by their (virtual) centroid and the specific data items contained in the cluster. Clustering also appears to be somewhat sensitive to the choice of parameters [Thomson, 2008] and requires some amount of tuning before the strongest results are produced. While clustering is useful to identify benchmark similarity across domains it appears not so well suited to represent the distinct characteristics of individual application domains. The combined consideration of decision trees and clustering approaches is promising, especially if clustering is used to further investigate the initial findings produced by decision tree data mining. We feel the possible "recipe"-like representation of the decision tree diagrams make them more favourable to compiler engineers who seek to identify performance bottlenecks and guidance in how to tune their compilers. Given that most compiler developers are typically not trained in statistics the graphical notation and rule-based interpretation of decision trees make them more accessible to this group.

4.7 Conclusions

In this chapter we have presented a methodology for an automated workload characterisation which may lead to the identification of tuning opportunities for domain-specific compilers. Our proposed method uses decision trees and produces human-interpretable results. Therefore, it can be used by compiler developers seeking to identify domain-specific performance bottlenecks and guidance on how to address these. We have demonstrated how decision trees can be used to detect the specific characteristics of five embedded applications domains. In the next chapter we demonstrate how the decision tree characterisation can be exploited to identify a missing domain-specific compiler optimisation for network applications. We demonstrate how our methodology directs the compiler engineer to a specific cache bottleneck, which can be resolved by applying non-standard data transformations.

Chapter 5

Data Transformations

In chapter 4 we have shown how to effectively use decision trees in conjunction with clustering techniques to point out the performance bottlenecks specific to a particular software application domain in terms of what differentiates one to the other. We discovered that the major issue for the network application domain lies in the predominance of data cache read misses. We therefore set up to investigate this aspect further by profiling network applications in detail and devise compiler optimisations to tackle high data cache misses.

In the context of NPUs, compilation techniques are certainly not a new area of research but much of it has focused on improved sequential code generation [Wagner and Leupers, 2002, Wagner and Leupers, 2001, Kim et al., 2002] and task partitioning and parallel thread management [Dai et al., 2005, Zhuang, 2006, Ramakrishna and Jamadagni, 2003, Ding and Liu, 2005, Chang and Kuo, 2009]. In this research we take a different approach and focus on *data transformations*. In chapter 4 we demonstrated that data accesses in network infrastructure applications form a performance bottleneck. In this chapter we set up to find the causes and evaluate the effectiveness of three *non-standard* data transformations: *structure splitting*, *array regrouping*, and *software caching*. These transformations aim at reducing the number of data cache misses which limit the overall performance on RISC based NPUs.

We evaluate their effectiveness against the industry standard EEMBC [The Embedded Microprocessor Benchmark Consortium, 2008] Networking benchmarks and two

embedded RISC processor cores (INTEL STRONGARM and ARC 750D). These processors are representative for many proprietary RISC cores of which more complex NPUs are composed. In fact, the STRONGARM is utilised in the first generation of INTEL's IXP network processor line and the ARC 750D can be found in many home networking devices.

Our results confirm that data transformations are key performance enablers for networking applications and speedups of up to 2.62 can be achieved through compiler-directed cache aware data restructuring. Rather than using *synthetic* and *unrealistic* data traces we use *real-world* network traffic and demonstrate that changing traffic scenarios require an *adaptive* approach, i.e. no single data layout and organisation performs equally well for every network traffic situation. This suggests that for the effective operation of NPUs *on-line* data transformations – under the control of the *control plane* processor and performed whenever a change in network traffic patterns is observed – may offer an attractive solution.

5.1 Motivation

Inspection of the induced decision trees depicted in Figure 4.2 of chapter 4 shows that networking benchmarks exhibit high data cache read miss rates across all the platforms considered. In fact, the decision trees ranks data cache miss rate as the most important criteria in distinguishing networking applications from the remaining domains, indicating we should focus our attention to the roots of this poor cache behaviour if we seek to improve the compiler performance for this domain. On the other hand other benchmark domains, such as automotive, exhibit much smaller cache miss rates and are more accurately described as compute-bound (StrongARM, ARC) or suffering from poor instruction-TLB (I-TLB) behaviour (MPC7410).

A simple data cache miss profiling across the five application domains (*automotive*, *consumer*, *networking*, *office*, *telecom*) of the industry standard EEMBC benchmarks for the three embedded processors considered (INTEL STRONGARM, ARC 750D, and FREESCALE MPC7410) confirms that networking applications show four to seven times higher data cache miss rates than for consumer or office applications (figure 5.1).

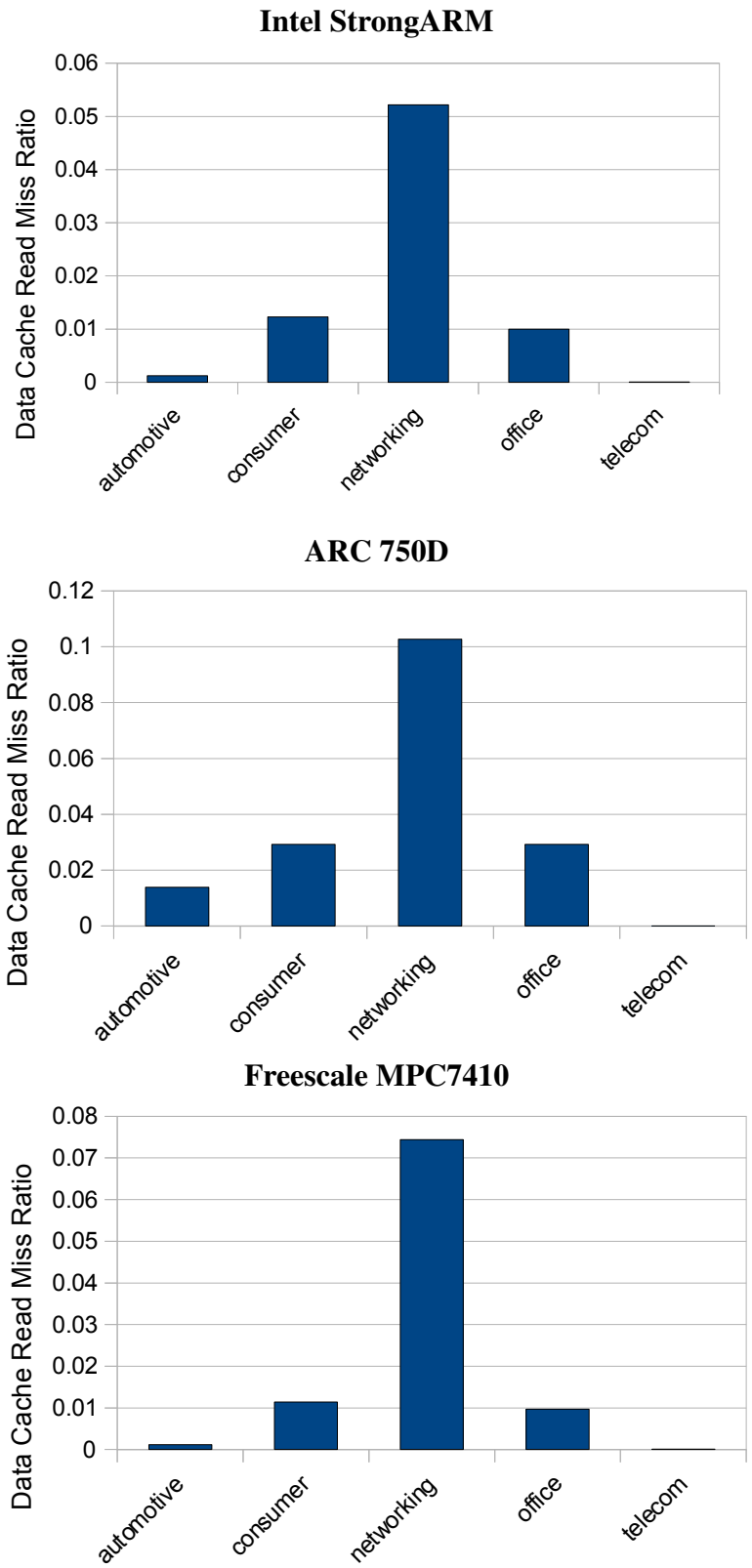


Figure 5.1: Average data cache miss rates across the five application domains represented by the EEMBC benchmarks and three different platforms.

Automotive and telecom applications, on the other hand, exhibit the lowest data cache miss rates across the three platforms. This observation cannot only be explained by the lack of temporal locality in packet processing benchmarks, but additional spatial locality issues contribute to the poor overall performance.

Hence, we perform a more detailed analysis of network applications. First of all we define exactly what kind of network applications we are looking at by differentiating between *data plane* and *control plane* applications and, with the help of two practical examples, we show how the latter show characteristics that are similar to control, rather than network applications. Then we carry out a memory access analysis on *data plane* network applications to identify the causes of the memory bottleneck shown in the decision tree workload characterisation analysis. This is also the way we expect users of our proposed workload characterisation to identify their compiler tuning opportunities: (a) Use the decision tree characterisation to pinpoint the critical resource, and then (b) use a specialised tool to perform a detailed analysis of that particular resource.

5.2 Control Plane Network Applications

Network applications for embedded systems can be divided into two distinct groups according to their workload, i.e. depending on how data-intensive they are. The decision trees shown above demonstrate that data plane network applications fall into two different leaves. These represent on one side programs like *IP packet check* or *TCP* that need to process a number of network packets in a very limited amount of time and which put a strain on the usage of the data cache and on the other programs like *route lookup* for which the data cache issue is less prominent. Nevertheless it has been shown that for both of them a high data cache miss rate is a bottleneck and data transformations are introduced to tackle that issue.

Control plane represents the other end of the spectrum of network applications. The goal of this section is to show where and to what extent they differ from data plane applications, i.e. to demonstrate that *control plane* software resembles management desktop applications more than *data plane* software and that compiler optimisations that are useful for the data plane are pointless for the control plane. We do this by

Data Cache Miss Ratio	Read	Write
INTEL STRONGARM	1.2%	0.3%
ARC 750D	2.9%	1.1%
FREESCALE MPC7410	1.8%	0.5%

Table 5.1: BSENSE data cache read/write miss ratios for the 3 architectures INTEL STRONGARM, ARC 750D and FREESCALE MPC7410

demonstrating how two different control plane applications, i.e. STIX, a management system for broadband wireless access network [Bernardi et al., 2010] and BSENSE, a system for automatic broadband census [Bernardi and Marina, 2010] show behaviour that make the decision trees wrongly classify them as non-networking applications.

STIX is a distributed system that provides an easy to use management instrument for wireless network providers. It is based on agents running on an array of embedded devices scattered across the network. Workflows and messages coded in XML format are exchanged between these agents which are also responsible for interpreting these workflows, running them when required and forwarding logs and results to other agents. The agent software performs management tasks and is therefore to be considered part of the control plane category of network applications.

On the other hand BSENSE is a system that performs customisable broadband connection measurements using a multi-platform distributed agent software. To perform the measurements, the application must be downloaded, installed on a user's computer and started. After a quick setup where the user is asked for location and provider-specific information, the application starts performing periodic remotely customisable broadband speed and latency tests and sends the results back to the server when finished. Although similarly based on an agent, the tasks that the BSENSE application performs and the platforms it runs on are substantially different from the STIX ones. Nevertheless they are still to be considered as belonging to the control plane as they don't require particularly stringent timing constraints to be met.

For this second control plane application we were able to perform the workload characterisation in a similar way to the one we performed with data plane applications. In particular we were able to simulate the data cache behaviour of all three architectures we took into account for the workload characterisation, INTEL STRONGARM, ARC 750D and FREESCALE MPC7410. We let the application run for one hour and

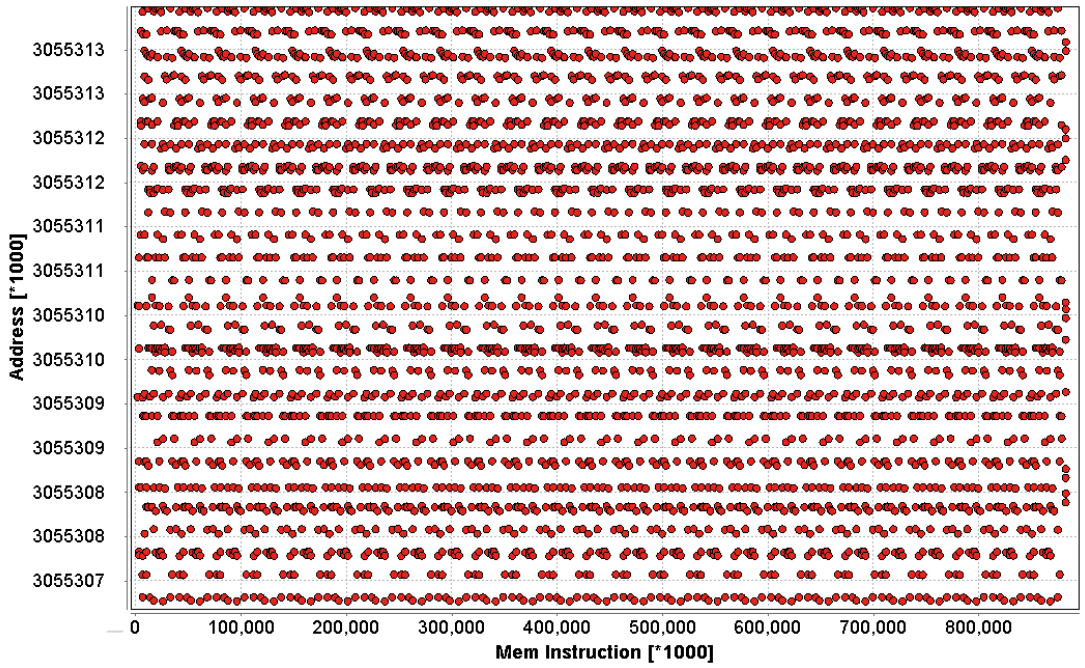


Figure 5.2: Trace graph for memory reads for *IP reassembly*.

collected the data cache miss rate results shown in table 5.1.

With this data we tried to classify the application using the decision trees created in the chapter 4 and shown in figure 4.2. By following the path in the decision trees from the root to the leaves we discovered that they classified BSense either as *automotive* or *consumer* for all platforms (in order to do that for INTEL STRONGARM we were able to extract the memory write instruction ratio, which is 0.13). This therefore confirms that *control plane* network applications behave in ways that are substantially different to *data plane* ones and more similar to *consumer*, *automotive* or other general purpose applications.

5.3 Cache-Aware Data Transformations

Following the results of the workload characterisation for network applications, we explore possible optimisations to be applied to the source code that would increase their performance.

```
struct mbuf {
 struct m_hdr m_hdr;       /* header */

 ...
 union {
  struct m_ext MH_ext;     /* link to external payload */
  char MH_databuf[MHLEN]; /* internal payload */
 } M_dat;
 ...
}
```

Figure 5.3: Original packet structure in the *IP Reassembly* benchmark.

5.3.1 Overview

In the previous sections we determined that the major issue that has to be addressed is the poor temporal locality of networking applications. To further investigate the causes of the distinctly higher data cache miss rate for networking applications we use the PIN tool [Luk et al., 2005] to trace their memory accesses. PIN instruments the executable code of an application by adding tracing instructions each time a memory access is performed. A memory trace resulting from the EEMBC *IP Reassembly* benchmark, but typical for other networking benchmarks such as *IP Packet Flow*, *IP Packet Check*, *IP Network Address Translator*, *Quality of Service* and *TCP* is shown in figure 5.2.

It reveals that memory is frequently accessed at discrete addresses with large unused gaps in-between. This leads to a characteristic regular access pattern with discrete horizontal stripes. The gaps between repeatedly accessed addresses are just 256 bytes wide and never subsequently accessed but nevertheless pollute the data caches. With this additional information available we have inspected the application source code and found that IP packets are stored in a single C structure containing separate elements for the IP header and the payload.

5.3.2 Structure Splitting

Consider the structure definition in the example in figure 5.3. This structure is an excerpt from the EEMBC *IP Reassembly* benchmark and defines the data structure for packets that carry a payload. Other networking applications feature similar packet data types. Among other fields the `mbuf` structure comprises an `m_hdr` structure for the packet header and an `M_dat` union holding either the internal payload or a link to an

```
struct mbuf_header {
 struct m_hdr m_hdr;       /* header */
}

struct mbuf_payload {
 ...
 union {
  struct m_ext MH_ext;     /* link to external payload */
  char MH_databuf[MHLEN]; /* internal payload */
 } M_dat;
 ...
}
```

Figure 5.4: Packet structure definition after *structure splitting*.

external payload.

The *IP Reassembly* application accepts previously split packets, reassembles and arranges them back in the right order which might be different from the order in which the pieces have been received. For this, *IP Reassembly* only needs to access the headers of each received packet, but it does not need to touch the internal payload. Still, when executed on a system with data caches, unused parts of the remaining `mbuf` structure including the internal payload are streamed through the memory hierarchy. This is because the header information used by the application is interleaved with payload data. As valuable cache space is taken up by data not touched by the application the resulting data cache utilisation is low and a high number of data cache misses can be observed.

