How I Did It

Paul-Jules Micolet

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2018

Abstract

This doctoral thesis will present the results of my work into the reanimation of lifeless human tissues.

Acknowledgements

Many thanks to my mummy for the numerous packed lunches; and of course to Igor, my faithful lab assistant.

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.

(*Paul-Jules Micolet*)

Table of Contents

Chapter 1

Introduction

Chapter 2

Background

This chapter covers the different topics that are present in this thesis. The background starts by briefly covering Chip Multicore Processors and Heterogeneous Chip Multicore Processors to motivate the existence of Dynamic Multicore Processors. Then the core-fusion technique, which is the main mechanism brought forward by Dynamic Multicore Processors is described in detail. This is followed by a description of the EDGE instruction set architecture which is used in the Dynamic Multicore Processor described in this thesis. Finally, streaming programming languages, which are used in Chapter ref, are explained.

2.1 Chip Multicore Processors

Chip Multicore Processors (CMPs) have become ubiquitous due to the difficulty in scaling single core performance. In a CMP, multiple processor cores are put on a single package as can be seen in Figure 2.1. The most common CMP uses homogeneous cores as they reduce the design complexity both from a hardware and software perspective []. Unlike single core systems, the performance improvement in CMPs come from running multiple tasks in parallel. These tasks can either be different programs or multiple threads from the same program running on the multiple cores. By defining speedup S to be the original execution time of the program over the new execution time with n processors and f representing the fraction of the program which can be parallelised; Amdahl's Law states

$$S = \frac{1}{(1-f)+\frac{f}{n}} \tag{2.1}$$

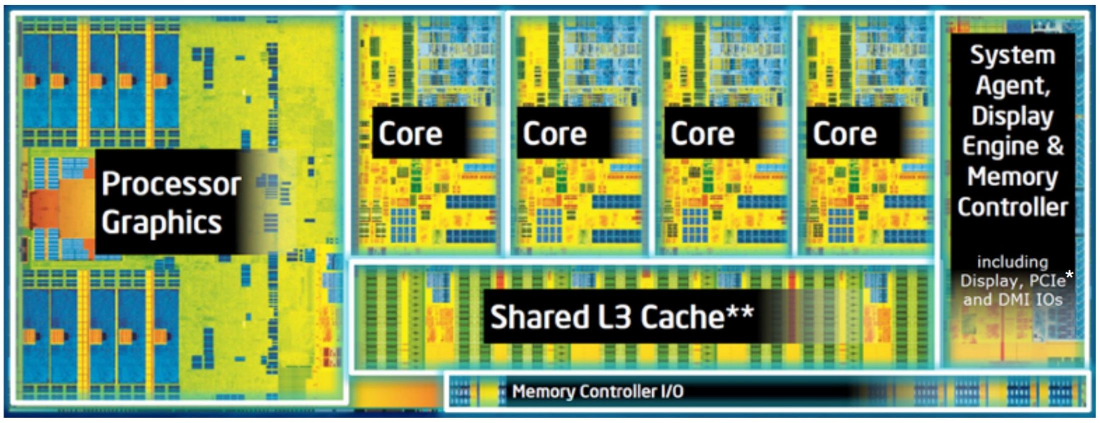
Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe* and DMI IOs
Shared L3 Cache**
Memory Controller I/O

Figure 2.1: Intel Core i7 processor internal die photograph taken from intel whitepaper

thus, given an infinite number of processor cores [?]

$$\lim_{n \to \infty} S = \frac{1}{(1-f)} \tag{2.2}$$

This second equation demonstrates how, given any program, the speedup obtained by using a CMP will be limited to the fraction f of parallel code found in the program itself. As all the processor cores are homogeneous this will cause serial bottlenecks to severely reduce the potential speedup as no core is adapted to speedup such regions. This implication has pushed research into finding ways of parallelising code to its fullest [], however this may not always be possible []. Thus whilst CMPs have become a mainstain in processor design, the homogeneous model has its limits.

2.2 Heterogeneous Chip Multicore Processors

Unlike CMPs, Heterogeneous Chip Multicore Processors (HCMPs) or Asymmetrical Chip Multicore Processors (ACMPs) bring a variety of cores onto a single package. This may come in different forms, such as having multiple instruction set architectures on the same system on chip (MPSoCs) [45, 44], or same ISA different size cores on an SoC [?]. For example, Figure 2.2 shows a schemata for ARM's big.LITTLE HCMP, where a high-performance Cortex-A15 is paired with a simpler, power efficient Cortex-A7. The two cores are connected via a cache coherent interconnect which provides data coherence at the bus-level, allowing the cores to make reads to its neighbor. Software is then run on one of the cores depending on a profile; if the user requires performance over energy, then the Cortex-A15 will be chosen, however if energy/power efficiency is required then the Cortex-A7 will be chosen.

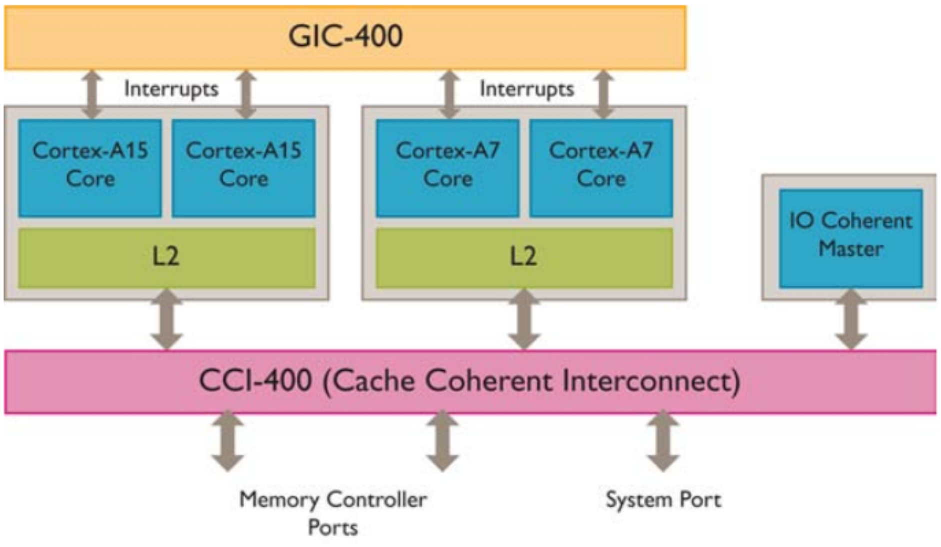

Figure 2.2: Example of a heterogeneous multicore processor proposed by ARM (big.LITTLE)

This small example already demonstrates an advantage of HCMPs; unlike CMPs, the variety of cores on an HCMP provide a flexibility to the hardware. This can be used for different purposes, such as security [44], energy/power savings [45] and speeding up applications [45]. In their 2014 paper, Venkat et al. [**?**] demonstrate that a multi-ISA HCMP can improve performance by up to 1.4x and achieve energy savings of up to 40% compared to a CMP on a peak-power budget of 40W. They motivate the idea that HCMPs with heterogeneous ISAs even improve over the performance of single-ISA HCMPs with speedups around 15% and energy savings of 21.5%.

However, whilst the hardware diversity in HCMPs is an advantage compared to CMPs, it also increases programming complexity. For example, Gupta et al. in [20] show that a single-ISA octa-core big.LITTLE architecture can have 20 CPU cores, combined with the ability to dynamically modify the voltage, this leads to 4000 different configurations. This highly increases the complexity of obtaining the correct settings for different programs. MPSoCs also face a similar issue as having more than a single ISA not only adds design challenges, but program migration between different cores may in fact deteriorate performance [9].

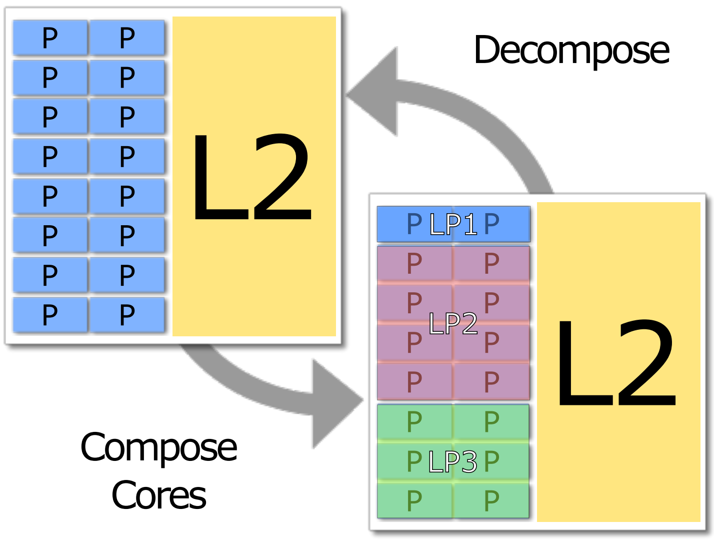

Figure 2.3: High-level view of a dynamic multicore processor that can modify its core count.

2.3 Dynamic Multicore Processors

In both CMPs and HCMPs, once the chip is fabricated the design cannot be modified, meaning that many of the trade-offs between power, performance and area cannot be changed after production. Dynamic Multicore Processors (DMPs) attempt to bridge the gap between the two previous designs by allowing the execution substrate to adapt dynamically at runtime. Mitall's survey [29] defines three types of modifiable resources: the core count [22], number of resources that each core has [21] and microarchitectural features [14, 2, 39].

2.3.1 Core Fusion Dynamic Multicore Processors

A DMP that modifies core count is composed of homogeneous cores with a reconfigurable fabric. Physical cores can function either on their own or as a group of physical cores; this is called a Logical Core (LC). Throughout this thesis, the term core-fusion will be used to define the mechanism of cores creating an LC. A logical core will fetch instructions from a single source and execute them accross all the physical cores that compose the LC. Cores can fuse dynamically and create a logical core of any sizes. For example in Figure 2.3, the DMP fuses cores into 3 LCs of sizes 1, 8 and 6 physical cores. The exact mechanism of core-fusion are described later on in Section **??**.

The advantage of a core-fusion DMP over the traditional CMP or HCMP is the ability to reconfigure the processor dynamically to better match the tasks at hand. For example, large sequential sections of code with high Instruction Level Parallelism (ILP) can be accelerated on a logical core that mimics a wide superscalar processor. On parallel workloads the DMP can be reconfigured by de-composing the logical cores as seen in Figure 2.3 to match the Thread Level Parallelism (TLP).

2.3.2 Resource Sharing Dynamic Multicore Processors

A more fine-grained reconfiguration can be found in resource-sharing DMPs. There exist different models for resource sharing DMPs. For example the WiDGET DMP by Watanabe et al. [51], cores are built out of Instruction Engine front-ends which function similarly to Out of Order (OoO) cores' front and back ends. They then are connected to Execution Units which they can choose to use. Each core in the WiDGET DMP also have access to their neighbors Execution Units, allowing for more variation. Another example of resource sharing can be found in Rodrigues et al.'s work [35] where a core can use resources such as Arithmetic Logic Units (ALUs) from other cores.

2.3.3 Microarchitectural Reconfigurable Dynamic Multicore Processors

A final example is a DMP which can reconfigure microarchitectural features to better fit the current application. Fallin et al. [?] observe that serial code can exhibit phases that fit different microarchitectural features. According to them, these phases may only been in the ten to hundred thousands instructions long. These DMPs can therefore modify microarchitecural features, such as in-order or out-of-order execution, to best match the current phase of a program.

2.4 EDGE Instruction Set Architecture

The Explicit Data Graph Execution [6] (EDGE) instruction set architecture (ISA) is a data-flow based ISA. Figure 2.4 shows a high-level overview of how EDGE differs from a traditional instruction set architecture. The EDGE compiler has a first pass which generates instructions from the original source code. Blocks are then generated from the basic-blocks found in the code generation pass.

Unlike traditional ISAs, blocks do not communicate via registers, but rather the

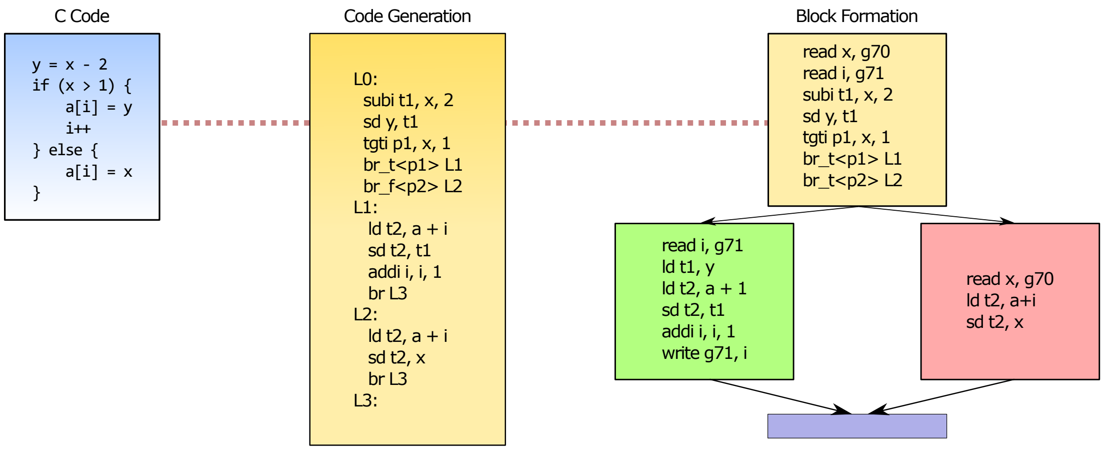

Figure 2.4: High-level view of the EDGE ISA flow.

output targets of instructions are encoded to instruction inputs [38]. Loads and stores in each EDGE block are assigned unique identifiers which are used resolve load-store dependencies. Thus, the EDGE ISAs encode dependencies between instructions at the ISA level, registers are only used for inter-communication between blocks. An EDGE block also contains a header that will inform the hardware about the number of stores and register writes contained in the block [**?**], this is used to facilitate committing blocks.

EDGE blocks also have a set of restrictions to satisfy correctness. If a block does not meet these requirements, it may need to be broken down into smaller blocks. These restrictions are:

- Block Size: an EDGE block may be between 4 to 128 instructions.
- Load/Store: an EDGE block may have at most 32 load/store instructions.
- Entry/Exit: an EDGE block may have a single exit but may have multiple exits.

To increase the average size of EDGE blocks, multiple blocks can be combined together to form one large block called a hyperblock. This is achieved through the use of instruction predication. For example given an if/else statement, the compiler can generate a single block, predicating all instructions of the else statement. As the compiler needs to declare the number of stores and register writes in the block header, extra instructions may need to be generated to ensure the block always executes the same amount of stores.

Overall, the EDGE ISA enables the architecture to dispatch blocks speculatively, with low overhead [34, 25], therefore, increasing exploitation of ILP.

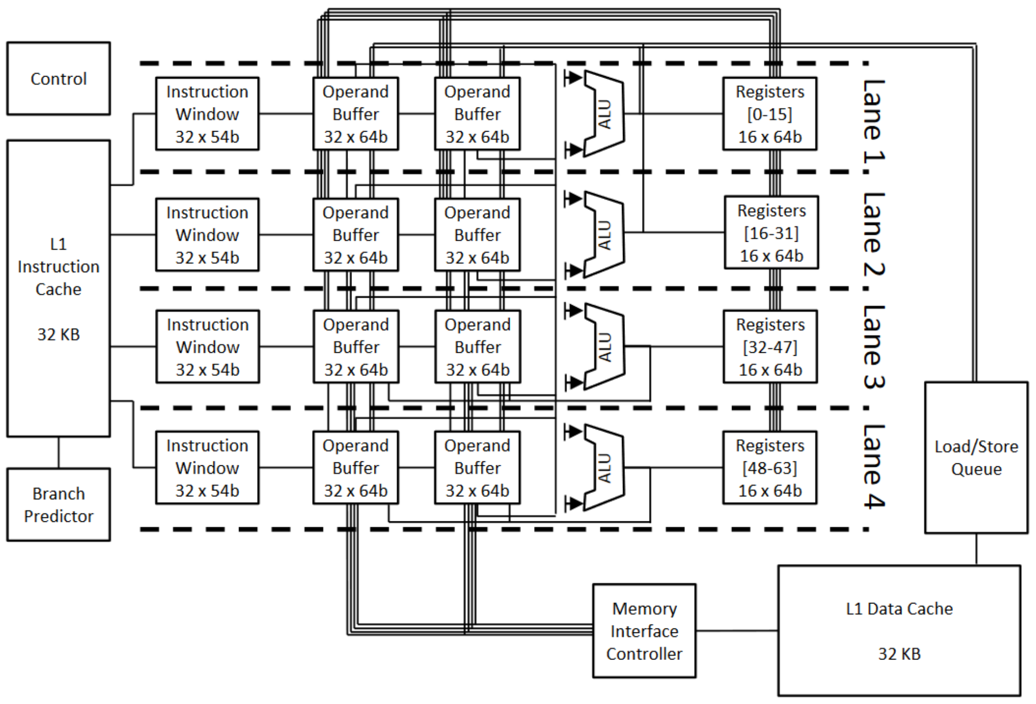
Control
L1 Instruction Cache 32 KB
Branch Predictor
Instruction Window 32 x 54b
Operand Buffer 32 x 64b
ALU
Registers [0-15] 16 x 64b
Registers [16-31] 16 x 64b
Registers [32-47] 16 x 64b
Registers [48-63] 16 x 64b
Lane 1
Lane 2
Lane 3
Lane 4
Load/Store Queue
Memory Interface Controller
L1 Data Cache 32 KB

Figure 2.5: Example of a four lane core on an EDGE processor taken from [?].

2.5 EDGE Processor

2.5.1 Core Lanes

EDGE instruction blocks can be up to 128 instructions long, however this often isn't the case. To maximize core-utilisation, each core on an EDGE Processor is segmented into a set number of lanes which can each fetch and decode their own blocks. A lane is able to fetch a block of maximum size

$$\frac{128}{NumberOfLanes} \tag{2.3}$$

For example, a four lane core as seen in Figure 2.5 can have up to four blocks of 32 instructions. Fetching blocks larger than 32 instructions will fill up more than one lane. Lanes allow EDGE cores to be more flexible to block size variability.

2.5.2 Core Fusion

Core Fusion is achieved by fusing a set of *physical* cores to create larger *logical* cores. This does not modify the physical structure of the chip, instead it provides a unified

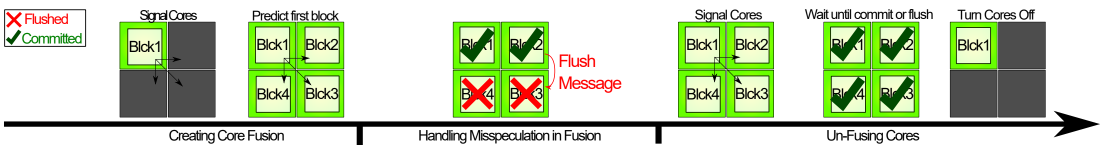

Figure 2.6: Core Fusion Mechanisms for our EDGE-based architecture.

view of a group of physical cores to the software. In the processor used throughout the thesis, the micro-architecture is distributed: register files, Load Store Queues (LSQs), L1 caches and ALUs all look like nodes on a network. This means that when cores fuse together, this is similar to adding an extra node to the network. Fusion is a dynamic modification and may occur during the execution of a program to better fit the workload. Unlike traditional CMPs, fused cores will operate on the same thread and attempt to extract Instruction Level Parallelism (ILP) rather than Thread Level Parallelism (TLP) [28, 32].

Figure 2.6 shows the different stages and mechanisms of core fusion for a four core system. When creating a new core fusion a master core informs all other cores about the fusion and sends the predicted next block address to the next available fused core. When we start a new thread on a fused core the OS and runtime write the new core mapping to a system register. The hardware then flushes these cores if they are not idle and sets the PC of the first block of that thread on one core in the logical processor and starts executing. When a core mispredicts a branch in a fusion, it informs the other cores which flush any younger blocks. When un-fusing, the master core informs the other cores, which then commit or flush their blocks and power down while the master core continues to fetch and execute blocks from the thread. The extra hardware required to support dynamic reconfiguration is very minimal [25] since most of the machinery already in place can be reused such as the cache coherence protocol when fusing and un-fusing the cores.

When a logical core fetches multiple blocks, it may execute them out of order. However memory instructions and instructions that modify registers pass through the LSQ and register-file and are executed in order. This ensures that blocks operate on memory in a consistent fashion. In case of a memory violation caused by undetected dependencies a flush of all blocks younger than the violator, including the violating block, is performed.

2.6 Streaming Programming Languages

Streaming programming languages are a branch of dataflow programming that focus on applications that deal with a constant stream of data. These applications, such as audio or video decoding can be commonly found in mobile devices. Unlike conventional programming languages such as C++, these languages abstract the concept of incoming and outgoing data to permit the programmer to focus on how the data should be treated. Programs are described as directed graphs where nodes are functions and their edges represent their input and output streams. These languages offer primitives to describe such a graph [42] which expose parallelizable and serial sections of the application directly to the compiler. Rates of incoming and outcoming data can also be defined to facilitate load balancing optimizations [8].

Features of streaming programming languages make them an ideal language for targeting multicore processors. The explicit data communication between the different tasks in the program, the ability to estimate the amount of work performed in each task and information about data rates between tasks allows the compiler to easily generate a multi-threaded application that can run on a dynamic multicore processor. However, the main challenge consists of deciding how to map the different tasks onto threads and how to allocate the right amount of resources to maximize performance.