Structure splitting is a data transformation that decomposes a structure into its components and modifies the use sites accordingly. Algorithm 1 illustrates how we can apply it to packet processing networking applications. The result of structure splitting

Algorithm 1 *Structure splitting* for packet processing networking applications.

for all *structures s containing network packets* **do**
  $split(s) \rightarrow h/p$ // h for header, p for the payload
  for all *buffers bs storing styped elements* **do**
    create bh/bp *to store* h/p *elements*
    remove bs
  end for
  for all *use of s and bs* **do**
    change to use h/p, bh/bp *respectively*
  end for
end for

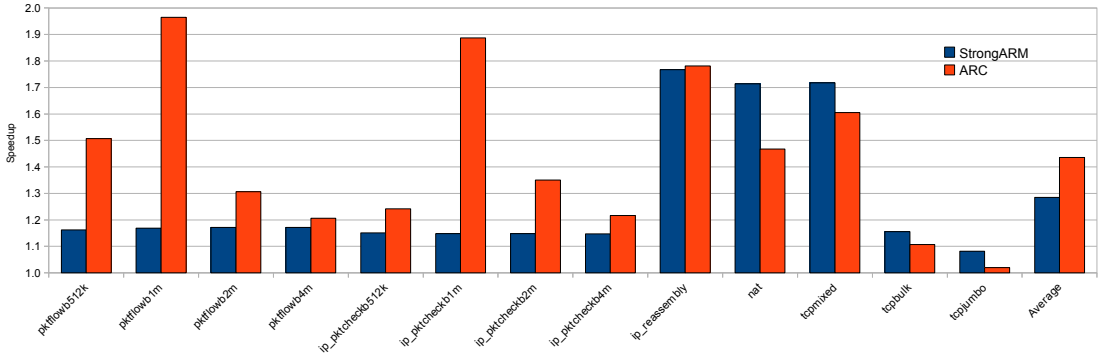


Figure 5.5: Speedups for EEMBC networking after structure splitting on the STRONGARM and ARC platforms.

applied to the packet data structure from the *IP Reassembly* benchmark is shown in figure 5.4. The original `mbuf` structure has been split into two structures `mbuf_header` and `mbuf_payload` that contain the header and payload data, respectively. The resulting code leads to fewer data cache misses as header and payload information are not interleaved any more and payload data does not further pollute the data caches.

For the transformed *IP Reassembly* code the data cache hit rate increases from 69% to 78% for the INTEL STRONGARM platform and from 92% to 94% for the ARC 750D. This results in overall speedups of 1.74 and 1.78, respectively.[1]

We have subsequently applied structure splitting to the complete set of EEMBC benchmarks and found that this transformation is applicable to the remaining networking codes in the same node of the decision tree as *IP reassembly*. Surprisingly, structure splitting is genuinely networking specific and ineffective for *any other* application contained in the EEMBC suite. The full set of results for the networking applications is shown in figure 5.5 where an average speedup of 1.27 for STRONGARM SA1100 and 1.44 for ARC 750D resulting from structure splitting can be observed.[2] Varying speedups between architectures are mainly due to the different cache organisations of the two targets. Variations in overall performance improvements for a single target originate from different buffer sizes and how well these match the cache line size.

[1] Relatively small cache hit increase results in quite big speedups due to the particularly high penalty for cache misses

[2] FREESCALE MPC7410 has been left out due to the complicated procedure to obtain results. See section 5.5.2.

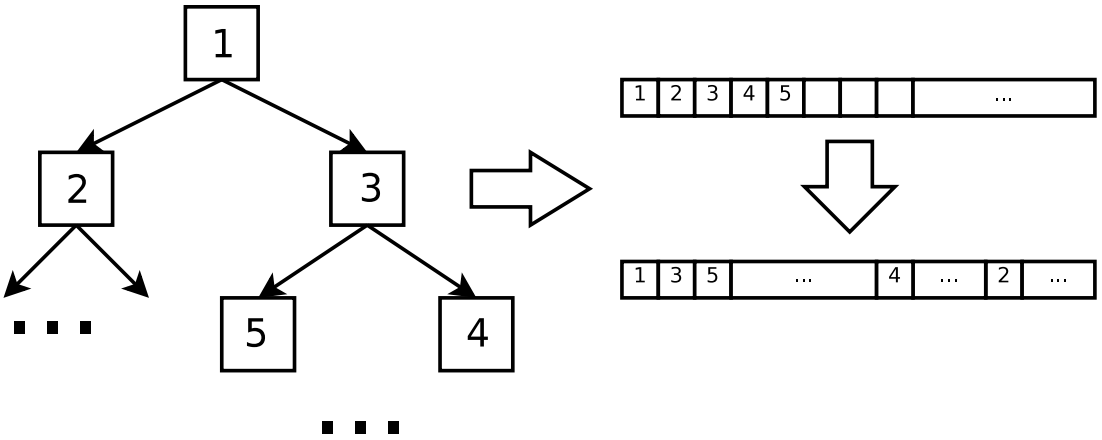


Figure 5.6: Example of *array regrouping* of a tree structure.

5.3.3 Array Regrouping

With the *array regrouping* transformation we try to improve the flat memory organisation of dynamically created data structures such as trees or graphs. Traditionally, tree or graph based data structures are heap allocated and consequently their actual memory layout is not only affected by the order in which nodes are inserted, but also by the details of the specific implementation of the `malloc` library function and the current state of the heap plays an important role, too. It is therefore little surprising that the resulting flat data organisation is not necessarily optimal with respect to its cache behaviour and, in particular, does not respect dynamic access patterns. Estimated reference affinity cost evaluation based on frequency profiling [Zhong et al., 2004] can lead to a better layout. First, if the data structure is *bound* and *quasi-static* it can be embedded in a fixed-size single-dimensional array. This is the case for most lookup structures such as routing tables and an example of this transformation is shown in figure 5.6. Second, taking into account dynamic access frequencies collected in a profiling pass an element ordering can be determined that reduces overall access cost due to cache misses [Zhong et al., 2004]. For example, if node 5 is repeatedly accessed along the path $1-3-5$ the layout on the bottom half of the right side of figure 5.6 has better spatial locality than the default layout above. Algorithm 2 shows the details of the procedure.

Algorithm 2 Implementation of *array regrouping* for networking applications.

for all *tree structures t* **do**
determine the max size of t
insert creation of array a to store elements of t
insert rearrangement of insertion order according to profiling information
change every malloc with reference to next element in a
end for

5.3.4 Software Caching

This transformation exploits different access frequencies to individual leaf nodes in a tree based lookup data structure and maintains a small, but efficient auxiliary data structure serving the most frequently accessed items. The operation of the auxiliary data structure is comparable to a *cache* and may be implemented using a hash table [Aggarwal, 2002].

This principle is shown in the diagrams in figure 5.7. In a lookup of a tree shaped data structure such as a routing table, node 3 may get frequently accessed. Upon determining that node 3 is *hot* it is entered in to the auxiliary hash table introduced next to the original tree structure. Subsequent accesses to the lookup tree are modified such that first the hash table is checked and a full traversal of the tree is only performed if the requested data item cannot be found in the hash table. The benefit of installing an auxiliary hash table arises from the fact that most queries can be served efficiently by the hash table and fewer accesses need to be resolved by the tree. The tree is not entirely replaced with a different data structure and can still support algorithms relying on depth-first or breadth-first order tree traversals.

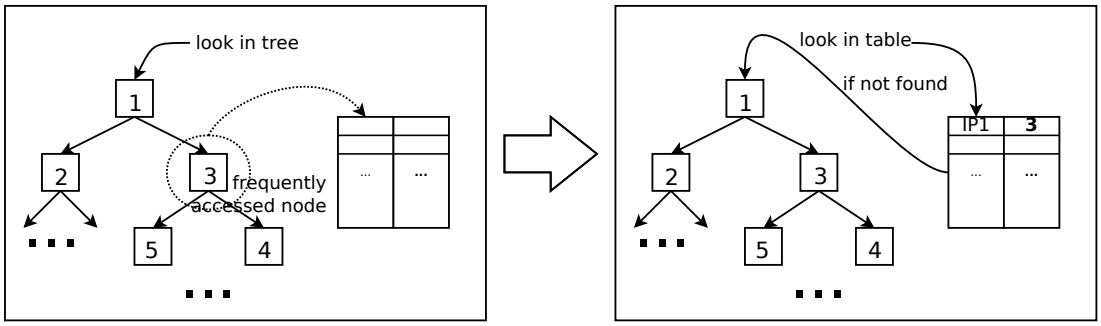


Figure 5.7: Example of *software caching* using a hash table supporting a tree structure.

Benchmark	Description
IP Packet Flow	IP packet store&forward
IP Packet Check	IP packet header check
IP Reassembly	IP fragmentation and reassembly
IP Network Address Transl. (NAT)	Network address translation
Route Lookup	IP route lookup
Open Shortest Path First (OSPF)	Dijkstra shortest path first alg.
Quality of Service (QOS)	Bandwidth and traffic shaping
TCP	TCP data transfer

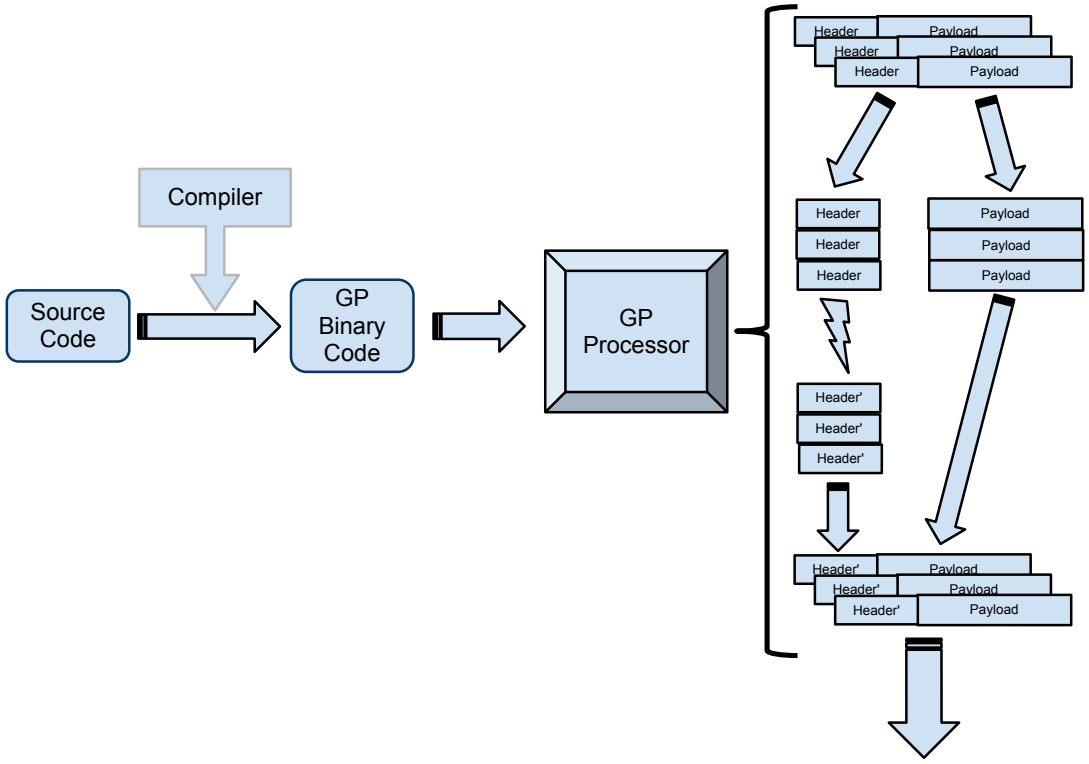
Table 5.2: EEMBC 1.1 & 2.0 networking benchmarks.

There are a number of prerequisites before auxiliary lookup data structures can be introduced by the compiler. First, shape analysis [Ghiya and Hendren, 1996, Lattner and Adve, 2003] is performed to determine the shape of the original data structure. In this work we restrict our transformation to binary trees. Second, in a profiling stage the most frequently accessed nodes and paths are determined. Third, the auxiliary hash table is created and read-only accesses to the original tree are augmented with prior lookups to the newly introduced table. Similarly, write accesses to the tree are augmented with corresponding updates to the table.

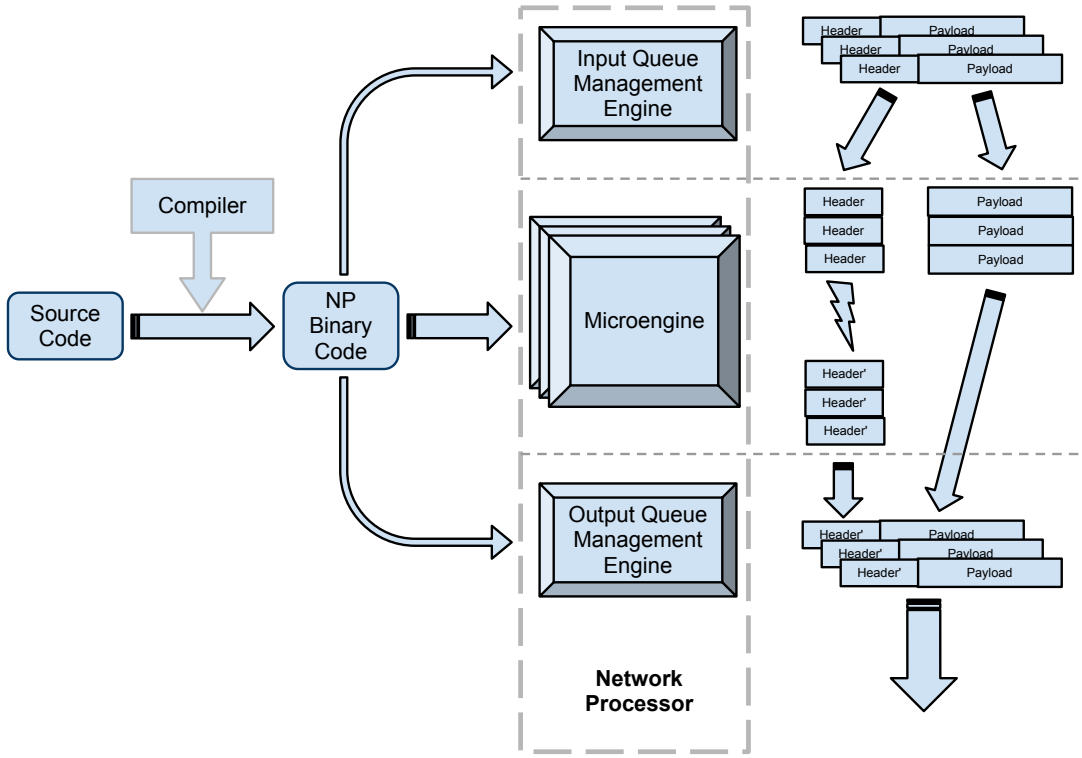
5.4 System View

In order to put the proposed data transformations into a system-wide perspective we need to specify where and how to implement them and how to integrate them with existing software and hardware. The system integration of *array regrouping* and *software caching* doesn't present any particular issues as they don't directly affect the underlying system real-time network packet handling.

For instance *array regrouping*, that handles a relatively small and static set of data (e.g a routing table), only involves rare copy or move operations when compared with the amount of packet data that would flow through the same network. The operations necessary for the data reorganisation can easily be added to the running code by a smart compiler and can dramatically increase the data locality (and consequently the cache performance) of the network application and largely compensate the additional costs of reorganisation operations. Similarly *software caching* doesn't affect the fast paced



(a) *Structure splitting* on general purpose processors



(b) *Structure splitting* on network processors

Figure 5.8: *Structure splitting* handling with general purpose processors and network processors including specialised input and output queue management engines.

handling of network packets since the management of the cache is performed very rarely and on a very limited amount of data. Additionally both transformations can be handled completely offline with respect to the network packet flow if an additional control plane processor is present (as is the case of most network processors).

However a different rationale lies behind the adoption of *structure splitting* since this transformation involves the interaction with the core of packet processing. In this case we have to distinguish between two scenarios: in the first one packet processing takes place on a general purpose processor, e.g. ordinary Linux machines as is becoming more and more common. In the second one, special hardware is available that handles network header splitting from the payload as proposed in previous research studies (e.g. [Huggahalli et al., 2005]) and as increasingly available on commercial network processors (such as Freescale DS4080). In the first case structure splitting can still be beneficial even if it involves reorganising part of the memory content of the network stack. Despite the fact that splitting takes precious computing time away from packet processing, it can be compensated by the time saved by avoiding unnecessary cache pollution while processing the data. In the second case the outcome of the *structure splitting* analysis can be used to drive the programmable engine of the network processor to perform data separation and rearrangement at no cost at the most suitable point in the packet processing computation. The difference is illustrated in figure 5.8.

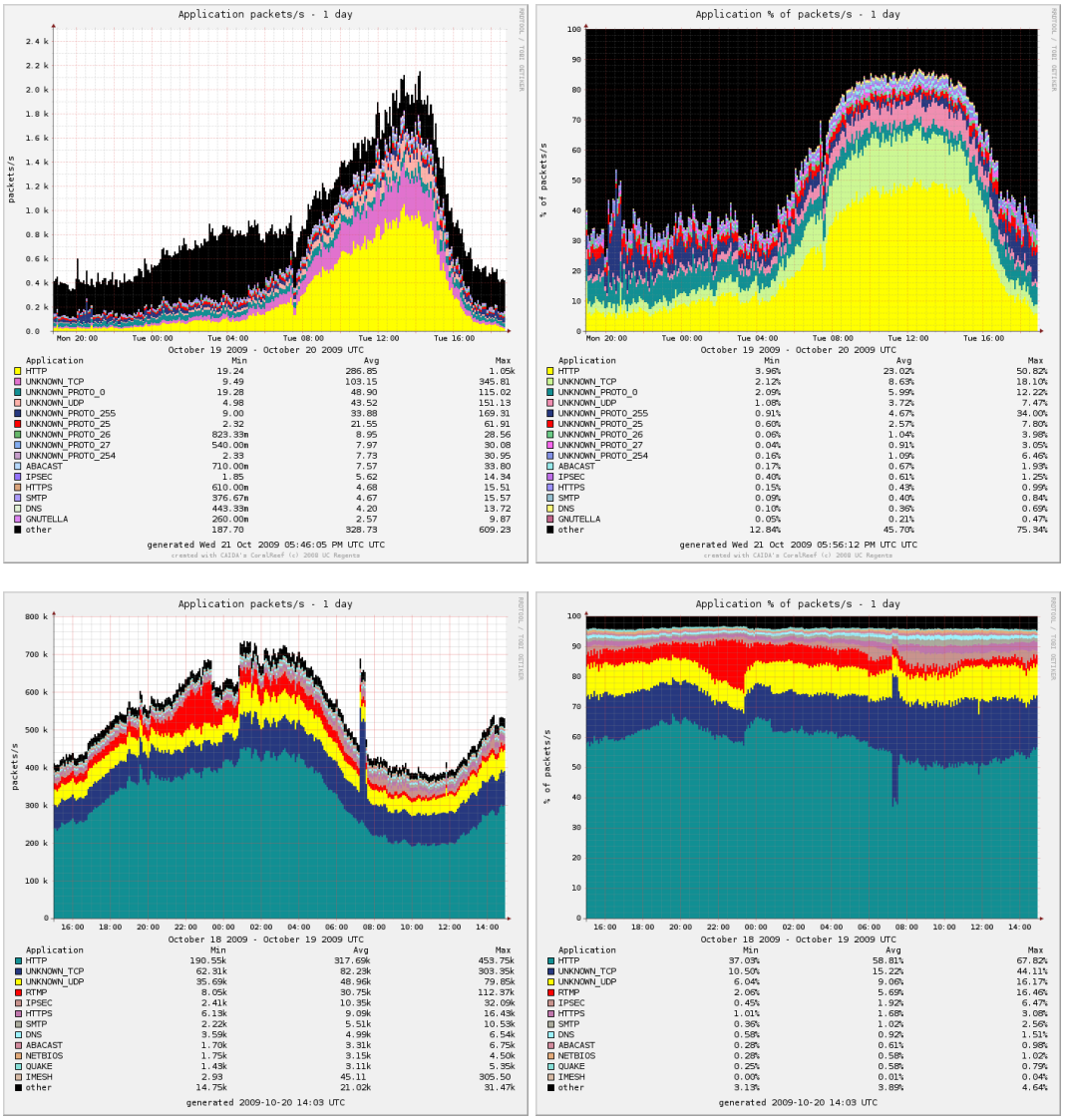


Figure 5.9: Graph showing the variation of traffic volume and type in CAIDA Equinix datacentres in Chicago and San Jose during 24h between 19/10/2011 and 20/10/2011. The graph clearly shows how traffic differs substantially at different places in the network (2 different datacentres) and how the volume as well as the type of data flowing through a single datacentre changes dramatically during the day (different column lengths and colours in the graphs).