2.7 Machine-learning guided optimisations

Chapter 3

Related Work

Chapter 4

Streamit

4.1 Introduction

Multicore processors are now common in all computing systems ranging from mobile devices to data centers. As advances in single threaded performance have slowed, multicore processors have offered a way to use the increasing numbers of transistors available. However, designing processors that scale to a large number of cores is difficult and a shift towards tiled architecture seems inevitable. A tiled architecture such as Tilera [3] or Raw [49] is composed of smaller simpler cores that are placed on a regular grid. This improves hardware scalability and enables multi-threaded applications to exploit the large core count.

However, workloads that require high single threaded performance are penalized by the simple nature of each core [13]. One solution to this problem is heterogeneous multicores which utilize cores with different levels of power and performance. Although heterogeneous multicores are common place in mobile devices, they have little reconfiguration or adaptive capabilities (e. g. only two type of cores available for ARM big.LITTLE). Dynamic multicore processors offer a solution to this problem by allowing cores to compose (or fuse) together [22] into larger logical cores to accelerate single threads. This produces "on-demand" heterogeneity where cores are grouped to adapt to the workload's demand.

While dynamic multicore processors sound like a promising approach, they come with their own challenges, particularly on the software side [52]. In most parallel programming models such as OpenMP, the user is directly responsible for mapping parallelism to the hardware; a difficult and time consuming task. This problem is further exacerbated when hardware resources can be combined since programmers have to take into account the dynamic behavior of the architecture [4].

To solve this problem, we first argue that there is a need to raise the programming abstraction and remove the burden of mapping parallelism from programmers. Dataflow programming models such as StreamIt [42] and Lime [1] offer one part of the solution. Applications are expressed as dataflow graphs and — ideally — the compiler or runtime determines the mapping of parallelism onto the available hardware and controls the grouping of hardware resources. However, optimally mapping parallelism and managing hardware resources remains an open problem given the sheer complexity of the resulting design space.

In this paper, we first conduct an analysis of the design space and show the impact of modifying resources and thread mapping. We conduct this analysis using a set of StreamIt programs and run them on a verified cycle-level simulator for a tiled reconfigurable architecture with support for core composition. We develop a machine learning model using the information gathered from our exploration. This model predicts the best number of threads for a given application and an optimal number of cores to allocate to each thread.

To demonstrate the viability of our approach we compare the results of the predictive model to the best sampled thread and core composition pairing in a space of more than 32,000 design points. The model matches, and even outperforms in some cases, the performance of the best sampled points in the space, with speedups of up to 9x on a 16 core processor compared to single threaded execution on a single core.

The main contributions of this paper are:

- An analysis of the co-design space of thread partitioning and core composition;
- A study on the impact of a simple loop transformation on the optimal core composition;
- A machine-learning model to determine the optimal core composition and thread partitioning;
- An analysis of the most important static code features used by the model.

The rest of the paper is structured as follow. Section **??** presents information on dynamic multicore processors and dataflow programming models. Section 4.2 motivates this work by showing the complexity of the design space. Section 5.4 describes our methodology and section 4.4 presents an in-depth analysis of the design space.

Section 4.5 develops a machine-learning model to predict the best thread mapping and core composition while Section 4.6 shows the performance achieved by our model. Related work is discussed in Section 5.9 and Section 4.8 concludes this paper.

4.2 Motivation

This section illustrates the difficulty of finding a good partition and resource allocation. A simple experiment is conducted where we take one StreamIt benchmark, *Beamformer*, and partition its tasks into threads and allocate various number of cores to each thread. A co-design of more than 32,000 combinations (exhaustive space) of thread mappings and core compositions is generated. Each design point is executed on a dynamic multicore simulator (exact details about the experimental setup are presented later in section 5.4).

Figure 4.1 presents the distribution of the execution times from the co-design space as a violin plot. For the unfamiliar reader, an intuitive way to think about this violin plot is to consider it as a smoothed histogram rotated by 90 degrees and mirrored. We observed that the majority of the sampled points have a cycle count around 525,000 with the worst points taking more than 2 millions cycles. The best performance is around 275,000 cycles which is about 2x faster than the majority of the data points. This shows that finding the right combination of thread mapping and core composition is critical since a wrong choice often leads to suboptimal performance.

This example illustrates the necessity for designing the technique to predict both the optimal number of threads and core composition to use. The next section will present a more in-depth analysis of the design space before presenting our machine-learning predictive model.

4.3 Methodology

In this section we present our design exploration of a set of streaming applications being executed on a DMP. We describe how changing the thread mapping and core composition affect the benchmarks and what we can learn from this. In addition, we look at the impact of loop unrolling and how it helps exploit larger fused cores.

4.3.1 Overview

Figure 4.2 presents the workflow of our system. First, we use the source-to-source StreamIt compiler to unroll loops as this is usually beneficial when cores are composed as we will see later. Then, we extract static code features such as the program's graph structure. These features are used as an input to our first machine-learning model to determine the Thread Level Parallelism (TLP). This information is used to partition the program into threads and the StreamIt compiler produces a C++ program which is then compiled using our C++ compiler.

Then, a second machine-learning model is used which uses static code features extracted from the SteamIt code. This model is used to decide on the core topology. This is achieved by finding the amount of Instruction Level Parallelism (ILP) in each thread and by determining how many physical cores should be fused for that thread. Finally, we reconfigure the processor to fuse the requested resources and execute the partitioned program.

4.3.2 Dynamic Multicore Processor

We use a Dynamic Multicore Processor for our research based on an Explicit Data Graph Execution (EDGE) Instruction Set Architecture that resembles [18]. This differs from other DMPs such as CoreFusion, WidGET and Shared Architecture [22, 51, 53] which utilize a CISC/RISC instruction set. To evaluate our work we use a customizable cycle-level simulator verified within 4% of RTL. The simulator is highly configurable, allowing us to model a variety of parameters such as the number of cores, details of the memory hierarchy and synchronisation schemes. For our experiments we use a 16 core dual issue configuration with 16 KB private L1 caches and a 2 MB shared L2.

4.3.3 StreamIt Benchmarks

StreamIt is a high-level synchronous dataflow streaming programming language that defines programs as directed graphs. StreamIt offers an elegant way of describing streaming applications, abstracting away how infinite data streams are managed to allow the programmer to solely focus on how the data must be treated. A StreamIt program is composed of functions - called *Filters* - which operate on streams of data. Filters can be connected via *Pipelines*, *SplitJoins* or *Feedback Loops*.

Pipelines represent a sequence of connecting filters operating on the same stream,

Parameter	Values
# of cores in the processor	16
# threads per application	1 – 15
# cores per thread	1 – 15
# sampled core compositions	100
# our sampled space	1316
# total sample space	32762

Table 4.1: Design space considered per application.

each filter operating on the output of the previous filter. In a SplitJoin, data in the stream is passed through a split filter and either duplicated and passed on in parallel to the filters or distributed amongst the filters in a round-robin manner. The output of all the filters in a SplitJoin are then concatenated in a round-robin fashion through a joiner filter. Finally a Feedback Loop provides a way for filters to operate on their outputs. The resulting program written in StreamIt represents a graph where the nodes are filters and their edges represent the incoming and outgoing data streams.

In this paper, we use 15 StreamIt benchmark all taken from the official StreamIt repository. For each benchmark we used the default input provided in the repository and the default iteration count of 10.

4.3.4 Design Space

The parameters and size of the space are given in table 4.1. In this study we use 16 cores and assign core 0 to the main thread and for runtime management. This leaves 15 cores available for each application. We restrict each core to running only a single thread (no preemptive scheduling) which leads to a possible number of threads between 1 and 15. Cores can be fused together to form a logical core with up to 15 physical cores, making the total number of cores assigned to a thread between 1 and 15. This leads to a total space size of 32,767 unique combination per benchmark.

4.3.5 Sample Space

Given a partitioning, any benchmark that is split into 15 threads requires 32,767 executions to cover the entire space. Running an exhaustive exploration of the space requires approximately a week of simulation on a 572+ node supercomputer. For this reason, we decided to sample 1,316 random points from the entire space. This roughly corresponds to 100 core compositions for each number of threads (the actual number, 1,316 is smaller than 1,500 since for low thread counts there are less than 100 possible different core composition). *InsertionSort* is the only exception since it can at most only be split into 5 threads leading to 415 sample points.

To gain confidence that the best configuration from the sample space is indeed close to the real best in the entire space, we used a statistical model based on the Stopping Criterion defined in [48]. This model estimates, given a sample of the total space, if the best observed performance of that sample space is within a percentage of the statistical best performance. Our results demonstrate that the sample space selected is representative of the whole space.

Figure 4.3 shows, for each of the benchmarks, the proximity to the statistical best when increasing the sub-sample space given a maximal uncertainty of 5% (i. e. minimum 95% confidence). As can be seen by the plain line, the model shows that the best sample point is actually within 5% (0.05 proximity) of the best for all benchmark. To further prove that the statistical model based on the Stopping Criterion is indeed accurate, we conducted an exhaustive exploration for five benchmarks. The dotted line in figure 4.3 shows the actual proximity to the best for *Audiobeam*, *Beamformer*, *BitonicSort*, *CFAR* and *FMRadio*. As can be seen after 1316 samples, the performance we achieve is actually very similar to the one predicted by the statistical model, hence confirming prior work [48]. To summarize, we can conclude that the best point found in our sample space of 1,316 points is at least within 5% of the real best in the exhaustive space with 95% confidence.

4.4 Design Space Exploration

We now conduct an exploration of the software/hardware co-design space. The software side includes partitioning the program, determining the number of threads and the loop unrolling compiler optimization. The hardware side is about finding out the best core composition that maximizes performance for a given partitioning.

4.4.1 Thread Partitioning

We start by analyzing the impact of thread partitioning on performance. Thread partitioning is about deciding how many threads to create and how to partition StreamIt filters into these threads. To simplify this study, we use the default streaming partitioner to decide on how to allocate filters to cores which is based on simulated annealing. On the hardware side, we consider two scenarios: the "without composition scenario" where there is exactly one core per thread and the "with composition scenario" where each thread receives between 1 and 15 cores.

Figure 4.4 shows how performance varies under both scenarios as a function of the number of threads. We observe that regardless of how cores are composed all curves follow the same trend. The optimal number of threads using core composition is very similar to the scenario without composition. This important observation means that we can estimate the optimal number of threads for a benchmark independently of the hardware composition. Our system can therefore proceed in two stages: first determine the optimal number of threads and then decide on a core composition.

Figure 4.4 also shows that the performance of most benchmarks starts deteriorating passed a certain number of threads making it critical to not over-allocate threads. This motivates our use of machine learning to decide the optimal number of threads to use. Finally we also observe that executions without compositions always perform worse. This demonstrates that composing cores is essential to obtain the best performance from a workload.

4.4.2 Core Composition

Using core composition, the processor fuses a number of cores and associates them to a thread to increase single threaded performance. Whilst this flexibility is advantageous, choosing the right amount of cores for a given thread is difficult due to the large number of possible configurations [19].

Figure 4.5 shows how threading and composition affects performance for the *Audiobeam* benchmark. The curves represent the density distribution for different core compositions as a function of the number of threads. The right hand side Y-axis represents the number of threads present in the current version of the benchmark normalized by the total number of points in the design space. For each of the threaded versions we ran the benchmark using on average 100 different compositions. The density curve for thread 15 is a single point as there exists only a single composition.