5.5 Empirical Evaluation

In this section we present the results of our experimental evaluation of the three data transformations introduced in section 5.3. Before we start, we briefly describe the settings of target platforms and benchmarks specific to the evaluation of the data transformations before we demonstrate the need for real-world network traffic data.

5.5.1 Benchmarks

EEMBC has defined a set of established benchmarks targeting different embedded application domains (see section 3.2). In table 5.2 we provide a short description of the relevant EEMBC networking benchmarks that have been used in this work.

Each of the benchmarks come with one or more *synthetic* data sets. While these data sets are suitable for processor performance evaluation they do not exhibit the temporal locality and variation over time that is characteristic for *real-world* network traffic. Therefore, we have used different, *publicly available* data sets representing *actual* Internet traffic and routing information in our experiments (see section 5.5.3 for a full analysis and description of new data sets sources).

5.5.2 Target Platforms

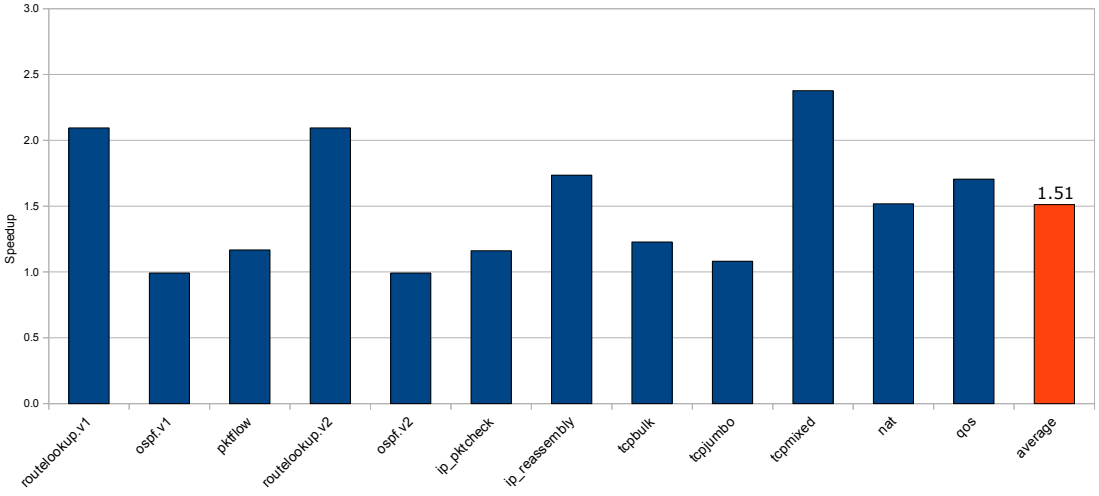
We have evaluated the data transformations against two of the embedded RISC processor cores introduced in chapter 3 (INTEL STRONGARM and ARC 750D). The details of the processors and their memory system, their simulators and the software development tool chains can be found in table 3.2 of that chapter. FREESCALE MPC7410 has been left out due to the complicated procedure to obtain results, which involved collecting tracing data from applications running on real hardware, and the unavailability of the hardware during later stages of our research. Nevertheless we believe that the use of the two available platforms doesn't detract from the strength of the results.

Source	Time	Description
CAIDA	31/03/2009 – 07:01	1 min traffic traces from the *equinix-chicago* backbone router [Walsworth et al., 2009]
	31/03/2009 – 12:28	IP Prefix to autonomous system map (routing table of backbone routers)
	13/09/2009	Topology of the IPv4 autonomous systems [Hyun et al., 2009]
UoE	12/01/2010 – 03,11,16:00	3 x 1 min traffic traces from the edge router of the Informatics School
	18/09/2009	Routing tables of the edge router of the Informatics School
	16/02/2010	Topology of the network of the Informatics School

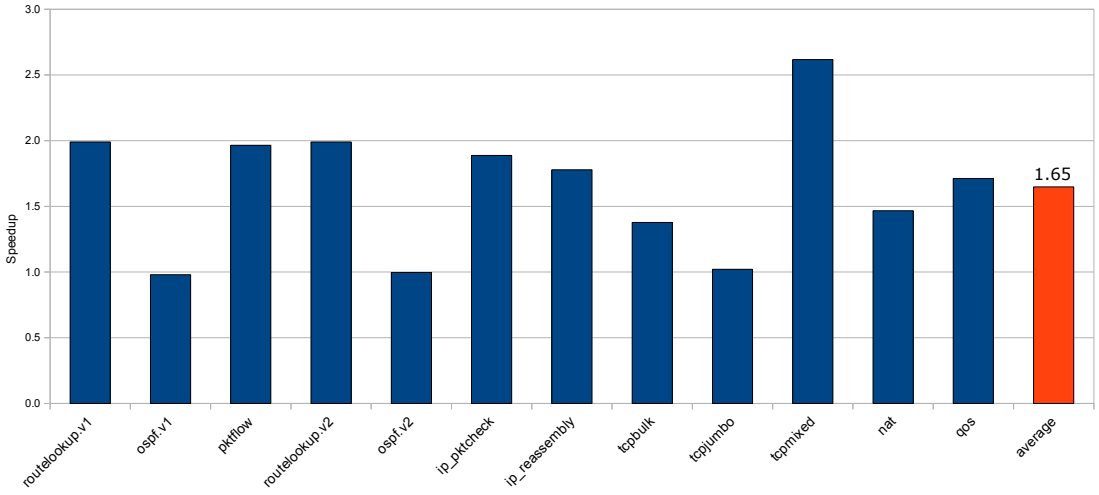
Table 5.3: New sources of input data for EEMBC networking applications.

5.5.3 Real-World Data Sets

For general processor benchmarking the *synthetic*, default data sets provided with the EEMBC benchmarks are convenient, however, they are less suitable for the evaluation of networking applications in a realistic environment. This is mainly due to two reasons: (a) the static data such as routing tables are very small and fit entirely into even small data caches, and (b) the traffic data is randomly distributed and lacks variation and temporal locality present in real network traffic. Example traffic during 24 hours at two different backbone routers provided by CAIDA [CAIDA, 2011] (shown in figure 5.9) makes it clearly visible that the amount and type of traffic varies over time and between different Internet routers. Fast transient variations are superimposed on underlying, slowly changing patterns representing different protocols. For this reasons we have decided to conduct our experiments using publicly available *real-world* data (see table 5.3). Large routing tables, network traffic variations over time and recurrent traffic patterns have been captured by CAIDA [CAIDA, 2011] for two Internet backbone routers. In addition, we used routing tables and network traffic captured at three different times at one of the fringe routers installed at the School of Informatics at the University of Edinburgh (UoE). Together, this allows us to accurately model variations in network traffic over different times of the day and for different locations on the Internet.



(a) INTEL STRONGARM



(b) ARC 750D

Figure 5.10: Overall performance results for all three data transformations and both platforms (INTEL STRONGARM and ARC 750D)

5.6 Results

Our overall performance results for both target platforms and all three data transformations are shown in figure 5.10 where each bar corresponds to the maximum speedup attained through either of the transformations. On average, data transformations contribute to average speedups of 1.51 and 1.65 for the INTEL STRONGARM and ARC 750D platforms, respectively. These results clearly demonstrate the potential of data

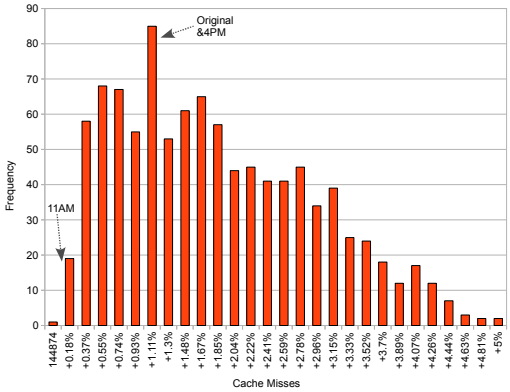
Benchmark	Structure Splitting	Array Regrouping	Software Caching
routelookup v1&2		✓	✓
ospf v1&2		✓	
pktflow	✓		
ip_pktcheck	✓		
ip_reassembly	✓		
tcp	✓		
nat	✓		
qos	✓		

Table 5.4: Applicability of data transformation to EEMBC networking benchmarks.

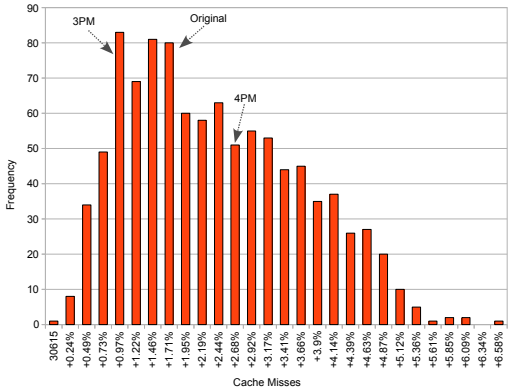
restructuring over the most aggressive code optimisations (-O3) performed by the default software development tool chains. In fact, on the *tcpmixed* benchmarks speedups of up to 2.37 and 2.61 for the INTEL STRONGARM and ARC 750D, respectively, have been observed.

In table 5.4 we show where transformations are applicable. *Structure splitting* is generally applicable to all packet processing benchmarks contained in the EEMBC suite. *Array regrouping* and *software caching* are only applicable to few benchmarks, however, *route lookup* and *shortest path (OSPF)* are among the most frequently performed operations in a network processor and, hence, any improvement of these operations increases overall system efficiency.

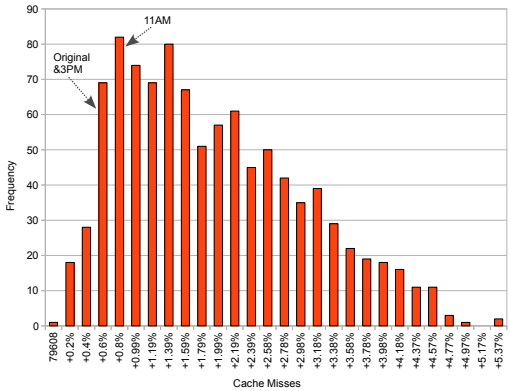
We now discuss each individual data transformation in turn. *Structure splitting* results in good speedups ranging from 1.02 to 2.61 across the applications where it is applicable and both platforms. While there is a noticeable correlation of the performance improvements between the two processors some benchmarks such as *ip_pktcheck* result in greatly different performance levels. This is mainly due to the particular data cache sizes and organisations (e.g. associativity) of the two platforms where a specific configuration aligns well with the structure sizes of the benchmark.



(a) UoE 3AM data set



(b) UoE 11AM data set



(c) UoE 4PM data set

Figure 5.11: Data cache miss frequencies of the EEMBC *routelookup* benchmark on the INTEL STRONGARM for the UoE datasets at three different times (3AM, 11AM and 4PM). The arrows represent the performance of the original data layout and the optimal layouts for the other two data sets (obtained from the runs with fewest cache misses in the other two frequency diagrams). The optimal data layout for one traffic scenario does not result in an optimal data layout when applied to traffic captured at the same router at a different time of the day.

Traffic \ Layout	**3AM**	**11AM**	**4PM**	**Original**
3AM		0.19	1.07	1.14
11AM	1.01		2.58	1.78
4PM	0.85	0.52		0.64

Table 5.5: Data cache miss increases compared to the best layout (in percentage).

5.7 Dynamic Behaviour

Array regrouping is most useful for the *route lookup* benchmark where speedups of 1.36 for the INTEL STRONGARM and 1.08 for the ARC 750D have been observed. For *OSPF* array regrouping is less beneficial and, in fact, has shown a slowdown. This is due to the failure of the *reference affinity* algorithm to accurately capture the common access patterns. For this reason we have conducted a further experiment investigating the impact of the data layout on the data cache miss rate. We have simulated 1000 runs of the *route lookup* benchmark with randomly created data layouts and plotted the resulting distribution of cache miss frequencies as shown in figure 5.11. The default layout highlighted in the diagram is at the peak of the resulting distribution. The bars to the left of this peak correspond to data layouts that result in fewer cache misses whereas the bars to the right related to data layouts with an increased number of cache misses. It becomes apparent that a significant reduction of cache misses over this baseline is possible if a suitable data organisation can be found. At the same time less-than-optimal data layouts can lead to a performance degradation. This is true for each of the three data sets evaluated in figures 5.11(a), 5.11(b) and 5.11(c) using traffic at three different times during the day (3AM, 11AM and 4PM).

We have now applied the best possible layout for each of the three data sets to the remaining two. For example, if the optimal data layouts for the UoE 3AM and 4PM data sets are applied to the 11AM network traffic data as shown by the arrows in figure 5.11(b) neither of the previously optimal layouts comes close to the optimal layout for the 11AM data set (at the left end of the graph). Whereas the 3AM data layout still improves slightly on the 11AM baseline the optimal layout for the 4PM data results in an increased number of cache misses when applied to the 11AM traffic. These results (also reported in table 5.5) demonstrate that no single data layout is optimal across different network traffic patterns, but optimality can only be achieved with respect to a

particular traffic scenario. In our future work we will address this issue and investigate dynamic data layout reorganisation to reflect changing access patterns.

The introduction of *software caching* shows a significant benefit for the *route lookup* algorithm for both architectures. However, the achievable performance improvements due to the caching in the auxiliary hash table vary across data sets. Speedups range from 1.11 for the 4PM UoE data set to 2.09 for the 11AM UoE network traffic data. This effect is due to the probabilistic nature of the processed network traffic where successful hits to the software cache depend on the specific traffic history. In the work carried out in this chapter we have used a very small hash table comprising only four elements. With more diverse traffic patterns such as the ones in the 4PM UoE data set larger sizes of the hash table might improve overall performance, but at the same time it remains important to trade off the potential benefits with the increased cost for a miss to the software cache. In the next chapter we will investigate different sizes and organisations of these structures and allow for dynamic adaption.

Our results for both *array regrouping* and the introduction of *software caching* show a strong dependence on the specific data set. For networking infrastructure applications the data sets are generally not known in advance, but we can rely on temporal locality and slowly changing patterns of the network traffic (see figure 5.9). It is therefore conceivable to perform online profiling and to use the continuously gathered information for periodic data restructuring to adapt to changing network traffic scenarios.

5.8 Conclusions

In this chapter we have shown that data cache misses form a performance bottleneck for network infrastructure applications running on RISC based NPUs. We have evaluated three non-standard data transformations (*structure splitting*, *array regrouping*, and *software caching*) that aim at improving overall cache efficiency for typical networking applications. For the EEMBC networking benchmarks, *real-world* data sets and two platforms we have demonstrate that speedups of up to 2.62 can be achieved. At the same time we show that changing network traffic patterns demand an adaptive approach to performing *online data transformation*. In the next chapter we therefore

investigate possibilities for continuous performance monitoring and efficient, on-the-fly data restructuring to be applied the data transformations we have introduced. In particular we focus on *software caching* as it shows the most promising adaptation opportunities. We introduce two different sampling policies to determine when reorganisation of data has to take place and we additionally propose two adaptation strategies for software caching parameters.

Chapter 6

Adaptation

The previous chapter has demonstrated how three different data transformations (*structure splitting*, *array regrouping* and *software caching*) can be used effectively to tackle the major performance bottleneck of network applications represented by their poor data cache utilisation. We showed that substantial speedups can be achieved by applying the data transformations to different network benchmarks but our analysis also indicated that for two data transformations, *array regrouping* and *software caching*, adaptation was necessary to keep performance gains high with all possible traffic environments.

In the next chapter we therefore investigate possibilities for continuous performance monitoring and efficient, on-the-fly data restructuring. In particular we focus on the *software caching* data transformation as it shows the most promising adaptation opportunities. We introduce two different sampling policies to determine when reorganisation of data has to take place and we additionally propose two adaptation strategies for software caching parameters.