The variance of each of the curves represents the influence of composition on the benchmark's performance for a given number of threads. For this benchmark the impact of core composition is actually very large for the best performing number of threads (1–5). Interestingly, as more threads are used, performance shifts worsens, echoing the results shown in the previous section.

4.4.3 Impact of Loop Unrolling

In this section we study the impact of one compiler optimization by focusing on loop unrolling. Filters containing large amounts of loops potentially contain high degrees of instruction level and memory level parallelism. Unrolling may increase the degree of parallelism which is advantageous to a wider fused processor. Loop unrolling may also yield similar results to vectorization when vectorization may not easily be applied or available.

Figure 4.6 presents an example of how loop unrolling affects performance on the *FMRadio* benchmark. The graph presents the same information as Figure 4.5 but with different executions of the benchmark when optimizing for speed and unroll factors 4, 16, and 64. Figure 4.6 shows that unrolling loops for *FMRadio* can greatly improve performance.

Another observation is that the best execution times for each of the threaded versions when unrolling does not follow the same trend previously described. The leftmost curve performance peaks at two threads whereas the rightmost peaks at five. As the number of cores fused can now be greater we encounter a resource problem when increasing the number of threads.

This example demonstrates that whilst the optimal number of threads is independent of the number of cores there still exists trade-offs between the two. This signifies that the amount of resources available to each thread must be taken into consideration before generating the program to balance the trade off between ILP and TLP.

4.4.4 Co-Design Space Best Results

This section presents the results of the entire co-design space exploration. Figure 4.7 characterizes how much of a performance increase is obtainable using a baseline of executing the benchmark on a single thread and single core without unrolling. For each benchmark, the *THREAD* bar represents the maximal speedup obtained by dividing the program into threads without fusing cores. The *CORE* bar represents the best speedup when we execute the benchmark in a single thread and fuse cores. *BOTH* represents the best speedup obtained for each benchmark using a combination of *THREAD* and *CORE*. Finally, for each benchmark, we obtained these results for both an unrolled and not unrolled to compare how unrolling affects performance. Figure 4.7 shows that when loops are not unrolled, composing cores will not greatly improve performance.

When studying the geometric mean we see that, without unrolling, finding the correct number of threads gives a speedup of 1.92 compared to 1.33 when using only core composition. This changes when taking unrolling into account as the core compositions can be used more efficiently. In this case, the speedup obtained from only composing cores is 13% worse than using only threads. The unrolling demonstrates that the StreamIt programs must be modified to take advantage of the core composition. Finally, it is important to note that whilst finding the optimal thread mapping is better than the best composition, the best performance is always obtained through a combination of both optimizations.

4.4.5 Summary

This section demonstrated that each parameter has a large effect on the performance of the workload. We have seen that regardless of using core composition or not, there exists for each benchmark an optimal number of threads. Unrolling is effective at exposing more opportunities for composition due to increased ILP but there is a balance to strike between extracting ILP and TLP. Figure 4.7 shows there is a 3x benefit (overall) by automating the partitioning of both the software (threads) and hardware (cores).

4.5 Machine Learning Models

As seen in the previous section, selecting the right number of threads and a good combination of cores is difficult. This difficulty arises from trying to balance between exploiting larger composed cores with block speculation and ILP and between exploiting a larger number of logical cores via TLP.

The problem can be decomposed into two stages; first, determining the right number of threads and then selecting a good core composition. In this section, we present two machine-learning models that predict the best thread partitioning and core composition to maximize performance.

4.5.1 Predicting the Best Number of Threads

4.5.1.0.1 Synthetic Benchmark Generation One of the difficulties of building a machine learning based model for StreamIt is the lack of benchmarks available [50]. Whilst there exists at least 30 realistic applications for StreamIt [40] this is simply not enough to create a large enough data set. To overcome this problem we generate synthetic StreamIt benchmarks and gather statistics from them in a similar style as in [50]. To ensure that the synthetic benchmarks are representative of realistic benchmarks we created them using filters from a set of micro-kernels found in some StreamIt examples. We have 30 different possible filters with different incoming and outgoing rates, different inputs and outputs. We also ensured that the total number of filters and split joins found in a synthetic benchmark are within the average of the realistic benchmarks.

For each generated application, 15 different threaded versions are generated. Each of these versions is ran using a single core per thread and the cycle count is recorded. We repeated this for 1000 unique randomly generated applications and record the best number of threads each time.

4.5.1.0.2 Extracting Features Once the benchmarks have been generated, the next step consists of gathering features for each applications. In order to build our two machine learning models we used an initial set of over 50 features extracted from StreamIt programs. These features were extracted using pre-existing tools within StreamIt and some extra counters added by us. The features selected for our models were determine through correlation analysis. In this section, when discussing correlation we specifically look at which variables correlate with the optimal number of threads. These features are used by the model to make a prediction about the number of threads to use.

Figure 4.8 shows the 10 variables that correlate the most with the optimal thread number. In StreamIt the term multiplicity references the number of times a filter will have to execute in a time slice when the graph is in a steady state [16]. In Figure 4.8 the highest correlating value, Number of Distinct Multiplicities, determines all different multiplicities found in the StreamIt graph. Unconditionally executed blocks represent sets of operations in a filter that will always execute.

There are very little variables that highly correlate beyond Number of Distinct Multiplicities. A high number of distinct multiplicities implies that subsets of filters will execute at different rates. This means that certain filters may be local bottlenecks in a

Pipeline for example. We suspect that when the number of distinct multiplicities is high this requires more threads to group filters with similar multiplicities. We can also see that the number of threads will depend on certain structural features such as Pipelines, SplitJoins and number of Filters. Yet, these variables seem to hold less influence on the number of threads a program needs than the different multiplicities found in the graph. This is most certainly due to the fact that whilst SplitJoins make parallelizable areas more visible, the amount of work contained in each stream of the SplitJoin, especially when this size is small, may actually make parallelizing the program worse due to ratio of communication to computation.

4.5.1.0.3 KNN Model We chose to use a k-Nearest Neighbor (kNN) to determine the number of threads to use for the application. Given a new application to predict, the kNN classifier determines the k closest generated applications in terms of the features. The distance between the features is measured using the Euclidean for each application. Once the set of k nearest neighbors has been identified, the model simply averages the best number of threads for each of the k nearest neighbors to make a prediction. The parameter k was determined experimentally using only the generated benchmarks. A value of $k = 7$ was found to lead to the best performance.

The features chosen are the variables displayed in Figure 4.8. Using cross validation we determine the efficiency by observing how close a classification is to our measured best thread number. We have determined that our model, using cross validation has a 33% accuracy of getting the predicted best thread number. This increases to 57% when we allow a prediction to be 1 thread away from the best and 67% when 2 threads away. Whilst the performance of pin-point accuracy is disappointing we do not incur more than a 12% performance penalty when choosing a thread number which is +/- 1 from the best and 19% when moving up to 2 threads away from the best. This average is measured by looking at the thread performances without composition.

4.5.2 Predicting Core Composition

4.5.2.0.1 Gathering Training Data Given that the optimal number of cores for a thread is independent of the number of threads found in the program, we only use the single threaded versions to determine the optimal number of cores. For example, all benchmarks will only have a single core per thread when the application is partitioned in 15 threads as this is the maximum amount of cores that may be given to each thread rather than it being the optimal solution. We include multiple versions of the benchmarks using different amounts of unrolling. To determine the optimal number of cores we only select training data that has a performance within 1% of the best.

4.5.2.0.2 Analyzing Features Figure 4.10 shows the highest correlating features with the optimal number of cores. The features are very different from the ones presented in Figure 4.8 and overall there are higher correlating features. The highest correlating value has a correlation factor of 0.88 which represents the number of operations found in a basic block of code. The second feature is similar but only takes into account blocks that will be executed unconditionally, we have chosen to exclude blocks found in loops for this metric as there is still some form of condition for those blocks to be executed. The next two feature compare the size of the average size of an unconditional block to the largest and smallest unconditional block. The fifth feature measures the ratio of the number of unconditional blocks to conditional.

Overall there are no features distinct to StreamIt, such as pipelines or splitjoins that correlate highly with the optimal number of cores. We can thus infer that the optimal number of cores is independent of the structure of a StreamIt program. Instead, it is more dependent on the amount of computation.

EDGE architecture's ability to fetch atomic instruction blocks and out-of-order execution encourages the focus on determining how much speculation is extracted from each filter. Unfortunately StreamIt programs do not tend to have a large quantity of conditional statements and when they do they tend to be quite small. This statement is reinforced by the correlation between the average number of conditional blocks with the optimal number of cores, which is only 0.2, compared to 0.809 for the average size of unconditional blocks. We thus do not focus on using any speculative features from the StreamIt graph.

4.5.2.0.3 Linear Regression Model Given that the optimal number of cores is highly correlated with a few features, a linear regressor is a natural choice to predict the best number of threads. Figures 4.9 represent how the first three highest correlating values affect the number of cores. This figure was obtained by finding the best number of cores for a single threaded benchmark. It is important to note that the top right corner points will always be flat as we can only allocate a maximum of 15 cores.

4.6 Results

This section describes the performance achieved by the model when predicting the number of threads and core composition to use for each of the StreamIt benchmarks.

4.6.1 Evaluation Methodology

Leave-one-out cross-validation is used for testing the linear model. This means that when testing the model on one application, this application is removed from the training set, the model is trained with the remaining application and finally the model is tested on the application. This process is repeated for each application. This is standard methodology in the machine-learning community ensuring that the training data is never used for testing. For the kNN model, the training data consists of all the generated synthetic benchmarks and we only test it on the real StreamIt applications not used for training. To obtain the speedup we compare the performance of our machine learning based result and the best from the sample space to running the StreamIt benchmark on a single core, single thread, using O2 compiler optimisations.

4.6.2 Evaluation

Figure 4.11 compares the performance of the machine-learning model and the best performance from our sample space and core composition. As explained in the earlier section, the sampled best is drawn from a sample size of 1,316 combinations of core compositions and thread partitions for each application when possible. The baseline is the original StreamIt application running with one thread and one core on our dynamic multicore processor. The average speedup obtained through our machine learning model is 2.6, this is only 16% smaller than the average of the best found, which is a speedup of 3.1. These results are positive as it means we are at least within 16% of the total best. As can been seen in Figure 4.11 our largest performance penalty resides in the performance of *ChannelVocoder*.

Table 4.2 presents the actual configuration found for the best sampled point and the machine learning model prediction. Each column represent a different threads and the number in the cell represents the number of core associated with that thread. We can see that for *ChannelVocoder* our model predicts only 8 threads rather than the optimal 13. Refering back to Figure 4.4 and Figure 4.7 from Section 4.4 *ChannelVocoder* always performs better when adding threads. This is the cause of the performance penalty, for *ChannelVocoder* it is more important to allocate a higher number of threads rather than compose cores.

Aside from this case, our machine learning model obtains similar speedups to the best sample.