Our results confirm that adaptive software caching is a key performance enabler for networking applications and speedups of up to 3.30 can be achieved through compiler-directed cache-aware data restructuring. Like in chapter 5, rather than using *synthetic* and *unrealistic* data traces we use *real-world* network but in this case we use traffic spanning several days to be able to better represent the network conditions over several days or months. We demonstrate that changing traffic scenarios require an

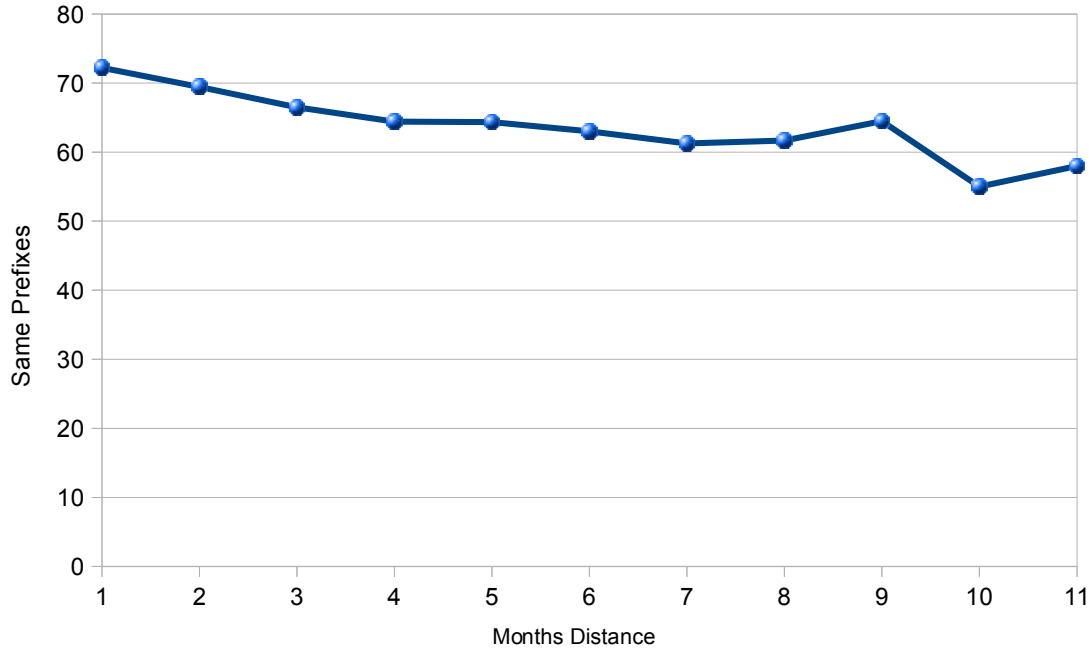


Figure 6.1: Average number of common routes in the set of the 100 most accessed ones for varying month distance.

adaptive approach, i.e. no fixed data organisation works well enough with a changing network traffic situation. This confirms that for the effective operation of NPUs *on-line* data transformations, under the control of the *control plane* processor and performed whenever a change in network traffic patterns is observed, provides an effective solution.

6.1 Motivation

The diagrams shown in figure 5.9 provide summaries of the network traffic at the Cooperative Association for Internet Data Analysis (CAIDA) Equinix datacentres in Chicago and San Jose [Walsworth et al., 2009]. For each of the datacentres the total traffic volume is shown in the diagrams in the left half of figure 5.9, whereas the diagrams in the right half show the distribution of traffic by network protocol. It can be clearly seen that both the traffic volume as well as the type of traffic changes over time for both datacentres. The largest traffic volumes can be expected between 8am and 4pm. In terms of network protocol usage fast transients are superimposed to underly-

ing, slowly changing patterns. Despite this, there is very little commonality between the traffic at the two datacentres. What the CAIDA traffic summaries suggest is that there exists temporal locality in the network traffic at each node. Intuitively, this means there is some correlation between packets processed in a certain window of time and caching of e.g. route lookup information could contribute to more efficient packet processing. Over longer period of times, however, network traffic changes. This is further supported by the graph in figure 6.1, which shows the average number of common routes in the set of the 100 most accessed ones in the CAIDA *equinix-chicago* router (we had access to route lookups of 1 day per month). The x-axis shows the distance in months between the 2 observed samples and the y-axis the number of route lookups in common between the 2 sets of 100 most common ones. What the graph unmistakably shows is that by growing time distance between the 2 samples, routing prefixes get more and more different. This strongly suggests that data transformation approaches such as software based caching need to be adaptive over time. At the same time, network traffic at different nodes is vastly different, such that additional spatial adaptivity is required.

6.2 Methodology

In this section we initially give an overview of our proposed software caching scheme, before we discuss this data transformation in detail. Next we discuss adaptation triggering, two policies for cache performance sampling, implementation issues and, finally, dynamic parameter adaptation.

6.2.1 Overview

Our approach applies to networking programs that use data structures such as linked lists, trees or a graphs mainly for data lookup operations. These are accessed using a key and return the node or element associated with that key. The *key idea* of our approach is to allocate a new auxiliary data structure with asymptotically faster access times next to the existing one and use it as a software cache. The new structure is filled with frequently accessed elements of the original data structure. When trying to

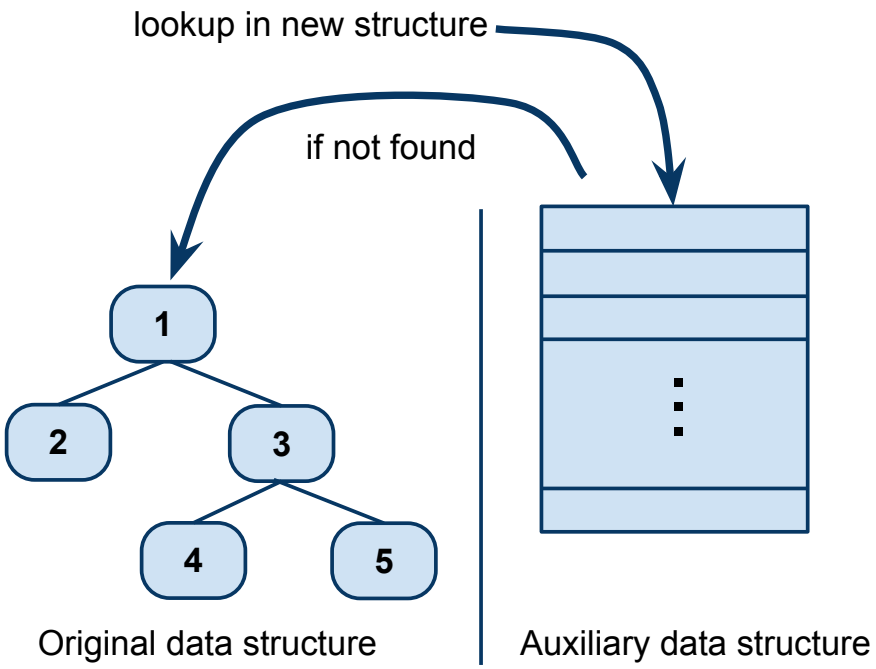


Figure 6.2: Example of a software cache supporting a binary tree based lookup data structure.

retrieve an element given a key, first the new data structure is searched. If the element is found we return it directly, otherwise we fall back to searching in the original data structure. Figure 6.2 illustrates this for a binary tree based data lookup operation. Overall, the behaviour of the extended data structure will be identical to the original structure, but either the asymptotic or actual access time (e.g. through improved cache behaviour) will be shorter.

Due to complex interactions between the application, its processed data and the underlying architecture it is not trivial to determine the optimal parameters (e.g. size) of the introduced software cache. Together with the changing nature of the network traffic data this motivates the use of an adaptive scheme, where a light-weight scheme for cache performance monitoring is used and cache parameters are modified adaptively. In the following paragraphs we introduce the basic transformation before we discuss two approaches to cache performance sampling and parameter adaptation.

6.2.2 Data Transformation

There are a number of prerequisites to be checked before auxiliary lookup data structures can be applied by a compiler. First, we need to identify the candidate data structures to be transformed later. This is done by means of shape analysis [Ghiya and Hendren, 1996, Lattner and Adve, 2003]. Second, the new software cache needs to be created, and read and write accesses to the original structure have to be augmented with prior lookups and updates to it as shown in algorithm 3. Third, the new structure has to be filled with elements that are accessed frequently. This can generally be done either with a profiling stage and an offline recompilation of the code or by dynamically adapting the elements in the new structure to recent lookups.

Algorithm 3 Auxiliary data structure integration

Original (without auxiliary structure)

Data Structures:
$original\_structure$ // Data structure as defined in original code

Lookup:
$element := original\_structure.lookup(key)$ // Lookup element in original structure

Update:
$original\_structure.put(key, element)$ // Update element in original structure

With auxiliary structure:

Data Structures:
$original\_structure$ // Data structure as defined in original code
$auxiliary\_structure$ // Additional auxiliary data structure

Lookup:
$element := auxiliary\_structure.lookup(key)$ // Lookup in auxiliary structure first
if $element \neq$ null **then**
  return $element$
else
  return $original\_structure.lookup(key)$ // If not found lookup original structure
end if

Update:
$element := auxiliary\_structure.lookup(key)$ // Lookup in auxiliary structure
if $element \neq$ null **then**
  $auxiliary\_structure.remove(key)$ // If found remove from auxiliary structure
end if
$original\_structure.put(key, element)$ // Update element in original structure

The natural candidate for the implementation of the auxiliary data structure for software caching is a hash table. This can easily be filled with the most frequently used elements and a lookup can be performed on it in O(1) asymptotic time. If reading from it does not incur any big overhead, the same cannot be said for writing. In fact since we aim at having two data structures that we use at the same time, we have to make sure that the data in both of them stays coherent. This implies updating the table every time a new node is added to the original structure. In a similar way to what happen in a hardware cache, we have to invalidate the hash table entry every time a new element that is present in the cache is updated. Though customary in hardware caches the same cannot be performed easily with software ones, since the computational overhead would be excessive. Instead, we adopted a simpler scheme that does not to take into account dynamic updates. This is justified by the fact that in the network applications changes to the routing tables are infrequent and can be dealt with by the control plane processor. Rather than to dynamically update the software cache on every access, we periodically refresh the software cache entirely as explained in section 6.2.3. Route lookup is the application that, given a network address, is responsible for finding the forwarding link of packet store-and-forward mechanisms in routers. It is present in a wide variety of routers, from small sized home access points to powerful backbone routers. In its most common implementation it uses a particular kind of binary tree, called *trie* as its core data structure. This trie is traversed at each lookup to find the outgoing port of a packet given its destination IP address. We use this route lookup application as a running example throughout the following experiments.

6.2.3 Adaptation Triggering

Due to the changing nature of networking traffic and to allow different software cache configurations (and contents) for routers deployed at different locations of the network we need the software caching mechanism to be able to adapt. First of all we identified two ways to determine when the adaptation has to be triggered, each of which is implemented with its specific cache sampling strategy described in section 6.2.4. One the one hand, we want quick adaptation to immediately follow changing trends in network traffic patterns. One the other hand, adaptation of software caches incurs a cost and we need to carefully balance costs and benefits arising from adaptation.

The first adaption strategy presented triggers if the software cache miss ratio grows above a certain threshold-ratio (e.g. 50%). The second strategy is slightly simpler and cheaper, but not accurate. It is based on a saturating miss counter that triggers if user-defined threshold is reached. The counter is increased when a lookup in the software cache is unsuccessful and decreased if successful (but saturates at 0).

6.2.4 Cache Performance Sampling Policies

We propose two different approaches for monitoring the performance of our software caches: *periodic sampling* and a *continuous sampling*.

Figure 6.3: Cache refill using periodic sampling.

The first strategy (periodic sampling) is illustrated in figure 6.3 and algorithm 4. During normal operation the hash table is used only to perform lookups for prefixes. After a certain number of lookups (*normal operations*) a sampling period is initiated. For a predefined number of lookups the hash table lookup misses are registered (*check*). At the end of this phase the miss ratio is computed and compared with a user-defined threshold. If the calculated miss ratio is below this threshold, a refill phase is triggered (*refill*), otherwise normal operation resumes. For a refill the hash table is flushed and completely refilled with the next IP prefixes encountered.

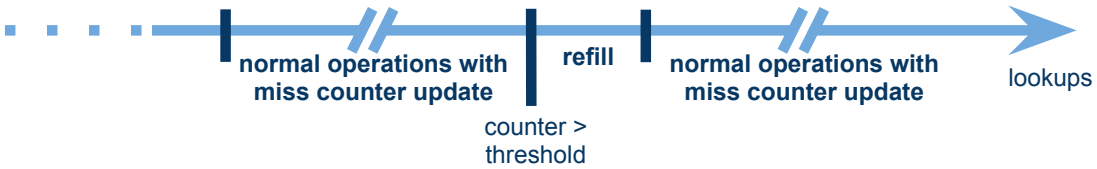


Figure 6.4: Cache refill using continuous sampling.

The second strategy (continuous sampling), shown in figure 6.4 and algorithm 5, continually keeps track of the number of hash table lookup misses by increasing and de-

Algorithm 4 Periodic sampling strategy to refill the cache

```
loop
  while normal_operations do                    // Normal route lookup operations
    element ← hashtable.lookup(key)             // Get element from the hash table
    if element = null then                      // If not there get it from the tree
      element ← tree.lookup(key)
    end if
  end while
  while checking_ratio do                       // Counting hash table hits
    element ← hashtable.lookup(key)
    if element = null then
      element ← tree.lookup(key)
      miss_count ← miss_count + 1               // Increase miss count if hash table hit
    end if
  end while
  miss_ratio ← miss_count/lookups               // Calculate miss ratio
  if miss_ratio > miss_ratio_threshold then     // Hash table refill if miss ratio above threshold
    hashtable.flush()
    while refilling do                          // Refill with the first elements looked up
      element ← hashtable.lookup(key)
      if element = null then
        element ← tree.lookup(key)
        hashtable.put(key, element)
      end if
    end while
  end if
end loop
```

creasing a counter by predefined constants (*normal operation with miss counter update*). The counter does not to decrease below 0. A new refill is triggered when it reaches a user-defined threshold. As with the periodic sampling strategy the hash table is refilled with the next prefixes encountered after initiation of a refill.

6.2.5 Implementation

For our implementation of the hash table we aimed at keeping dynamic overheads at a minimum. First we use a hash table with separate chaining to avoid costly collision detection. To compute array indexes we decided to rely on a simple shift and multiply function and feed it with the 24 highest bits of the address to be searched as shown in

Algorithm 5 Continuous sampling strategy to refill the cache

```
miss_count ← 0
loop
  while miss_count < miss_threshold do        // Iterate until miss count above threshold
    element ← hashtable.lookup(key)
    if element = null then                    // If hash table lookup not successful
      element ← tree.lookup(key)
      miss_count ← miss_count + inc_const     // Increase miss count
    else
      if miss_count > 0 then
        miss_count ← miss_count − dec_const // Decrease miss count
      end if
    end if
  end while
  hashtable.flush()
  while refilling do                          // Refill with the first elements looked up
    element ← hashtable.lookup(key)
    if element = null then
      element ← tree.lookup(key)
      hashtable.put(key, element)
    end if
  end while
  miss_count ← 0
end loop
```

figure 6.5. By doing this we deliberately aggregate all addresses with the same 24-bit prefix.

We allocate the hash tables globally and avoid reallocating it every time the table has to be refilled. The hash table is filled up to a load factor of 0.5. Slightly higher load factors (0.6–0.75) might fill the memory more efficiently without affecting performance. We nevertheless decided to use the $1/2$ ratio to facilitate checks and handling (usually obtained with simple shifts) and taking into account the fact that the size of the hash table index is irrelevant compared with the size of the data in the hash table and the size of the routing tree. Using a hash table also means that we need to specify a size for it. Instead of opting for a fixed size, we analyse the behaviour of our software caching strategies with different sized hash tables, i.e. with 2, 4, 8, 16, 32, 64 and 128 elements.

Additionally the implementation of the two sampling strategies presented in section 6.2.4 introduce an additional new set of variables to take into account (table 6.1). With *pe-*

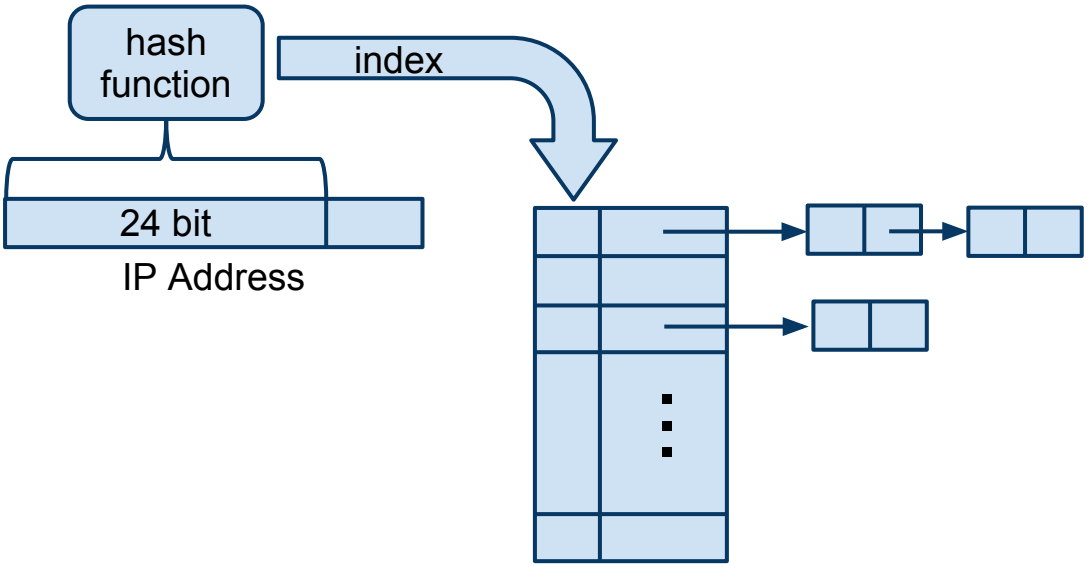


Figure 6.5: Hash table implementation details.

riodic sampling we have the number of lookups during normal operation before the sampling phase is started, the length of the sampling phase and the threshold that determines if a table refill has to be performed. For *continuous sampling* we can vary sampling threshold and the increment/decrement ratio. Each of these parameters needs to be carefully set in order to obtain the best results. As with the size, we decided to explore multiple values for each parameters.

6.2.6 Dynamic Adaptation

All parameters mentioned in the previous section can take on different values. The set of values we used in our experiments include a wide combination of parameter values but, as we will show in section 6.3.3, optimal results are highly dependent on network size, traffic and time. In order to achieve a complete adaptability of the data transformations we need to adapt the software caching parameters as well. Therefore we propose two different options for dynamic parameter adjustment and we apply them to find good lengths for normal operation in *periodic sampling* and for the miss threshold in *continuous sampling*, respectively.

The idea behind the first dynamic approach is that if, after checking the ratio we do not refill the table, it means that the table was still filled with “good enough” data. We can

	Parameter	Description
Periodic	Hash table size	Size of the array allocated for the hash table. Load factor is 0.5.
	# lookups normal operations	Number of lookups of normal operations before a new ratio check begins
	# lookups ratio check	Number of lookups during which the miss ratio counts are updated
	Ratio threshold	Miss ratio threshold that decides whether a refill is needed
Cont.	Hash table size	Size of the array allocated for the hash table. Load factor is 0.5.
	Miss count threshold	Miss count threshold that decides whether a refill is needed
	Miss count increase/decrease	Miss count increase (hash table miss) and decrease (hash table hit) constants

Table 6.1: Parameters for *periodic* and *continuous sampling*.

therefore increase the length of the normal operation phase since we can assume relative slower changes of network traffic patterns. Algorithm 6 summarises the procedure for an increment/decrement step of 2, which is the one we use in our experiments.

The second dynamic approach is applicable to a wider set of parameters since it does not imply a specific direction for the adaptation, but instead it relies on a trial-and-error method. For the time being we have applied it to adapt the the miss count threshold of the *continuous sampling* strategy, but the same underlying principle could be used to adapt most of the other parameters (e.g. hash table size and number of lookups used for ratio check), provided that there is a way to assess the quality of a parameter change (e.g. lookup miss ratio increase/decrease). As with the previous parameter adaptation method we are trying to find miss count thresholds that result in fewer hash table misses. However, we cannot easily estimate in advance if better results will be found with a higher or lower threshold. To overcome this problem we simply try one direction at a time. More specifically we try to double the threshold or halve it every four times we reach the threshold and refill the hash table. The next time the threshold is reached the miss ratio is computed and, if significantly higher than the previous one (e.g. twice as high), the new threshold is taken and a new threshold in the same direction is tested immediately (to reach the equilibrium quicker, especially at start time). If the threshold is not significantly higher, the previous one is restored and we

Algorithm 6 Dynamic adaptation of number of lookups in normal operation

```
...                                                    // See algorithm 4
if miss_ratio > miss_ratio_threshold then              // If miss count above threshold
   hashtable.flush()
   while refilling do                                  // Refill hash table
      element ← hashtable.lookup(key)
      if element = NULL then
         element ← tree.lookup(key)
         hashtable.put(key, element)
      end if
   end while
   #normal_operations ← #normal_operations/2          // Halve number of normal operations
else
   #normal_operations ← #normal_operations × 2        // Double number of normal operations
end if
...                                                    // See algorithm 4
```

wait for four more periods before the next threshold adaptation attempt is made (this time in the opposite direction). The whole procedure is summarised in algorithm 7.

6.3 Empirical Evaluation

In this section we present our benchmarks and platforms, experimental methodology and results.

6.3.1 Benchmarks and Platforms

To evaluate our software caching for networking application we use the EEMBC benchmark set. EEMBC has defined a set of established benchmarks targeting different embedded application domains and one of them includes networking applications (see chapter 3). The network application we focus on is *route lookup*, which implements the IP lookup mechanism that allow IP routers to forward incoming IP datagrams to the right outgoing port as introduced in section 6.2.2. Route lookup is found in all sized routers, from backbone to LAN, and has to be performed fast enough to exceed the speed at which the router can receive packets from all incoming links.

Algorithm 7 Dynamic adaptation of miss count threshold

```
loop
  while miss_count < miss_threshold do
    ...                                              // see algorithm 6.4
  end while
  if adapt_threshold = 0 then
    if (new_miss_ratio × 2) < old_ratio then         // If new ratio twice as good as old one
      if adapt_up then                               // If going up
        miss_threshold ← miss_threshold × 2          // Double miss threshold
      else                                           // If going down
        miss_threshold ← miss_threshold/2            // Halve miss threshold
      end if
      adapt_threshold ← 1                            // Wait 1 cycle until next adaptation attempt
    else                                             // If new ratio not much better or worse
      adapt_up ← ¬adapt_up                           // Reverse next adaptation direction
      adapt_threshold ← 4                            // Wait 4 cycles until next adaptation attempt
    end if
  end if
  adapt_threshold ← adapt_threshold − 1
  ...                                                // see algorithm 6.4
end loop
```

Route lookup is very suitable for applying our software caching technique because all its original implementations make use of the same family of binary trees (e.g. patricia trie).

We have evaluated our software caching approach against two embedded RISC processor cores (INTEL STRONGARM and ARC 750D). The details of the processors and their memory system, their simulators and the software development tool chains are shown in chapter 3. As for experiments conducted for *data transformation* (see section 5.5.2), FREESCALE MPC7410 has been left out due to the complicated procedure to obtain results, which involved collecting tracing data from applications running on real hardware, and the unavailability of the hardware during later stages of our research.

6.3.2 Experimental Methodology

For general processor benchmarking the *synthetic*, default data sets provided with the EEMBC benchmarks are convenient, however, they are less suitable for the evaluation of networking applications in a realistic environment. This is mainly due to two reasons: (a) the static data such as routing tables are very small and fit entirely into even small data caches, and (b) the traffic data is randomly distributed and lacks the variation and temporal locality present in real network traffic. For this reason we have decided to conduct our experiments using publicly available *real-world* data (see table 6.2). Large routing tables, network traffic variations over time and recurrent traffic patterns have been captured by CAIDA [CAIDA, 2011] for two Internet backbone routers. In addition, we used routing tables and network traffic captured at three different times at one of the fringe routers installed at the School of Informatics at the UoE. Together, this allows us to accurately model variations in network traffic over different times of the day and for different locations on the Internet.

Source	Time	Description
CAIDA	2009	Traffic traces throughout 1 year from the *equinix-chicago* backbone router [Walsworth et al., 2009]
		IP Prefix to autonomous system map (routing table of backbone routers)
UoE	12/01/2010	1 day traffic traces from the edge router of the Informatics School
		Routing tables of the edge router of the Informatics School

Table 6.2: Alternative sources of input data for EEMBC networking applications.

For each one of the two adaptation methods, there are a number of parameters that have to be set. Instead of fixing them to predetermined values, we decided to compare the results obtained applying all combinations of predefined sets for each parameter. Table 6.3 summarises the tested values for each parameter of both adaptation methods

6.3.3 Results

Our overall performance results for both target platforms are shown in figure 6.7 where each bar corresponds to the maximum speedup attained over both cache replace-

	Parameter	Values
Periodic	Hash table size	2, 4, 8, 16, 32, 64, 128
	# lookups normal operations	1, 10, 50, 100, 1000, 10000, 100000 1000000, 10000000, 100000000
	# lookups ratio check	20, 100, 1000, 10000
	Ratio threshold	30, 50, 70
Cont.	Hash table size	2, 4, 8, 16, 32, 64, 128
	Miss count threshold	5, 10, 20, 50, 100, 1000, 10000, 100000 1000000, 10000000, 100000000
	Miss count increase/decrease	1/1, 1/2, 1/3, 1/4, 2/3, 3/4

Table 6.3: Parameter range for *periodic* and *continuous sampling*.

ment policies and all parameters. By choosing appropriate parameters for the INTEL STRONGARM platform we can achieve a maximum speedup of 1.08 for the original EEMBC data set, 3.13 for the for the UOE data set and 2.12 for the CAIDA dataset. Slightly lower results are obtained with the ARC 750D platform as summarised by table 6.4.

Figure 6.6 shows how the speedup varies depending on parameter changes. The x-axes represent the number of lookups of normal operation for *periodic sampling* graphs and the miss count threshold for *continuous sampling* graphs whereas the y-axis represents speedups. The blue areas represent the interval between the best and the worse speedup for all parameters. Parameter values taken into consideration to plot the graph are the ones reported in table 6.3. The blue lines inside the blue areas represent examples of speedup results with specific fixed parameters of hash table size (64), number of lookups used to calculate the ratio (100), ratio threshold (50) and increase/decrease ratio (1/1).

	EEMBC	UoE	CAIDA
StrongARM	1.08	3.30	2.12
ARC	0.99	2.27	1.57

Table 6.4: Maximum speedups over all datasets and architectures.

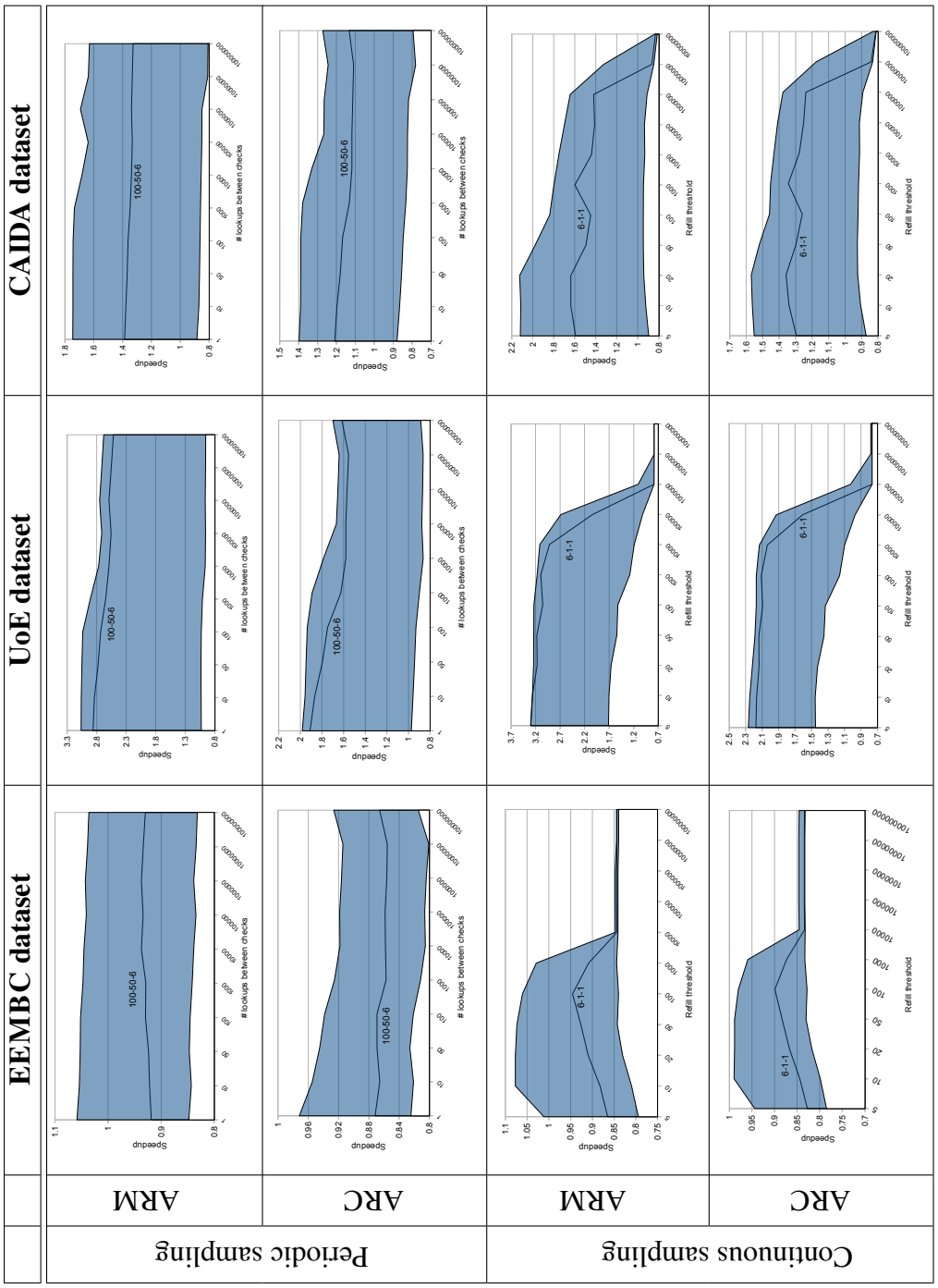


Figure 6.6: Range of speedups using the two different sampling methods (periodic and continuous) using three different datasets (EEMBC, UoE and CAIDA) on the StrongARM and ARC platforms. The x-axis represents the number of lookups of normal operation for *periodic sampling* and the miss count threshold for *continuous sampling*. The blue areas represent the interval between the best and the worse speedup for all other parameters and the dark blue line shows an example of speedup behaviour with fixed hash table size (64), fixed number of lookups used to calculate the ratio (100) (only for *periodic sampling*), fixed ratio threshold (50) (only for *periodic sampling*) and fixed increase/decrease ratio (1/1) (only for *continuous sampling*).

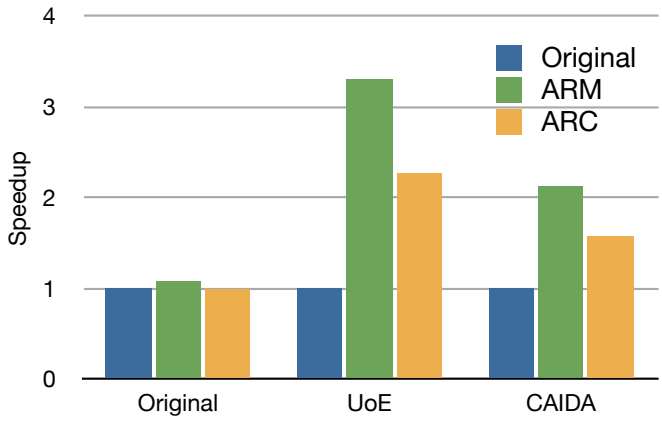


Figure 6.7: Maximum speedups achieved over all datasets and architectures.

In general ARM shows the best results. The gap with ARC is most likely due to the different data cache behaviour of the two architectures. While both of them have an 8KB data cache, StrongARM SA-1110, with it's 32-way has a far higher associativity than ARC 750D (4-way) and is therefore able to organise data in a better way. This is particularly useful if there are multiple data structures in different locations in memory that behave differently.

Taking a closer look at the periodic sampling results, we notice how generally there is a constant distance between maximum and minimum speedups (constant width of the blue area across the x-axis) but the speedup globally decreases with increasing number of lookups in normal operations. In other words, it is better to keep calculating the ratio continuously instead of doing it only from time to time. We can therefore infer that the overhead of calculating the ratio continuously is lower than the gain obtained by having a hash table filled at the right time. This is also supported by the results of the continuous sampling. In fact having few or no lookups in normal operation mode for periodic sampling is equivalent to having a continuous ratio computation. It is not surprising then to see how continuous sampling speedups are slightly better than periodic sampling, as increasing and decreasing its counter at every cycle is not as intensive as computing a ratio. Depending on the dataset, continuous sampling reaches its maximum speedup with a miss count threshold of 5 (UoE), 10 (EEMBC) or 20 (CAIDA). Increasing the threshold worsens the speedup and, more importantly, reduces the difference between the best and the worst speedup sharply. This is mainly due to the fact that if the miss threshold is high, it is (almost) never reached.

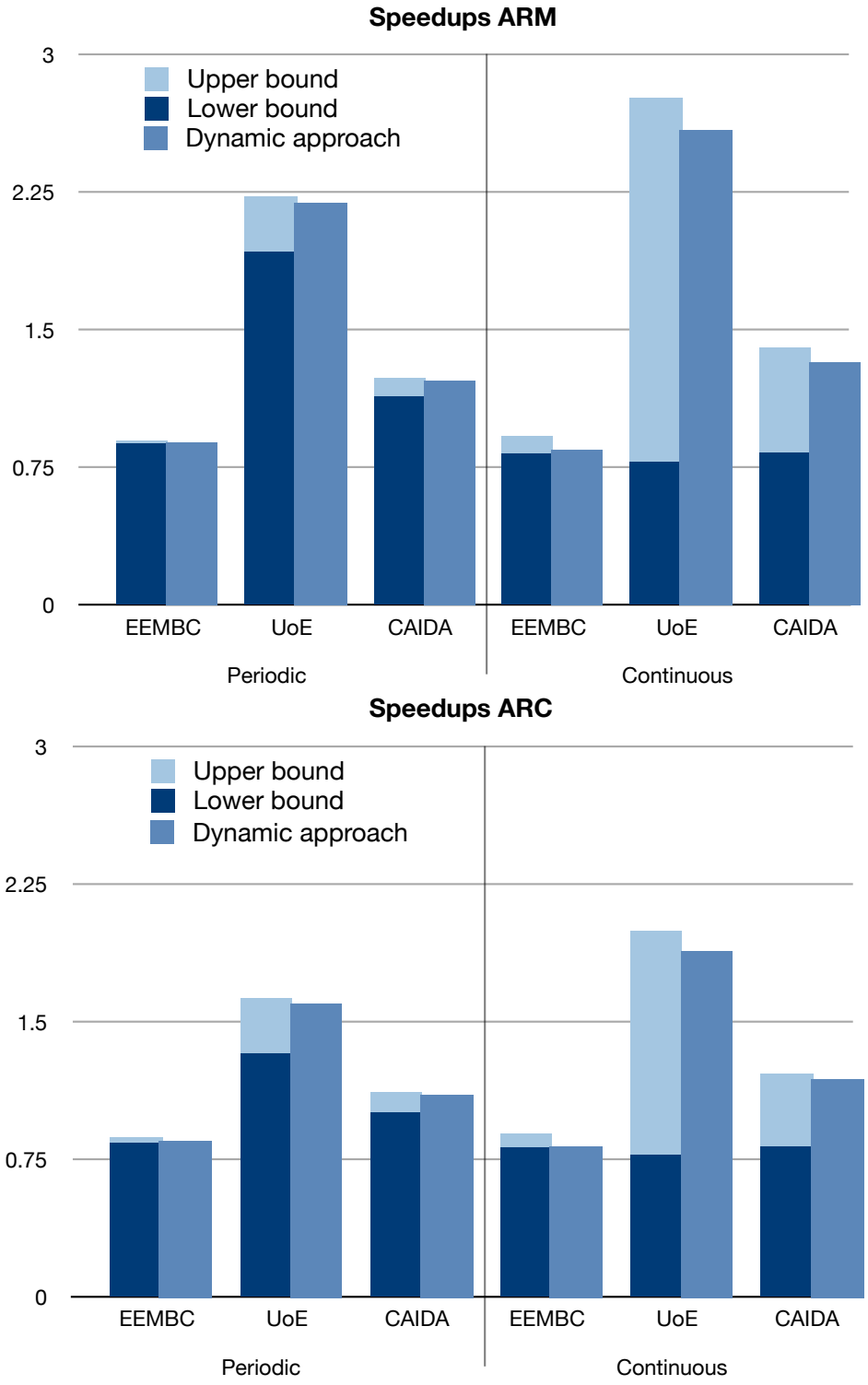


Figure 6.8: Comparison between hypothetical performance range (speedup lower bound/upper bound, left stacked bars) and performance of dynamic parameter approach (right bars). The dynamic approach results in speedups close to the hypothetical maximum.