4.6.3 Summary

This section has demonstrated that it is possible to build a machine-learning model that achieves high level of performance using simple source code static features. In many applications, the model even comes very close to the best from the sampled space, showing that the features used by the model contain enough information to inform the model about the best decision.

4.7 Related Work

4.7.0.0.1 Dynamic Multicore Processors DMPs such as CoreFusion [22] differentiate themselves to EDGE based DMPs on their Instruction Set Architecture (ISA). CoreFusion uses a CISC/RISC based architecture which limits the degree of scalability (fusion), whereas EDGE based DMPs have shown promising scalability [26, 18]. Other types of DMPs such as WidGET [51] and Sharing Architecture [53] present a fine-grain level of composition. In these two architectures, cores can be created out of different components on the processor, including ALUs, floating point units and memory units. This differs from CoreFusion and EDGE where a logical core is composed out of a set of physical cores. This fine-grained composition can allow for even more optimisation but it increases the complexity of the problem.

4.7.0.0.2 Core Configuration Little work has been done on automatically determining the correct core composition for a given application. The work conducted in [22, 26] manually configure their processors before running benchmarks. In [36] they use information provided by the application to determine how to reconfigure some components of the processor. This initial information then assists the rest of the reconfiguration, this process still requires input from the programmer though. Therefore we present a novel method for automating the choice of core composition.

4.7.0.0.3 Streaming Programming Languages There exist streaming languages that target different architectures. For example Brook [5] is designed to be used on GPUs and WaveScript for embedded systems [30]. These languages present different constructs to StreamIt, in particular they lack the graph oriented constructs. Lacking such constructs make these languages less attractive for tiled processors.

4.7.0.0.4 Partitioning StreamIt on multicore chip Previous work on scheduling streaming applications onto DMPs or heterogenous multicore chips focuses on finding mathematical ways of partitioning the graph onto the chip [7, 27]. In Carpenter et al.'s work [7] they restrain themselves to partitioning a StreamIt application maintaining connectedness. Connectedness can be defined as a subgraph where the filters are connected. This restriction reduces the number of potential partitions that can be generated by their algorithm and will put TLP in favour of ILP. Kudlur et al. in [27] choose to represent the partitioning problem as an integer linear programming problem. They start by fissionioning stateless filters to obtain the optimal load balance across all cores and assign the filters to a core using a modulo scheduler. Farhad et al. also use integer linear programming in [15] to schedule StreamIt programs on multicore. They profile the communication costs of the streaming programs by running the program using different multicore allocations and feed that information into their integer linear programming model.

4.7.0.0.5 Machine Learning Using a machine learning model to partition StreamIt programs was previously explored in the work of Wang et al. in [50]. They use a k nearest neighbor model to determine the perfect partitioning of a StreamIt program for a multicore system. The features we extracted using correlation analysis are similar to those presented in the work of [50]. Unlike our work their model is used to find ways of fusing and fissioning filters to discover a new graph that can then be mapped onto a multicore system.

4.8 Conclusion

In this paper we presented the problem of partitioning both software and hardware for a Dynamic Multicore Processor. We analysed a set of streaming workloads based on StreamIt, extracting features which highly influence both the required number of threads and core composition. Using this data we introduced a machine learning model which is able to determine how many threads a StreamIt application needs and pick an appropriate chip topology. The model predicts configurations close to the performance of the best design points from the sampled space. By automating the decision of core composition we motivate the use of DMPs for accelerating applications without any involvement from the programmer.

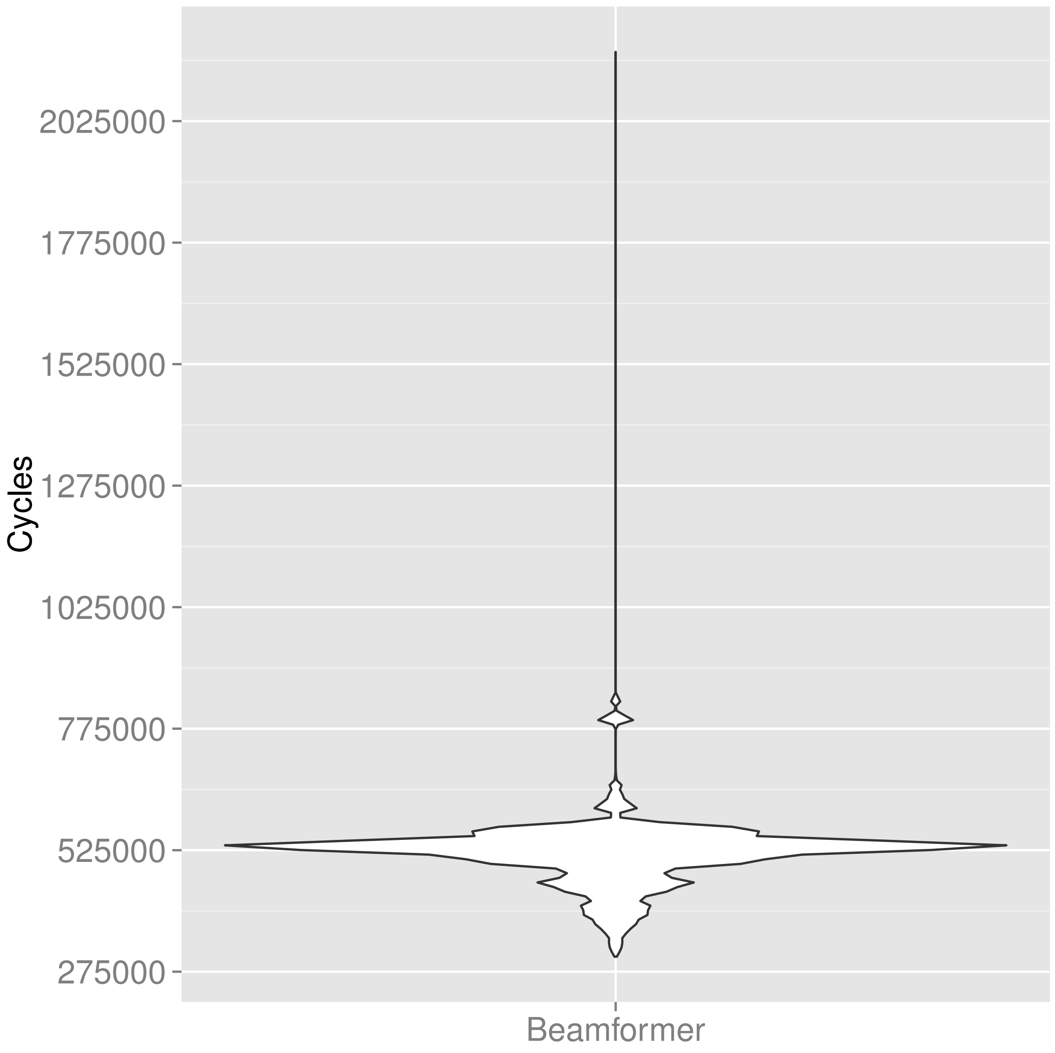

Figure 4.1: Distribution of the runtime for Beamformer resulting from an exhaustively exploration of the hardware/software co-design space. The application has been partitioned into different number of threads and core compositions.

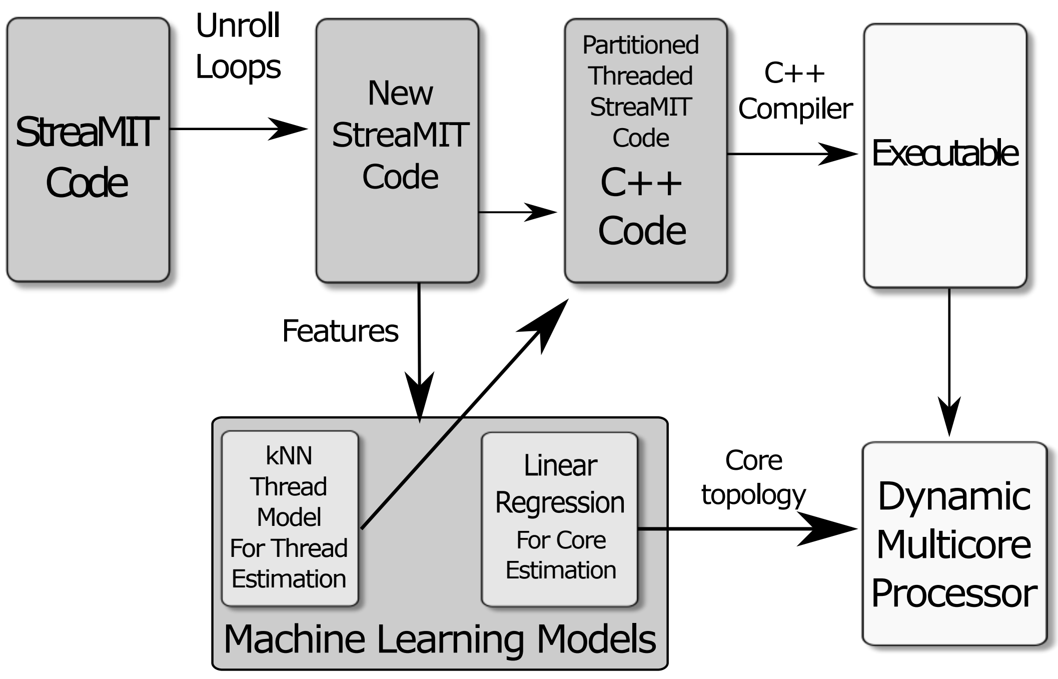
StreaMIT Code
Unroll Loops
New StreaMIT Code
Partitioned Threaded StreaMIT Code
C++ Code
C++ Compiler
Executable
Features
kNN Thread Model For Thread Estimation
Linear Regression For Core Estimation
Machine Learning Models
Core topology
Dynamic Multicore Processor

Figure 4.2: Description of our workflow. Two distinct machine-learning models are used to predict the optimal thread partitioning and core composition based on static code features.