		EEMBC	UoE	CAIDA
StrongARM	Periodic	1.05	3.06	1.75
	Cont.	1.05	3.18	2.07
ARC	Periodic	0.96	1.96	1.39
	Cont.	0.96	2.18	1.54

Table 6.5: Maximum speedups with dynamic parameter adaptation.

Soon, the hash table contains stale data, independently of other parameters, and this causes most lookups to fail.

As detailed in section 6.2.6, instead of trying to find good fixed parameter values, we introduced two ways of dynamically adapt them depending on the current network and traffic environment. We applied the first method to the length of the interval of normal operations in *periodic sampling* and the second method to the miss threshold of *continuous sampling*. Table 6.5 shows the maximum speedup attained by using the two methods on both the ARM and ARC architecture. Speedups from 1.05 to 3.18 can be reached on both platform and datasets (except for the original EEMBC dataset on ARC). However, as figure 6.8 shows, our results are even stronger when compared with the hypothetical minimum and maximum speedups attainable with fixed parameters presented in the first part of the result section. The graphs show a set of two bars: on the left are the ones that show speedup lower and upper bounds with fixed parameters and on the right the ones that show the speedup of the dynamic approach. Interestingly the dynamic approach ones are always very close to the hypothetical maximum. In fact if we had to choose fixed parameters we would obtain average speedups somewhere in the middle between maximum and minimum and substantially lower than the ones obtained with the dynamic approach.

6.4 Conclusions

In this chapter we have introduced adaptive software caching as a non-standard data transformation suitable for improving performance of packet processing network applications. We have shown how temporal adaptivity can be achieved, supporting network traffic scenarios that change over time and have evaluated the effectiveness of our

proposed data transformation against the industrial EEMBC networking benchmarks and real-world data sets and speedups of up to 3.30 and 2.27 have been achieved for two different platforms (ARC 750D and INTEL STRONGARM). Our results clearly demonstrate that adaptive data organisation schemes are necessary to ensure optimal performance under varying network loads.

Part II

Supporting Static Analyses

Chapter 7

Static Analysis

Introduction

The last chapters showed that for data plane network applications, data transformations at source or compilation level can dramatically improve performance. We demonstrated it in three successive steps: by introducing a new type of workload characterisation that highlights differences between application domains using decision trees and applying it to network applications, by devising three different data transformations that target the bottlenecks discovered with the characterisation and finally by making the data transformations adaptive to changes in network environment (traffic, type, size, location, time, etc.).

In this chapter we are taking a step back from dynamic adaptation to take a look at how make use of static analysis techniques to gather information about network applications without these being executed and profiled or dynamically adapted. Static analysis plays an important role in making the network specific data transformations we introduced effective. On one side it allows to discover upper bounds for the size

of data structure which can be effectively used to define the size of arrays in the *array regrouping* data transformation or the size of the hash table for *software caching*. This can also lead us to have more information on the running time of applications where data structure traversals are performed. On the other hand static analysis can reveal how many resources in terms of memory usage and execution time are needed for a certain procedure or for a whole program to finish. This information can be hugely advantageous when deciding whether to apply a specific optimisation or not and to establish which parameters achieve the best results.

Overview

This section presents a tool that performs static code analysis using a combination of amortised analysis and separation logic. Given a minimal amount of input information it is able to give precise resource utilisation constraints for various heap based data structures. This information can be used to dynamically adjust compiler optimisations directed to networking applications such as *software caching* and *array regrouping* as presented in chapter 5.

The tool we present relies on a combination of amortised complexity analysis and separation logic that has first been theoretically proved in [Atkey, 2010]. This technique allows resource analysis of imperative programs which manipulate heap-based data-structures such as trees and linked lists. Our tool applies this analysis to real code with a minimal amount of specifications obtained from programmer-supplied annotations. Although Java is probably not the language of choice for most data plane network application, we chose it for this analysis due to its numerous constraints and special constructs such as annotations that are useful for providing additional information to the code. However the analysis performed on it doesn't rely on any of its specific run-time characteristics and the concepts expressed with this method can be extended to other programming languages.

7.1 Specifying resource consumption

Consider the following code:

```
class IntList {
   int head;
   IntList tail;

   IntList concat (IntList p, IntList q) {
      if (p == null) return q;
      else {
         IntList t = p;
         while (t.next != null)
            t = t.next;
         t.next = q;
         return p;
      }

   ...
   }
}
```

This defines a simple class of linked lists with integer entries and a method which concatenates two lists `p` and `q`. If `p` is empty then `append` returns `q`, otherwise a pointer `t` traverses `p` until it reaches the final cell, then adjusts its `next` field to point to `q` and returns `p`.

Suppose that we want to describe the behaviour of `concat`. Let us introduce an assertion `lseg(a,b)` which states that there is a well-formed list segment in the heap for which `a` points to the first cell and `b` points to the final cell. Intuitively, in the heap we have a picture of the form shown in figure 7.1, with all cells distinct.

A first attempt at a specification for the `concat` method might be as follows:

```
@Requires (lseg(@arg p, null), lseg(@arg q, null))  // precondition
@Ensures (lseg (@ret, null))                        // postcondition
IntList concat (IntList p, IntList q) { ... }
```

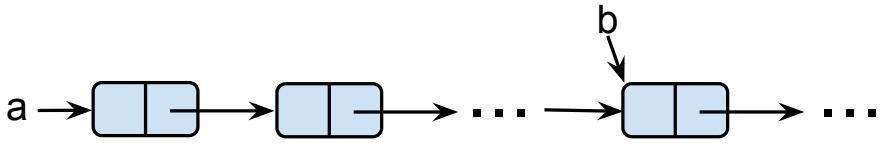


Figure 7.1: List segment `lseg(a,b)`.

The intended meaning of this specification is that *if* when we enter the method, the arguments `p` and `q` point to well-defined list segments, *then* when we reach the end of the method, the return value will also point to a well-defined list segment. The specification may be regarded as a contract with the user: if the inputs to the method satisfy the precondition, then the output is guaranteed to satisfy the postcondition. Ideally, we will be able to *prove* that the implementation of the method does actually guarantee this behaviour.

Unfortunately, there is a problem with the above specification. If `p` and `q` are pointers to the same location in memory (in other words, they are just different names for the same list), then we will end up with a circular structure: we will iterate along to the end of the list pointed to by `p` and then adjust the `next` pointer to point back to the head of the list: see Figure 7.2.

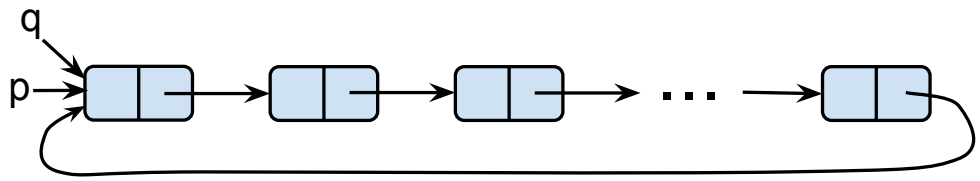


Figure 7.2: `concat` result in case of two pointers (`p` and `q`) pointing to the same list segment.

This violates the intended meaning of our `lseg` predicate, and hence the postcondition is false. One might attempt to deal with this by modifying the method to check whether `p==q`, but that would still not work since if the lists pointed to by `p` and `q` share any cells we will still end up with heap structures containing loops (Figure 7.3). Modifying the method to detect such situations would make it unnecessarily complicated; a better strategy is to amend the assertions to exclude problematic inputs from the outset. The key to this is to base the assertions on *Separation Logic* [Reynolds, 2002], a logic which is designed for arguing about non-overlapping structures and has proven very useful in the analysis of heap-allocated data structures in recent years.

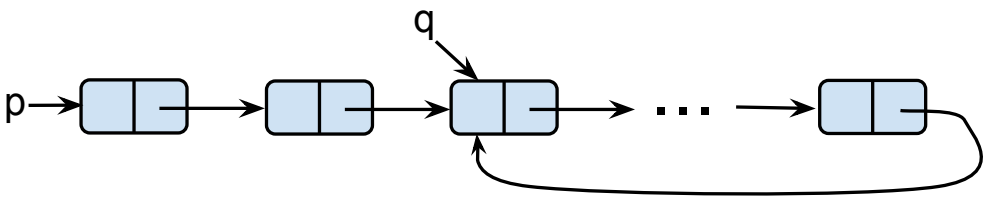


Figure 7.3: `concat` result in case of a pointer `q` pointing to an internal element of a list segment pointed by `p`.

Separation Logic has a number of novel logical connectives for arguing about non-overlapping objects. For example, in addition to the usual logical conjunction $\wedge$ ($A \wedge B$ means that A is true and B is true), Separation Logic has the *separating conjunction* *: $A * B$ is true if A is true and B is *separately* true. In the context of heap-allocated structures $A * B$ will be interpreted as meaning that A and B are both true, but on *disjoint regions of the heap*. See [Reynolds, 2002] for full details of Separation Logic.

If we use the separating conjunction to rewrite our original specification as

```
@Requires (lseg(@arg p, null) * lseg(@arg q, null))
@Ensures (lseg(@ret, null))
```

then it becomes valid: if the user supplies two well-formed lists which share no memory cells then the return value is also a well-formed list.

Amortised analysis with Separation Logic

The techniques of the previous section allow us to specify functional properties of methods which manipulate heap-allocated structures. However, our main interest is in the resource consumption of methods, where "resource" refers to some quantity which is consumed by the method: for example, we might consider the number of heap objects allocated by a method, the number of bytecode instructions executed, or the number of network packets sent. In this research, we will consider the problem of finding the number of times a special method called `consume` is executed, but this can easily be replaced by other methods or bytecode instructions in order to deal with other resources.

In [Atkey, 2010], it is shown that assertions in separation logic can be neatly extended to include information about resource consumption, and that it is possible both to verify annotations and to infer resource usage for methods where iteration is driven by the processing of heap allocated data structures.

This is based on the idea of *amortised analysis* [Tarjan, 1985] of algorithms involving data structures. The approach we will take here is to imagine that each node of a data structure is equipped with a number of tokens, and that one of these is consumed each time the node is processed. Consider our previous example, with some calls to `consume` added:

```
IntList concat (IntList p, IntList q) {
   consume();
   consume();

   if (p == null) return q;
   else {
      IntList t = p;
      while (t.next != null) {
         t = t.next;
         consume();
      }
      t.next = q;
      return p;
   }
```

We see that `consume` is called twice at the start of the method and then once for each node in the list `p`. We can express this by extending our Separation Logic specifications to include costs:

```
@Requires (lseg(1, @arg p, null) * lseg(0, @arg q, null), 2)
@Ensures (lseg (0, @ret, null), 0)
```

The extra numeric annotations are interpreted as saying that if we enter the method with one token for each node of `p` plus two extra tokens then the method can successfully execute and we are left with no tokens at the end; since each call to `consume` requires one token, we see that `consume` is called at most `length(p)+2` times.

We do not require that the annotations specify the minimal number of tokens required, only a number which is sufficient to allow the method to complete. For example

```
@Requires (lseg(7, @arg p, null) * lseg(2, @arg q, null), 5)
```

would also be a valid precondition: if we have the specified number of tokens then the method is still able to complete, but this time we will have some tokens left over which would enable further processing of the result at a later stage. The left-over tokens can be included in the postcondition. For example

```
@Requires (lseg(7, @arg p, null) * lseg(2, @arg q, null), 5)
@Ensures (lseg(2, @ret, null), 3)
```

is also a valid (albeit non-optimal) specification.

Atkey shows that it is possible to use a linear programming technique based on the ideas of Hofmann and Jost [Hofmann and Jost, 2003] to verify that resource annotations such as those above are valid; in fact, he shows that it is actually possible to *infer* minimal resource annotations and thus the resource consumption of a method. We will see examples of this later.

7.2 Amortised Analysis for Java bytecode

We have developed an analyser which implements the ideas of the previous chapter. Source programs are equipped with Java annotations giving preconditions and postconditions of the type described above; the current version also requires annotations giving loop invariants.

Our analyser works on compiled class files. The Java compiler stores source annotations in the class file, and the analyser retrieves these and uses them to perform the analysis on JVM bytecode.

Annotations for amortised analysis

We use Java annotations to equip methods with resource-usage specifications. In this section we will expand on the informal description given in earlier sections.

Java annotations are a specific form of metadata that can be added to Java source code. They can be associated with Java classes, fields, methods and method parameters, and are embedded in class files when compiling Java code. They are defined using class-like structures in files named after the annotation.

In order to be able to provide amortised analysis on methods we have defined three annotations. These allow user to specify method preconditions and postconditions together with loop invariants which are required by the analyser.

- `@Requires(assertions)` for preconditions
- `@Ensures(assertions)` for postconditions
- `@Invariant(assertions)` for loop invariants

All of them are given in the form of a Java `String` containing assertions. A shortened version of the assertion syntax in EBNF is shown in Figure 7.4.

Data assertions consist of comma-separated lists of assertions of the form `term == term` or `term != term` and are used to provide the analyser with information about when references point to the same or different Java objects. These are mostly required in loop invariants.

There are two types of *heap assertion*, which are Separation Logic predicates enriched with indications of field content. The first type of heap assertion indicates that part of the heap forms either a list segment or a tree (e.g. `lseg(r1, @arg x, null)`, `tree(r1, @arg x)`); both of these expand to more complex assertions built from Separation Logic primitives. The second type of heap assertion indicates to the analyser that a particular field points to a particular heap location or to some undetermined location; these are mostly required in preconditions and postconditions, where they facilitate interprocedural analysis by exposing information about heap structure for use in reasoning with Separation Logic.

Finally, *resource assertions* are linear expressions such as `3*r1 + 5*r2 + r3 + 7`

```
assertions = [exvars] assertion ('||' assertion)* ;
assertion = '{' data-assertions '|'
                heap-assertions '|'
                resource-assertion '}' ;

exvars = 'exists' typedvar + '.' ;
typedvar = id ':' type ;
     (* id is a Java identifier: [A-Za-z_][A-Za-z0-9_]* *)
type = 'int' | 'long' | 'float' | 'double' | 'ref' ;

data-assertions = data-assertion (',' data-assertion)* ;
data-assertion  =
     term '==' term
   | term '!=' term ;

heap-assertions = heap-assertion (',' heap-assertion)* ;
heap-assertion =
     '[' field '->' location ']'
   | 'lseg (' resource-assertion ',' term ',' term ')'
   | 'tree (' resource-assertion ',' term ')' ;


term =
     id             (* existential variable *)
   | '@arg' id (* only @arg variables allowed in preconditions *)
   | '@var' id
   | 'null'
   | '@ret' ;  (* @ret only allowed in postcondition *)

field = '(' term ')' '.' id ':' type ;

location = var | '?' ;

resource-assertion = linear-expression ;
```

Figure 7.4: Assertion EBNF

which specify constants and variables which we want the analyser to use to infer the resources associated with the method before and after execution (i.e. in the precondition and postcondition); they are also used in list and tree predicates to indicate resources associated with each node of the structure.

We have also introduced a dummy method called `consume` which does nothing except tell the static analyser that at the point where it is introduced, a unit of resource is being used. We can imagine such a dummy method being hidden inside library code in the future, stating the amount of resource used by each method defined in each class, so that programmers will not need to add it explicitly but rather implicitly use it by invoking library methods. In the present implementation though, libraries have not been modified and developers have to specify resource consumption explicitly in their code.

Invariant localisation in Java code

Loop invariants have to be given for each loop for the amortised analysis to be effective but the Java language does not allow the inclusion of annotations inside the code. This is problematic if the method being analysed contains two or more loops, since we need a way to decide which invariant refers to which loop. We could simply give invariants in the same order as the loops appear in the code, but it is possible that a compiler might produce bytecode in which the order of the loops in the bytecode does not correspond to the order in the source code. Our way to obviate this limitation was to identify each loop with an integer identifier. Each loop invariant is given a identifier (`@Invariant(id, assertion)`) and the same identifier has to match the argument of a dummy method `Loop.invariant(id)` placed just before the loop in the code. Thus by searching the code, the analyser can associate the declared invariants with the corresponding loop.

7.3 Examples and output interpretation

We illustrate the operation of the analyser by returning to our earlier list concatenation example.

```
import uk.ac.ed.inf.resa.*;

public class IntList {

  private int data;
  private IntList next;
  ...

  @Requires("{ | lseg(1, @arg p, null) * lseg (0, @arg q, null)| 2 }")
  @Ensures("{ | lseg(0, @ret, null) | 0 }")
  @Invariant("{ @var t != null | lseg (0, @var p, @var t) *"
              + "lseg (1, @var t, null) * lseg (0, @var q, null) | 0}")
  public static IntList concat (IntList p, IntList q) {
    Amortised.consume();
    Amortised.consume();
    if (p==null) return q;

    IntList t = p;
    Loop.invariant (0);
    while (t.next != null) {
      t = t.next;
      Amortised.consume();
    }
    t.next = q;

    return p;
  }
  ...
}
```

We have supplied a loop invariant which describes the state of resource usage as the program progresses. Whenever we reach the head of the `while` loop, we have used up the resources associated with the part of the first input list `p` which has already been processed (between `p` and `t`), we still have one unit of resource available in the remaining part of `p` (between `t` and `null`), and we do not require any resources to process `q`. Note the `@arg` and `@var` annotations attached to variable names. These are used to distinguish between the current value of a variable (`@var`) and the value of a method argument at entry to the method (`@arg`); they are not strictly necessary in this example, but are required in more complex examples where variables representing method arguments are modified. The `@ret` annotation refers to the method return value.

If we compile `IntList.java` and invoke the analyser then the output is as follows.

```
$ resa -amortised examples/IntList concat
...
Solved VCs
Verification successful
```

This shows that the analyser has succeeded in proving that the precondition and post-condition are satisfied. However, if we amend the precondition to say

```
@Requires("{ | lseg(1, @arg p, null) * lseg (0, @arg q, null)| 1 }")
```

then the verification fails because there is only one "constant" unit of resource available, and two are required by the calls to `consume` at the start of the method:

```
LP is infeasible
```

More interestingly, we can supply *generic* annotations and the analyser will infer suitable values for them.

```
@Requires("{ | lseg(x1, @arg p, null) * lseg (x2, @arg q, null)| x3 }")
@Ensures("{ | lseg(y1, @ret, null) | y2 }")
@Invariant("{ @var t != null | lseg (z1, @var p, @var t) *"
            + "lseg (z2, @var t, null) * lseg (z3, @var q, null) | z4}")
```

With these annotations the analyser outputs

```
Optimal solution for resource variables:
x1 = 1, x2 = 0, x3 = 2
y1 = 0, y2 = 0
z1 = 0, z2 = 1, z3 = 0, z4 = 0
```

We can also specify the amount of resource which we require when the method returns: if we replace the postcondition with `@Ensures("{ | lseg(3, @ret, null) | 7 }")` then we get `x1 = 4, x2 = 3, x3 = 9`, so that if the postcondition is to be satisfied then we must have 4 units of resource for each element of `p`, 3 for each element of `q`, and 9 extra units before the method is called.

In addition to this example we have been able to successfully analyse a number of other standard operations on lists, such as reversal, iteration, and deleting and inserting elements, together with similar operations for trees.

7.4 Implementation details

Our analyser is implemented in OCaml, and Java class files are represented by a collection of datatypes. For program analysis, the most important part of this is the representation of method bytecode. Our design is intended to be fairly general-purpose since we intend to support multiple analysis techniques, including the amortised analysis described earlier.

Java classfiles are initially converted into a low-level OCaml representation which is a fairly faithful representation of the structure of the class as described in the JVM specification [Lindholm and Yellin, 1999]. This is then converted into a higher-level representation which is more suited to analysis. We will give an outline of this representation here, but space limitations preclude a detailed description.

Bytecode instructions are decompiled into a form where the JVM stack has been eliminated. We keep track of which constants and local variables have been loaded onto the stack, and these are represented by a datatype called `value`, which has constructors for

constants of type `int`, `long`, `float`, `double`, `String` and `class`, together with variables. Variables are represented by integer identifiers which are tagged with the type of the corresponding value: this is one of `I`, `L`, `F`, `D`, or `A` (representing integers, long integers, floats, doubles, and addresses). We do not have special types for `boolean`, `byte`, `char` and `short` since the JVM treats all of these as 32-bit integers for most purposes. We also tag all references with the single type `A`, and make no attempt to keep track of the most specific class or interface to which the variable belongs; it would be possible to infer this information, but so far we have not required this.

There are two kinds of variables: *local* variables and *stack* variables. The former represent values loaded onto the stack from JVM local variables by means of instructions such as `iload`, and the latter represent intermediate values which have been created on the stack by arithmetic operations, method calls and so on. Each variable is marked with an integer which for local variables represents the number of the associated JVM local variable, and for stack variables is simply a counter which is incremented every time a new value is created on the stack and decremented when that value is consumed. Some care is required here since stack operations can duplicate values on the stack. The decompilation process handles this by creating new copies of variables using a special pseudo-instruction called `Copy`. Another complication is that one can load a local variable *r* onto the stack and then modify the contents of *r* before the earlier value (still on the stack) has been consumed; again, the `Copy` operation is used to avoid confusion by creating a new variable which represents the earlier value.

Instructions which act on the stack are represented by a datatype of operations which take values as arguments. This has 19 constructors which suffice to represent all of the JVM operations (`putfield`, `getfield`, `invokevirtual` and so on) except for those which involve control-flow transfer. Basic blocks are represented by a list of pairs consisting of operations together with the local variable or stack location where the result (if any) of the operation is to be stored. At the end of a basic block we have a member of a `continuation` datatype: this represents various types of jump, comparison, switch, and return operation, together with information specifying which blocks control can flow to after leaving the current block. We do not provide any representation for the `jsr` and `ret` instructions used by exception handlers, since these complicate analysis and are supposedly deprecated in current Java releases (although

we have encountered them in a few of the standard API classes in `rt.jar`). If one of these instructions is encountered during decompilation then an exception is thrown and the class is rejected.

The bytecode for an entire method is represented by an array of blocks, stored in preorder. This can be regarded as a graph, and we have a module which computes useful information such as successors, predecessors, and dominators, and can also provide other views of the graph, such as postorder and reverse postorder, which can be useful in some analyses.

Proof search

The most important part of the amortised analysis is the proof search procedure which is used to verify that the precondition of a method implies its postcondition, and also to check or infer the associated resource annotations.

This uses the method described by Atkey in [Atkey, 2010], and indeed our implementation uses an adapted version of code from a prototype implementation by Atkey, which analysed a textual form of a subset of Java virtual machine (JVM) bytecode. The basic idea is to visit the code in postorder, working backwards from the postcondition to infer weakest preconditions for each instruction, and then to prove that the weakest precondition for the first instruction is implied by the (user-supplied) precondition for the method.

Visiting the nodes of the control flow graph (CFG) in postorder ensures that when we visit a given node n, the preconditions for all of its successors are already known, and can thus be combined (by logical conjunction) to obtain a postcondition for n. However, this strategy fails when we encounter a back-edge $s \to t$ (i.e., when t is at the head of a loop). In this case we will not have visited t when we arrive at s, and it is for this reason that we require annotations for loop invariants: the annotation will provide a precondition for t, and will therefore be available to contribute to the postcondition for t.

The proof search procedure also collects linear constraints describing the resource usage of the bytecode instructions, as described in [Atkey, 2010]; once the analysis has

been completed and the basic Separation Logic annotations have been checked to make sure that the precondition implies the postcondition, the resource constraints are converted into a linear programming problem which is then solved using the Parma Polyhedra Library [Bagnara et al., 2008].

7.5 Example Usage

To prove the usefulness of our static analysis tool, we applied it to the transformation techniques developed in the first part of this thesis to help improving the data layout of network applications. In particular we applied it to discover when it is worth using software caching in the *route lookup* software described in section 5.3.4. To implement software caching we used a hash table with separate chaining as shown in figure 6.5 and we want to compare how an access to the hash table compares to one to the routing trees in terms of number of cycles needed.To determine the worst-case scenario where introducing software caching will surely be beneficial for performance, we assume that the full routing tree fits completely in the cache memory, which can be accessed in a single clock cycle. On the other hand we assume that none of the hash table entries is in cache and set the number of cycles to access main memory to 20 cycles, which is roughly the amount of cycles needed by STRONGARM 1100, one of the embedded architectures we have taken into consideration while empirically evaluating data transformations. Figure 7.5 illustrates this theoretical worst-case scenario. On a real platform the access to the hash table entries would be faster after the first fetch as the entry will be put into cache. We nevertheless decided not to take this fact into account since we are aiming to determine the absolute worst-case scenario. We want to determine the maximum decision tree size from which the use of a software caching strategy will surely incur benefits.

We linked the static analyser with the software caching data transformation by rewriting the route lookup algorithm and software caching code in Java and instructing it by introducing resource usage calls into the code (`Amortised.consume()`). Our goal was to roughly introduce a resource consumption for each clock cycle of the STRONGARM 1100. Therefore we introduced one consume call for each arithmetic instruction. On the other hand, we added one consume call for each load operations on the routing tree,

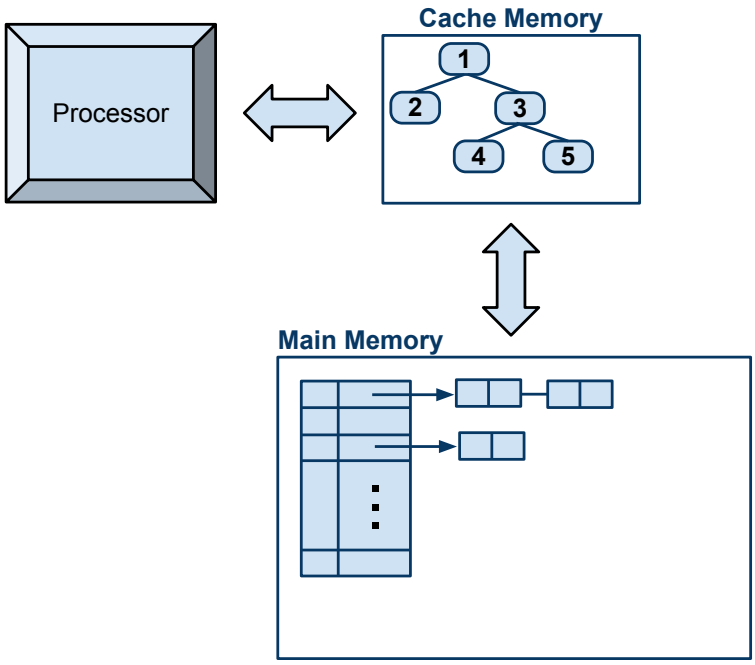


Figure 7.5: Theoretical worst-case memory utilisation scenario for software caching using a hash table.

thus simulating the fact that it can be found in cache and added 20 consume calls for load operations on the hash table to imitate the retrieval from main memory.

After instructing the code we ran the static analyser on it, which gave us resource boundaries for lookups in both structures, routing tree and hash table. In both cases we came up with a constant (per lookup) resource component and a component that is dependent on the amount of nodes traversed in the tree and in the chained list in the hash table respectively. These are summarised in table 7.1.

We assume that, if we keep the hash table big enough (twice the size of the number of elements stored or more), we can limit the number of linked elements to one in the vast majority of cases. In this case the number of resources needed per lookup in the hash table will be 60. For the routing tree case instead, the total number of

	Constant	Per Node
Tree	9	11
Hash Table	10	50

Table 7.1: Comparison of the resources needed for 1 lookup in the routing tree and hash table (constant and per node component).

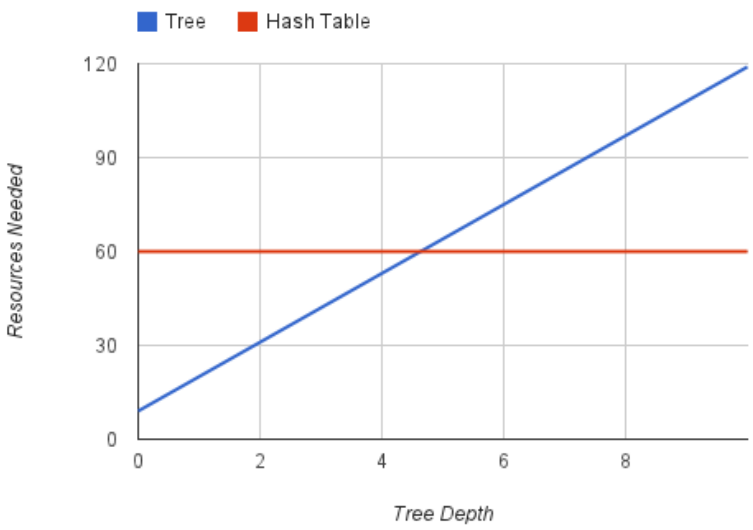


Figure 7.6: Comparison of resource utilisation over routing tree depth between routing tree and hash table. If the tree is deeper than 4.6, the usage of a hash table is surely beneficial.

resources needed is always proportional to the number of nodes traversed, which in turn corresponds to the depth of the tree. We try to illustrate and compare the two solutions in figure 7.6. The blue line corresponds to the number of resources needed for 1 lookup for increasing tree depth. The red line corresponds to the number of resources needed for 1 lookup in the hash table. The lookup in the hash table requires more resources if the tree depth is larger than 4.6, at which point the blue line crosses the red one, i.e. the lookup in the tree requires more resources than in the hash table.

The whole procedure to determine this tree depth can be made statically by analysing the code with our static analyser. Although our worst-case resource usage estimation has been performed with some approximation, this proves that the analyser can be used effectively to predict worst-case resource usage of real applications.

7.6 Conclusions

In this chapter we have shown that using a technique that combines amortised analysis with separation logic provides effective resource constraints for real world source code

from single procedures up to entire applications. This information can be used to effectively apply network software data transformations that require knowledge of upper bounds of how much memory or how much time is needed such as *array regrouping* and *software caching*.

In the next chapter we are going to look at another group of network software, i.e. *control plane* network applications. Contrary to the *data plane* applications approached in the previous chapters, *control plane* don't require strict deadlines to be met to process network data. They are supposed to be run on control plane processors of network processors or on regular desktop machines and include software to manage networks, to supply servicing and statistics, to perform data structure management etc. Their behaviour is in most aspects closer to generic management applications than data plane networking applications and don't benefit from the optimisations introduced in previous chapters. Nevertheless in the next part of this thesis we introduce two examples to highlight and explain the difference between the two networking application sets. The first application, STIX, is a broadband wireless network management system whereas the second, BSENSE, is an application that enables broadband census.

Chapter 8

Summary and Conclusions

In this research we focused on data plane network applications, analysed their behaviour and presented ways to improve their performance by means of software and compiler alterations. In more detail, these are the aspects we looked at and the solutions we devised in order to tackle the problems we encountered:

1. Analysis of application workloads to bring to light the bottlenecks specific to each application domain.
   In chapter 4 we presented a methodology for an automated workload characterisation which may lead to the identification of tuning opportunities for domain-specific compilers. Our proposed method uses *decision trees* combined with *clustering* to produce human-interpretable results. Therefore, it can be used by compiler developers seeking to identify domain-specific performance bottlenecks and guidance on how to address these. We have demonstrated how decision trees can be used to detect the specific characteristics of five embedded application domains.

2. Design of novel compiler optimisations to target network applications that take into account the issues discovered in point 1.
   In chapter 5 we demonstrated how the decision tree characterisation can be exploited to identify a missing domain-specific compiler optimisation. We showed that data cache misses form a performance bottleneck for network infrastructure applications running on RISC based NPUs. So, based on these results, in chap-

ter 5 we evaluated three non-standard data transformations (*structure splitting*, *array regrouping* and *software caching*) that aim to improve overall cache efficiency for typical networking applications. For the EEMBC networking benchmarks, *real-world* data sets and two platforms we demonstrated that speedups of up to 2.62 can be achieved. At the same time we showed that changing network traffic patterns demand an adaptive approach to performing *online data transformation*.

3. Adaptation of the compiler optimisations found in point 2 to changing network conditions in order to always get the maximum possible performance improvement in all environments.
   The three data transformations introduced to target point 2 of the goals work well for some network conditions but their efficiency is not constant across all of them. To achieve maximum efficiency we introduced adaptivity to *software caching* as a non-standard data transformation suitable for improving the performance of packet processing network applications with all network conditions. We showed how temporal adaptivity can be achieved, supporting network traffic scenarios that change over time. We evaluated the effectiveness of our proposed data transformation against the industrial EEMBC networking benchmarks and real-world data sets achieving speedups of up to 3.30 and 2.27 for two different platforms (ARC 750D and INTEL STRONGARM) and thus demonstrating that adaptive data organisation schemes are necessary to ensure optimal performance under varying network loads.

4. Exploration of static code analysis tools to allow the effective implementation of the compiler optimisations introduced in point 2.
   In part II we showed an implementation of a static analyser using a combination of amortised analysis and separation logic. The static analyser takes source code augmented with specific annotations and indications of unit resource usage as input and is able to work out the resource utilisation of whole programs.

For control plane applications instead we aimed to show how these differ from data plane ones. To achieve this we showed the details of the implementation of two different control plane applications—STIX, a distributed wireless broadband management system, and BSENSE, a distributed system for broadband census. We demonstrated

that, even though the agent software of both systems are considered network applications, they differ substantially from the data plane applications introduced before, especially because they don’t have any strict timing constraints.

Acronyms

PCA principal component analysis

TLB translation lookaside buffer

IPC instruction per cycle

BPU branch prediction unit

***k*-NN** *k*-nearest neighbour

EEMBC Embedded Microprocessor Benchmark Consortium

MTU maximum transfer unit

LSU load store unit

AS autonomous system

CAIDA The Cooperative Association for Internet Data Analysis

UoE University of Edinburgh

WiMo Wireless & Mobile Networking Group

ICSA Institute for Computing Systems Architecture

EBNF extended Backus-Naur form

CFG control-flow graph

NPU network processing unit

SNR signal to noise ratio

JVM Java virtual machine

ISP internet service provider

CDN content delivery network

IDT inter departure time

NP network processor

RISC reduced instruction set computer

ICMP internet control message protocol

CFG control flow graph

TLB translation lookaside buffer

I-TLB instruction-TLB

Bibliography

[Abid et al., 2007] Abid, F., Izeboudjen, N., Sahli, L., Lazib, D., Titri, S., Louiz, F., and Bakiri, M. (2007). Opencores based embedded system on chip for network applications. *Proceedings of the International Conference on Circuits, Systems, Signals*.

[Agakov et al., 2006] Agakov, F., Bonilla, E., J.Cavazos, B.Franke, Fursin, G., O'Boyle, M., Thomson, J., Toussaint, M., and Williams, C. (2006). Using machine learning to focus iterative optimization. In *Proceedings of the 2006 International Symposium on Code Generation and Optimization (CGO)*.

[Aggarwal, 2002] Aggarwal, A. (2002). Software caching vs. prefetching. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 157–162, New York, NY, USA. ACM.

[Albert et al., 2007] Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. (2007). Cost Analysis of Java Bytecode. In Nicola, R. D., editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag.

[Albert et al., 2008] Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. (2008). COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer.