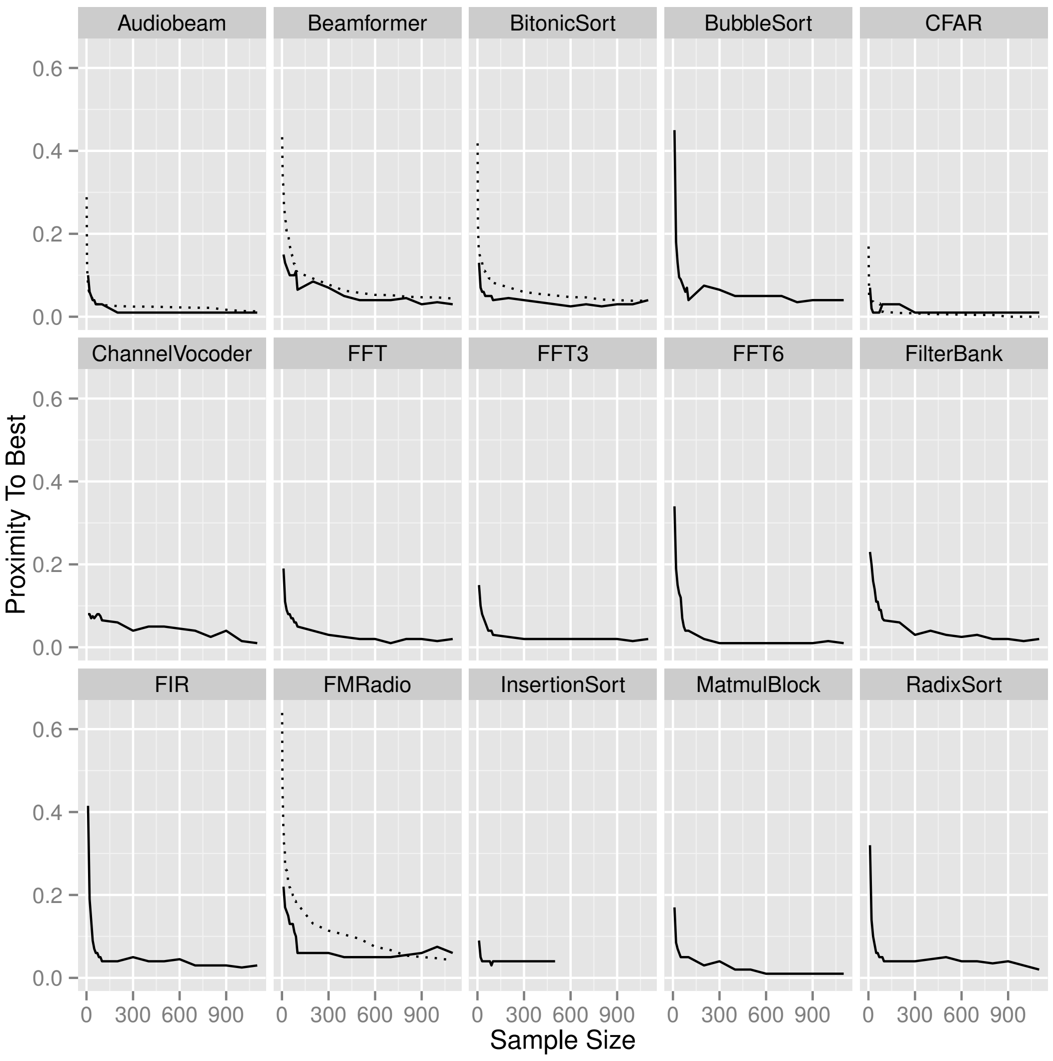

Figure 4.3: Statistical (plain line) and actual proximity (dotted line) to best performance using a subset of the sample space.

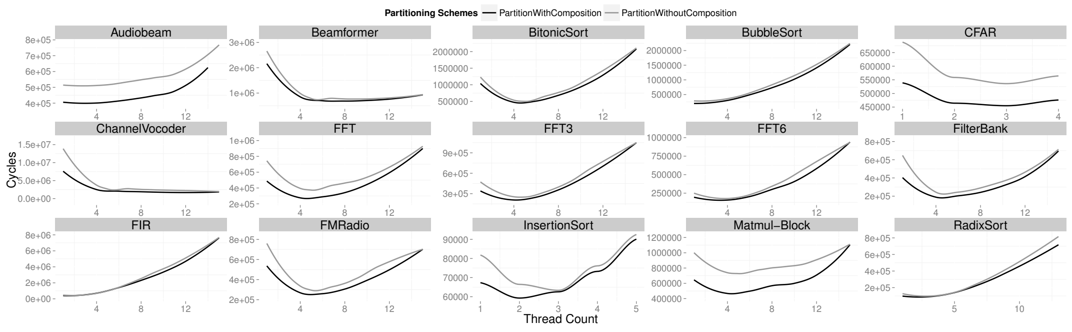

Figure 4.4: Performance as a function of the number of threads. The performance metric is number of cycles. Each benchmark has the performance measured with cores composed and with threads mapped to a single core.

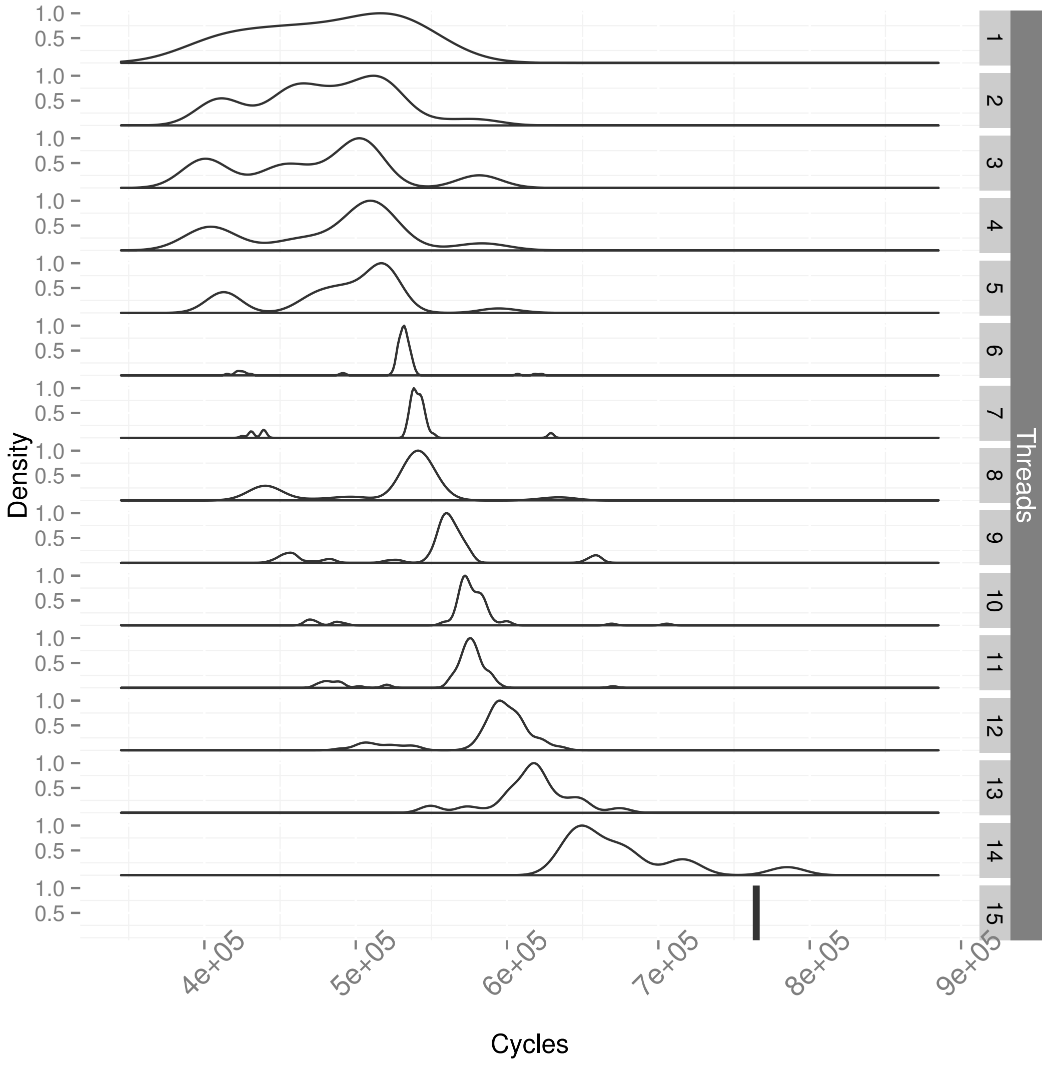

Figure 4.5: Distribution of Audiobeam performance when modifying the amount of threads and compositions.

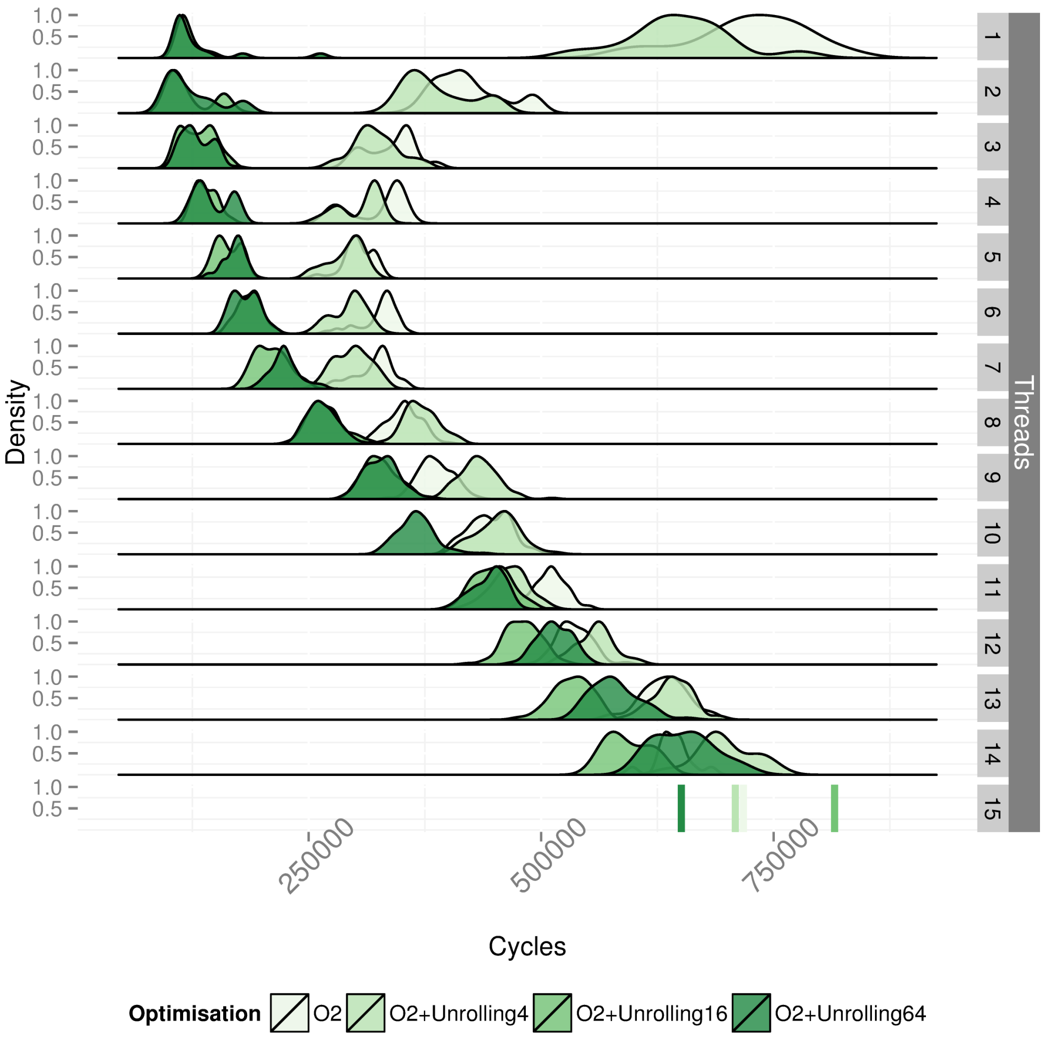

Figure 4.6: Distribution of FMRadio performance when modifying the amount of threads, composition and unrolling factor.

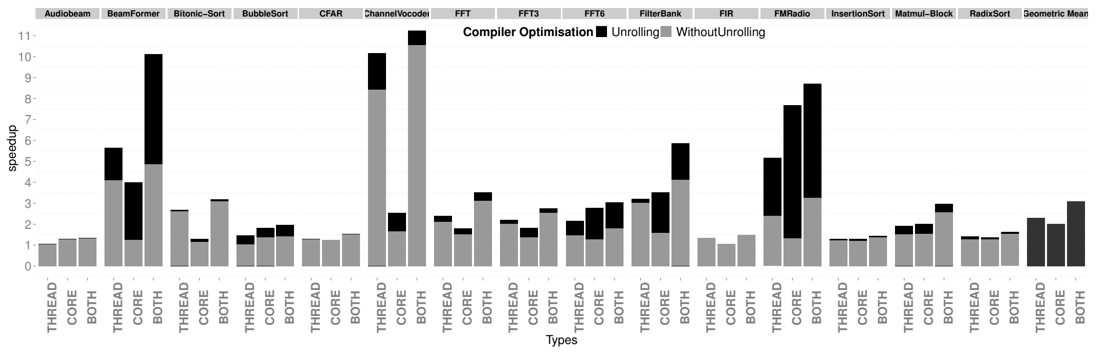

Figure 4.7: Speedup obtained by choosing best core composition, best thread number and the combination of both optimisations. The baseline for the speedup measurement is single core, single thread execution using O2 compiler optimisations. Higher is better.

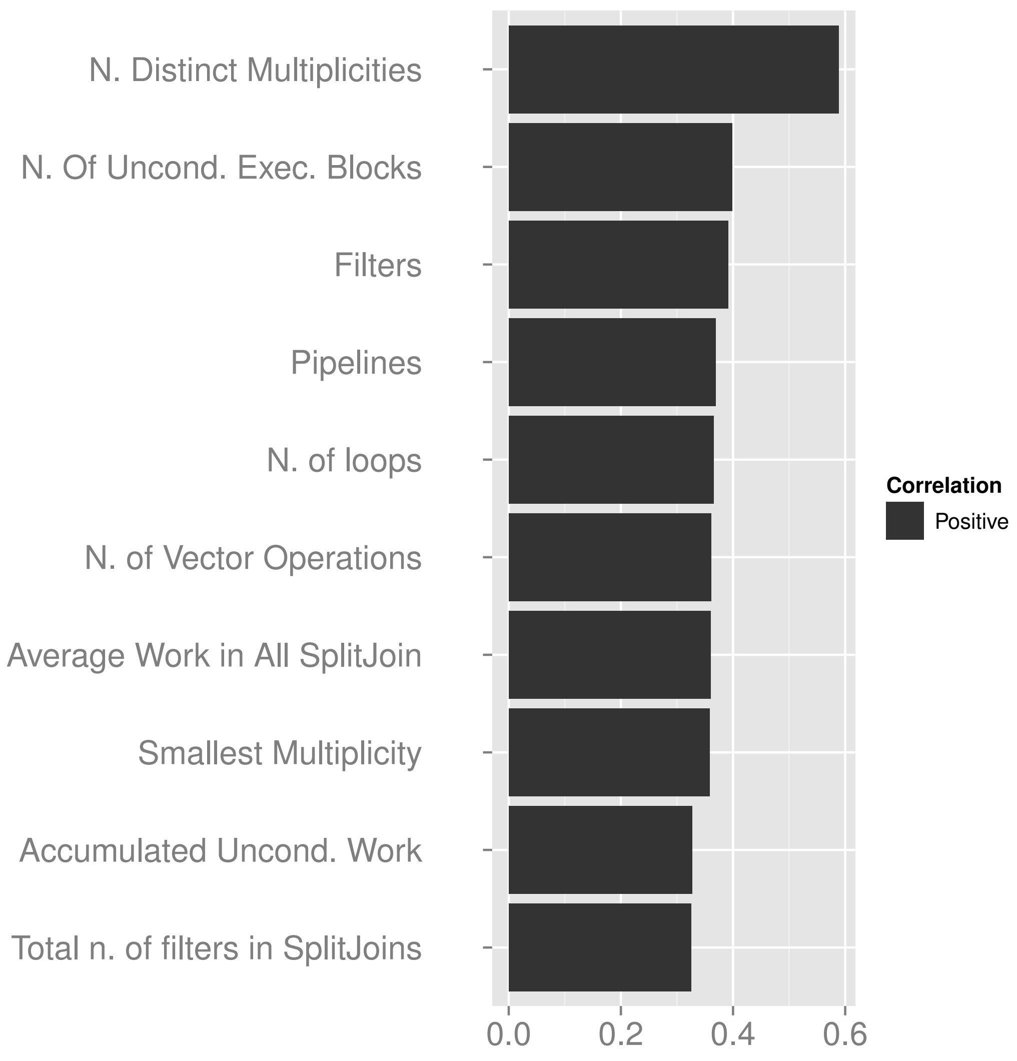

Figure 4.8: The ten highest correlating features with the best number of threads for 1000 synthetic benchmarks.

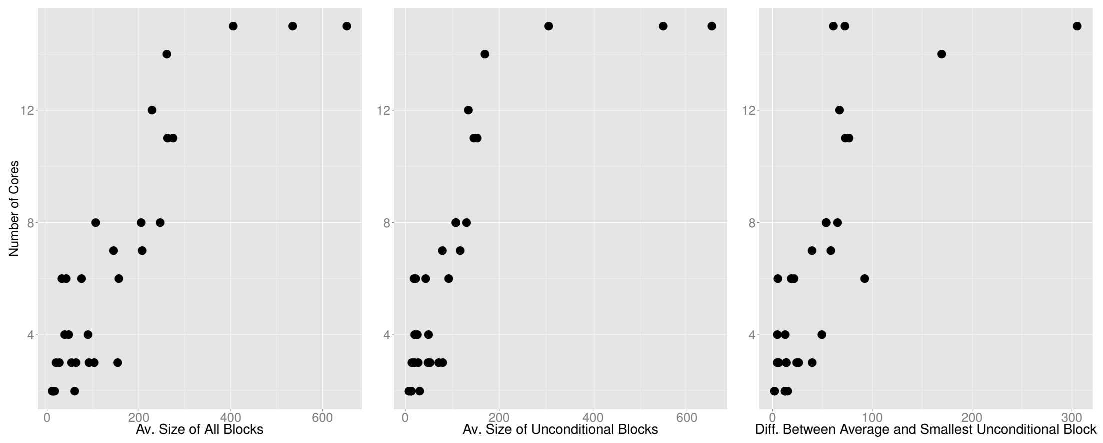

Figure 4.9: Optimal number of cores in relation to the three highest correlating features. The maximum number of cores plateaus on the right hand side as this is the maximum possible amount.

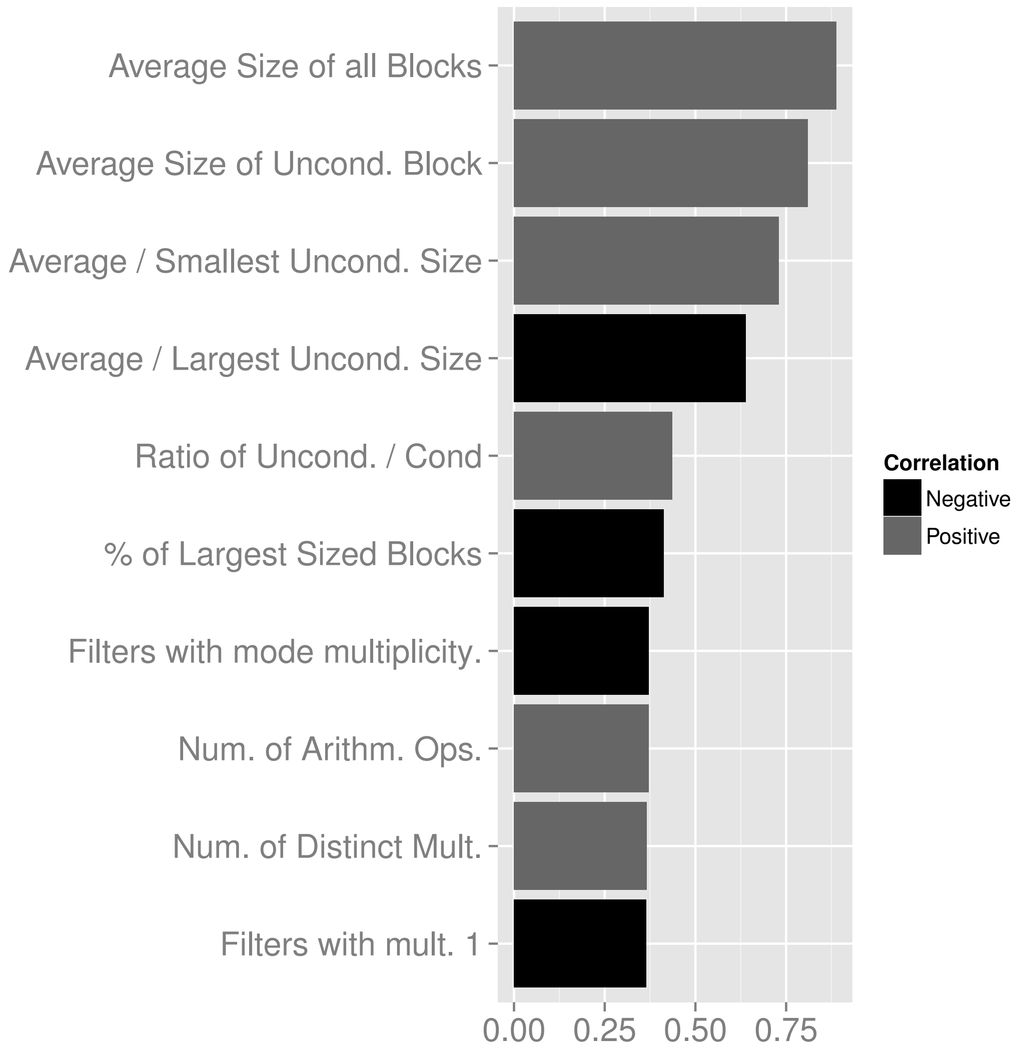

Figure 4.10: The ten highest correlating features with the optimal number of cores.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
B Audiobeam	3	2								
M Audiobeam	2	3								
B Beamformer	1	4	2	4	4					
M Beamformer	6	4	4							
B BitonicSort	3	2	2	2						
M BitonicSort	1	2	2	1	2	2	2			
B BubbleSort	3	3								
M BubbleSort	2									
B CFAR	3	2								
M CFAR	2	2	1	2						
B ChannelVoc.	4	1	1	1	1	1	2	1	1	1
M ChannelVoc.	2	2	1	2	2	2	2			
B FIR	3	2								
M FIR	2	2								
B FFT	3	3	5							
M FFT	6	5	2							
B FFT3	3	2	2							
M FFT3	3	2	3	3	3	3				
B FFT6	7	8								
M FFT6	14									
B FilterBank	4	5	6							
M FilterBank	4	5								
B FMRadio	7	6								
M FMRadio	7	4								
B InsertionSort	3	2								
M InsertionSort	3									
B MatmulBlock	3	4	6	2						
M MatmulBlock	4	4								
B RadixSort	3	3								
M RadixSort	2	2								

Table 4.2: Number of Threads and Cores used for Best of Sample Space and Machine Learning Model.