[Alfred et al., 1986] Alfred, V., Sethi, R., and Jeffrey, D. (1986). *Compilers: Principles, Techniques and Tools*. Addison-wesley.

[Atkey, 2010] Atkey, R. (2010). Amortised resource analysis with separation logic. In *ESOP*, pages 85–103.

[Bagnara et al., 2008] Bagnara, R., Hill, P. M., and Zaffanella, E. (2008). The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21.

[Beringer et al., 2003] Beringer, L., MacKenzie, K., and Stark, I. (2003). Grail: a functional form for imperative mobile code. In *Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop*, number 85.1 in Electronic Notes in Theoretical Computer Science. Elsevier.

[Bernardi et al., 2010] Bernardi, G., Calder, M., Fenacci, D., Macmillan, A., and Marina, M. (2010). Stix: a goal-oriented distributed management system for large-scale broadband wireless access networks. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, pages 245–256. ACM.

[Bernardi and Marina, 2010] Bernardi, G. and Marina, M. (2010). BSense: a system for enabling automated broadband census: short paper. In *Proceedings of the 4th ACM Workshop on Networked Systems for Developing Regions*, pages 1–2. ACM.

[Bessa Maia and Holanda Filho, 2009] Bessa Maia, J. and Holanda Filho, R. (2009). One-against-all methodology for features selection and classification of internet applications. *IP Operations and Management*, pages 27–38.

[Bishop, 1995] Bishop, C. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press, USA.

[CAIDA, 2011] CAIDA (2011). CAIDA: The cooperative association for internet data analysis. `http://www.caida.org`.

[Calder et al., 1997] Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., and Zorn, B. (1997). Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222.

[Caruana and Niculescu-Mizil, 2006] Caruana, R. and Niculescu-Mizil, A. (2006). An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168. ACM.

[Cascón et al., 2011] Cascón, P., Ortiz, A., Ortega, J., Díaz, A. F., and Rojas, I. (2011). Accelerating network applications by distributed interfaces on heterogeneous multiprocessor architectures. *The Journal of Supercomputing*, pages 1–12.

[Cavazos et al., 2007] Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M. F. P., and Temam, O. (2007). Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization (CGO)*, pages 185–197, Washington, DC, USA. IEEE Computer Society.

[Chang and Kuo, 2009] Chang, Y.-K. and Kuo, F.-C. (2009). Packet processing with blocking for bursty traffic on multi-thread network processor. In *HPSR'09: Proceedings of the 15th international conference on High Performance Switching and Routing*, pages 160–165, Piscataway, NJ, USA. IEEE Press.

[Chen et al., 1993] Chen, W., Bringmann, R., Mahlke, S., Anik, S., Kiyohara, T., Warter, N., Lavery, D., Hwu, W., Hank, R., and Gyllenhaal, J. (1993). Using profile information to assist advanced compiler optimization and scheduling. *Languages and Compilers for Parallel Computing*, pages 31–48.

[Chin et al., 2005] Chin, W.-N., Nguyen, H. H., Qin, S., and Rinard, M. (2005). Memory usage verification for oo programs. In *In SAS 05*, pages 70–86. Springer.

[Chiueh and Pradhan, 2000] Chiueh, T.-c. and Pradhan, P. (2000). Cache memory design for internet processors. *IEEE Micro*, 20(1):28–33.

[Chung et al., 1999] Chung, E., Benini, L., and De Micheli, G. (1999). Dynamic power management using adaptive learning tree. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 274–279.

[Cooper et al., 2005] Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S., Subramanian, D., Torczon, L., and Waterman, T. (2005). ACME: adaptive compilation made

efficient. *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 40(7):69–77.

[Cooper and Waterman, 2003] Cooper, K. D. and Waterman, T. (2003). Investigating adaptive compilation using the MIPSPro compiler. In *In Proc. of the Symp. of the Los Alamos Computer Science Institute*.

[Dai et al., 2005] Dai, J., Huang, B., Li, L., and Harrison, L. (2005). Automatically partitioning packet processing applications for pipelined architectures. *SIGPLAN Not.*, 40(6):237–248.

[Degermark et al., 1997] Degermark, M., Brodnik, A., Carlsson, S., and Pink, S. (1997). Small forwarding tables for fast routing lookups. *ACM SIGCOMM Computer Communication Review*, 27(4):3–14.

[Ding and Liu, 2005] Ding, T. and Liu, N. (2005). A parallelizing compiler approach based on ixa. In Yang, L. T., Zhou, X., Zhao, W., Wu, Z., Zhu, Y., and Lin, M., editors, *ICESS*, volume 3820 of *Lecture Notes in Computer Science*, pages 720–725. Springer.

[Ding and Newman, 2000] Ding, Y. and Newman, K. (2000). Automatic workload characterization. In *Proceedings of CMG '00*.

[Downey and Feitelson, 1999] Downey, A. B. and Feitelson, D. G. (1999). The elusive goal of workload characterization. *ACM SIGMETRICS Performance Evaluation Review*, 26(4):14–29.

[Edler and Hill, 1999] Edler, J. and Hill, M. D. (1999). Dinero IV trace-driven uniprocessor cache simulator. `http://pages.cs.wisc.edu/~markhill/DineroIV`.

[Elakkumanan et al., 2005] Elakkumanan, P., Liu, L., Kumar Vankadara, V., and Sridhar, R. (2005). CHIDDAM: A data mining based technique for overcoming the memory bottleneck problem in commercial applications. In *Proceedings of the 48th IEEE International Midwest Symposium on Circuits and Systems*, pages 1888–1891.

[Erman et al., 2006] Erman, J., Arlitt, M., and Mahanti, A. (2006). Traffic classification using clustering algorithms. In *Proceedings of the 2006 SIGCOMM workshop on Mining network data*, pages 281–286. ACM.

[Ghiya and Hendren, 1996] Ghiya, R. and Hendren, L. J. (1996). Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA. ACM.

[Gopalan and Chiueh, 2002] Gopalan, K. and Chiueh, T.-c. (2002). Improving route lookup performance using network processor cache. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–10, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Gulwani et al., 2009] Gulwani, S., Mehra, K. K., and Chilimbi, T. M. (2009). SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139.

[Gulwani and Zuleger, 2010] Gulwani, S. and Zuleger, F. (2010). The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 292–304, New York, NY, USA. ACM.

[Gupta et al., 2002] Gupta, R., Mehofer, E., and Zhang, Y. (2002). Profile guided compiler optimizations. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, pages 143–174.

[Hagog and Tice, 2005] Hagog, M. and Tice, C. (2005). Cache aware data layout reorganization optimization in GCC. In *Proceedings of the GCC Summit*.

[Hofmann and Jost, 2003] Hofmann, M. and Jost, S. (2003). Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197.

[Hoste and Eeckhout, 2006] Hoste, K. and Eeckhout, L. (2006). Comparing benchmarks using key microarchitecture-independent characteristics. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 83–92. IEEE.

[Hoste and Eeckhout, 2007] Hoste, K. and Eeckhout, L. (2007). Microarchitecture-independent workload characterization. *The 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 27(3):63–72.

[Hoste and Eeckhout, 2008] Hoste, K. and Eeckhout, L. (2008). Characterizing the unique and diverse behaviors in existing and emerging general-purpose and domain-

specific benchmark suites. *2008 IEEE International Symposium on Performance Analysis of Systems and software (ISPASS)*, pages 157–168.

[Hoste and Eekhout, 2008] Hoste, K. and Eekhout, L. (2008). COLE: Compiler optimization level exploration. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, New York, NY, USA. ACM.

[Hoste et al., 2006] Hoste, K., Phansalkar, A., Eeckhout, L., Georges, A., John, L. K., and Bosschere, K. D. (2006). Performance prediction based on inherent program similarity. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques (PACT)*, pages 114–122, New York, NY, USA. ACM.

[Hubert et al., 2010] Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T. P., Monfort, V., Pichardie, D., and Turpin, T. (2010). Sawja: Static analysis workshop for Java. *CoRR*, abs/1007.3353.

[Hubicka, 2005] Hubicka, J. (2005). Profile driven optimisations in gcc. In *GCC Developers Summit*, page 107.

[Huggahalli et al., 2005] Huggahalli, R., Iyer, R., and Tetrick, S. (2005). Direct cache access for high bandwidth network I/O. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 50–59, Washington, DC, USA. IEEE Computer Society.

[Hundt et al., 2006] Hundt, R., Mannarswamy, S., and Chakrabarti, D. (2006). Practical structure layout optimization and advice. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 233–244, Washington, DC, USA. IEEE Computer Society.

[Hunter and Hwu, 2002] Hunter, H. C. and Hwu, W.-M. W. (2002). Code coverage and input variability: effects on architecture and compiler research. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 79–87, New York, NY, USA. ACM.

[Hyun et al., 2009] Hyun, Y., Huffaker, B., Andersen, D., Aben, E., Shannon, C., Luckie, M., and kc claffy (2009). The CAIDA IPv4 routed/24 topology

dataset - 13/09/2009. `http://www.caida.org/data/active/ipv4_routed_24_topology_dataset.xml`.

[Joshi et al., 2006] Joshi, A., Phansalkar, A., Eeckhout, L., and John, L. (2006). Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782.

[Jung and Pedram, 2010] Jung, H. and Pedram, M. (2010). Supervised learning based power management for multicore processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(9):1395–1408.

[Kim et al., 2002] Kim, J., Jung, S., Paek, Y., and Uh, G.-R. (2002). Experience with a retargetable compiler for a commercial network processor. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 178–187, New York, NY, USA. ACM.

[Kind et al., 2002] Kind, A., Pletka, R., and Stiller, B. (2002). The potential of just-in-time compilation in active networks based on network processors. In *Proceedings of IEEE OpenArch*, volume 2.

[Kulkarni et al., 2003] Kulkarni, C., Gries, M., Sauer, C., and Keutzer, K. (2003). Programming challenges in network processor deployment. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 178–187, New York, NY, USA. ACM.

[Lattner and Adve, 2003] Lattner, C. and Adve, V. (2003). Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign.

[League et al., 2001] League, C., Trifonov, V., and Shao, Z. (2001). Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[Lee and Chen, 2007] Lee, Y.-H. and Chen, C. (2007). An efficient code generation algorithm for non-orthogonal DSP architecture. *Journal of VLSI Signal Processing Systems*, 47(3):281–296.

[Li et al., 2008] Li, H. et al. (2008). *Workload characterization, modeling, and prediction in grid Computing*. LIACS, Computer Systems Group, Faculty of Science, Leiden University.

[Li et al., 2002] Li, J., Boyer, F., and Aboulhamid, E. (2002). Retargetable c compiler for network processors. *Proceedings of 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002), Orlando, Fl.*

[Lin et al., 2008] Lin, J. M., Chen, Y., Li, W., Tang, Z., and Jaleel, A. (2008). Memory characterization of SPEC CPU2006 benchmark suite. In *Proceedings of the 2008 Workshop for Computer Architecture Evaluation of Commerical Workloads (CAECW), co-located with HPCA '08*.

[Lindholm and Yellin, 1999] Lindholm, T. and Yellin, F. (1999). *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[Liu et al., 2006] Liu, D., Hua, B., Hu, X., and Tang, X. (2006). High-performance packet classification algorithm for many-core and multithreaded network processor. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 334–344. ACM New York, NY, USA.

[Liu et al., 2008] Liu, Z., Yu, J., Wang, X., Liu, B., and Bhuyan, L. (2008). Revisiting the cache effect on multicore multithreaded network processors. In *DSD '08: Proceedings of the 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 317–324, Washington, DC, USA. IEEE Computer Society.

[Lloyd, 1982] Lloyd, S. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137.

[Luk et al., 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, New York, NY, USA. ACM.

[Mierswa et al., 2006] Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., and Euler, T. (2006). YALE: Rapid prototyping for complex data mining tasks. In Ungar, L., Craven, M., Gunopulos, D., and Eliassi-Rad, T., editors, *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 935–940, New York, NY, USA. ACM.

[Monsifrot et al., 2002] Monsifrot, A., Bodin, F., and Quiniou, R. (2002). A machine learning approach to automatic production of compiler heuristics. In *Artificial Intelligence: Methodology, Systems, Applications*, pages 389–409. Springer Verlag.

[Muchnick, 1997] Muchnick, S. (1997). *Advanced compiler design and implementation*. Morgan Kaufmann.

[Mudigonda et al., 2008] Mudigonda, J., Vin, H., and Yavatkar, R. (2008). A case for data caching in network processors. Technical report, University of Texas at Austin.

[Nie et al., 2005] Nie, X., Wilson, D., Cornet, J., Damm, D., and Zhao, Y. (2005). Ip address lookup using a dynamic hash function. In *Electrical and Computer Engineering, 2005. Canadian Conference on*, pages 1642–1647. IEEE.

[Pentakalos et al., 1996] Pentakalos, O. I., Menascz, D. A., and Yesha, Y. (1996). Automated clustering-based workload characterization. In *In Proceedings of the 5th NASA Goddard Mass Storage Systems and Technologies Conference*.

[Phansalkar et al., 2005] Phansalkar, A., Joshi, A., Eeckhout, L., and John, L. (2005). Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 10–20.

[Poovey, 2007] Poovey, J. (2007). Characterization of the EEMBC benchmark suite. `http://www.eembc.org`.

[Portillo et al., 2003] Portillo, A. J. R., Hammond, K., Loidl, H.-W., and Vasconcelos, P. (2003). Cost analysis using automatic size and time inference. In *Proceedings of the 14th international conference on Implementation of functional languages*, IFL'02, pages 232–247, Berlin, Heidelberg. Springer-Verlag.

[Quinlan, 1993] Quinlan, J. (1993). *C4.5: programs for machine learning*. Morgan Kaufmann.

[Raatikainen, 1993] Raatikainen, K. E. E. (1993). Cluster analysis and workload classification. *SIGMETRICS Perform. Eval. Rev.*, 20(4):24–30.

[Ramakrishna and Jamadagni, 2003] Ramakrishna, S. and Jamadagni, H. (2003). Analytical bounds on the threads in ixp1200 network processor. *Digital Systems Design, Euromicro Symposium on*, 0:426.

[Reynolds, 2002] Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*.

[Shah, 2001] Shah, N. (2001). Understanding network processors. *Master's thesis, University of California, Berkeley*.

[Sipkova, 2003] Sipkova, V. (2003). Efficient variable allocation to dual memory banks of DSPs. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2003)*.

[Skinner and Kramer, 2005] Skinner, D. and Kramer, W. (2005). Understanding the causes of performance variability in HPC workloads. *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, pages 137–149.

[Spoto, 2005] Spoto, F. (2005). JULIA: A generic static analyser for the Java bytecode. Proceedings of FTfjP'2005, Glasgow.

[Stephenson et al., 2003] Stephenson, M., Amarasinghe, S., and Martin, M. (2003). Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90. ACM Press.

[Swanson and Gilder, 2008] Swanson, B. and Gilder, G. (2008). Estimating the exaflood: The impact of video and rich media on the internet??a zetabyte?of data by 2015? *Discovery Institute Report*.

[Tarjan, 1985] Tarjan, R. E. (1985). Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318.

[The Embedded Microprocessor Benchmark Consortium, 2008] The Embedded Microprocessor Benchmark Consortium (2008). EEMBC benchmarks. `http://www.eembc.org`.

[Thomson, 2008] Thomson, J. (2008). *Using Machine Learning to Automate Compiler Optimisation*. PhD thesis, School of Informatics, University of Edinburgh.

[Uh et al., 1999] Uh, G.-R., Wang, Y., Whalley, D., Jinturkar, S., Burns, C., and Cao, V. (1999). Effective exploitation of a zero overhead loop buffer. *Proceedings of the 1999 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 34(7):10–19.

[University of St Andrew/Heriot-Watt University,] University of St Andrew/Heriot-Watt University. The hume programming language.

[Vallée-Rai et al., 1999] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. (1999). Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135.

[von Luxburg and David, 2005] von Luxburg, U. and David, B. S. (2005). Towards a statistical theory of clustering. In *PASCAL Workshop on Statistics and Optimization of Clustering*.

[Wagner and Leupers, 2001] Wagner, J. and Leupers, R. (2001). C compiler design for an industrial network processor. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 155–164, New York, NY, USA. ACM.

[Wagner and Leupers, 2002] Wagner, J. and Leupers, R. (2002). Advanced code generation for network processors with bit packet addressing. In *Proceedgins of the Workshop on Network Processors (at HPCA8)*.

[Walsworth et al., 2009] Walsworth, C., Aben, E., kc claffy, and Andersen, D. (2009). The CAIDA anonymized 2009 internet traces - 31/03/2009. `http://www.caida.org/data/passive/passive_2009_dataset.xml`.

[Wang and Kaeli, 2003] Wang, Y. and Kaeli, D. (2003). Source level transformations to improve i/o data partitioning. In *Proceedings of the international workshop on Storage network architecture and parallel I/Os*, page 35. ACM.

[Wang and O'Boyle, 2008] Wang, Z. and O'Boyle, M. F. (2008). Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 75–84, New York, NY, USA. ACM.

[Xu et al., 2009] Xu, B., Chang, J., Huang, S., Xue, Y., and Li, J. (2009). Efficiency of cache mechanism for network processors. *Tsinghua Science & Technology*, 14(5):575 – 585.

[Yan, 2004] Yan, K. (2004). Characterization and classification of modern microprocessor benchmarks. Master's thesis, New Mexico State University.

[Yu et al., 1992] Yu, P. S., syan Chen, M., Heiss, H.-U., and Lee, S. (1992). On workload characterization of relational database environments. *IEEE Transactions on Software Engineering*, 18:347–355.

[Zhong et al., 2004] Zhong, Y., Orlovich, M., Shen, X., and Ding, C. (2004). Array regrouping and structure splitting using whole-program reference affinity. *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 39(6):255–266.

[Zhuang, 2006] Zhuang, X. (2006). *Compiler optimizations for multithreaded multicore network processors*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA. Adviser-Pande, Santosh.

[Zhuang and Pande, 2004] Zhuang, X. and Pande, S. (2004). Balancing register allocation across threads for a multithreaded network processor. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 289–300. ACM New York, NY, USA.