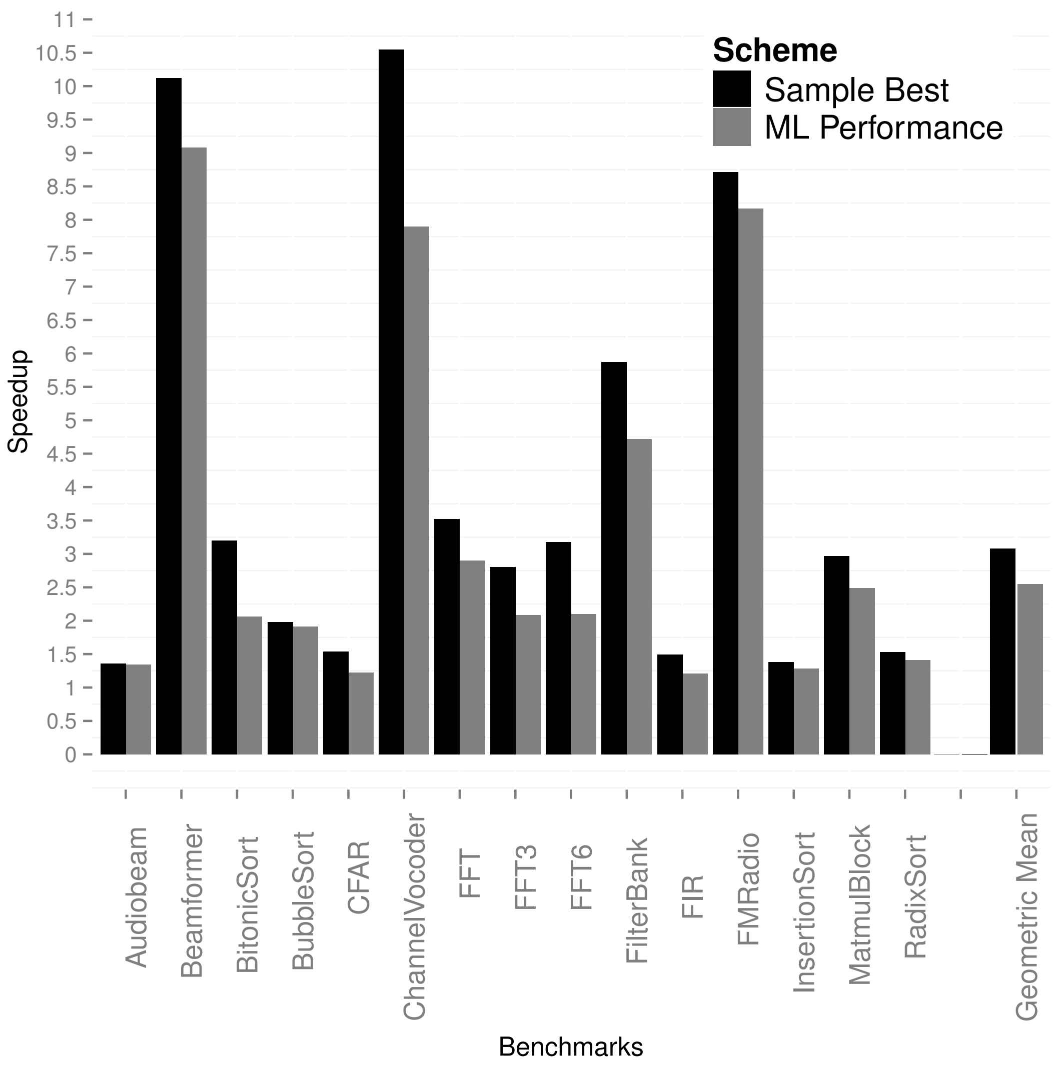

Figure 4.11: Performance of our machine learning model against the best execution from random sampling. The baseline for the speedup measurement is single core, single thread execution using O2 compiler optimisations. Higher is better.

Chapter 5

CASES

5.1 Introduction

Chip Multicore Processors (CMP) are now ubiquitous in embedded computing as single threaded performance improvements have slowed. CMPs have to be carefully designed, balancing the size of each core with the total number of cores on the chip. Larger cores are typically good at exploiting instruction level parallelism (ILP) but might potentially be very power hungry. Smaller cores on the other hand require less power but offer limited performance, forcing software developers to parallelize their code with multiple threads, which is a tedious process. As the size and the number of cores is fixed at design time, choosing the right balance is difficult [11, 43].

Asymmetric Chip Multicore Processors (ACMP) have been proposed [29] to overcome this issue. These processors feature either different sized cores [23] or different Instruction Set Architectures [46] to efficiently tackle a multitude of different workloads. Dynamic Multicore Processors (DMP) push this further by introducing Core Fusion [22]. Similar to ACMPs, Core Fusion allows the chip to have different sized cores, but this can be changed at runtime. In a DMP, cores can be fused dynamically to create larger cores similar to a superscalar processor. Any number of cores can potentially be combined together whenever a workload exhibits a large amount of ILP. When a program exhibits low ILP, the DMP can decouple fused cores to conserve energy.

While a large number of DMPs have been proposed in the literature [22, 32, 25, 51], these efforts focus on the hardware and microarchitectural design. They evaluate the hardware using a fixed number of fused cores or provide an oracle for dynamic fusion. There exists little [28] to no literature on predicting core fusion from a software perspective. To the best of our knowledge, there has been no study on dynamically changing the number of cores fused to better match the phases of a workload in a homogeneous DMP compared to ahead of time fusion.

We start with an explanation of the theoretical limitations of core fusion and what we can expect in terms of performance. We then discuss how classical loop optimizations such as unrolling can have a large impact on performance when fusing cores. Using the San Diego Vision Benchmark Suite [47] (SD-VBS) as a use case, we show that programs exhibit various phases with different amounts of ILP. We then perform a limit study on the potential for decreasing energy consumption while maintaining performance when adapting the number of cores for each program phase. Our results show that using dynamic core fusion can save up to 42% on average while maintaining the same level of performance as a fixed number of cores. We also show how latency introduced by reconfiguring the system can influence the impact of core fusion. Finally, we build a simple online model using linear regression that predicts the optimal number of cores per phase for reducing energy consumption while maintaining performance. This practical model leads to an average of 37% saving in energy with no performance loss.

To summarize, our contributions are:

- We analyze the limits of core fusion using an analytical model.
- We study the loop optimizations required to ensure efficient use of core fusion.
- We offer an in-depth comparison of static and dynamic core fusion schemes on the San Diego Vision Benchmark Suite.
- We show that core fusion has the potential to offer a large reduction in energy savings.
- We show how a simple linear-regression based model can predict the number of cores to fuse for different program phases.

5.2 Motivation

This section motivates the use of dynamic core fusion and its impact on performance and energy. It also shows that loop optimizations have a significant performance impact when fusing cores.

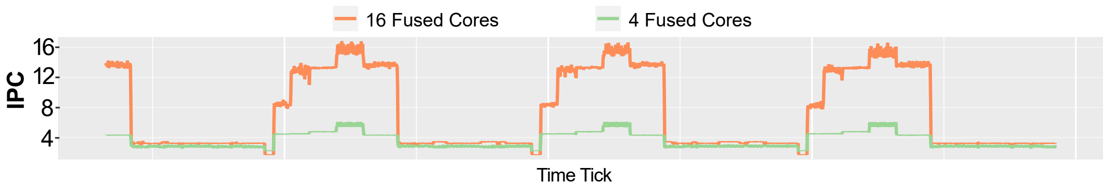

Figure 5.1: IPC of a typical benchmark (Disparity from SD-VBS) when executing on a fused 4 or 16 core processor.

5.2.1 Dynamic Core Fusion

Previous work in core fusion has focused on delivering performance improvements [22, 25] and demonstrated how to predict static core fusion [28]. A static fusion will fuse cores into a single logical core (LC) and execute a thread on this new core. As evident from this prior work, fusion improves the performance of the program by maximizing speed. However, as we will show, static core fusion may not be the perfect match for all situations.

Figure 5.1 plots the Instruction Per Cycle (IPC) performance variation over the execution of the *Disparity* Benchmark [47] on a fused 4 core and 16 core processor. On 4 cores, the performance oscillates between an IPC of 2 and 6 depending on the phase while on 16 cores the IPC can be as high as 16. More importantly, for some phases (half of the time), the same level of IPC is achieved (˜3) whether the program runs on 4 or 16 cores. A DMP could exploit this to minimize energy consumption while maximizing performance by fusing only 4 cores during the low IPC phases and fusing 16 cores during the high IPC phases.

5.2.2 Code Optimizations

When cores are fused they execute blocks of instructions in parallel on each physical core in the LC. In order to obtain the best results, we must generate large blocks as this leads to a higher IPC on the LC [28]. The optimizations we apply include aggressive loop unrolling, inlining and replacing conditional statements with either software predication or architecture-level predication. These optimizations are well known and do not require any structural modifications of the program.

Figure 5.2 illustrates the impact of applying loop transformations compared to a standard compiler not specifically tuned for an EDGE architecture. As can be seen, the impact of these transformations can be large and are necessary to sustain a high IPC for a long enough period. More details about the loop transformations are given

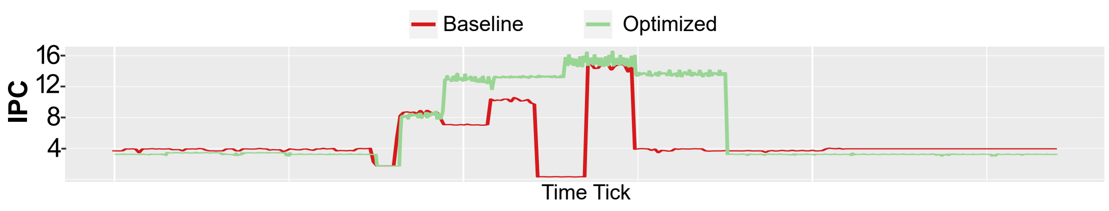

Figure 5.2: Impact of loop transformations on fused cores for the Disparity benchmark.

in section 5.5 but this example illustrates the need for careful tuning of the compiler to achieve high performance on such an architecture.

5.2.3 Summary

This section has shown that programs exhibit phases with various amount of ILP available. A dynamic multicore processor can take advantage of this property to fuse a large number of cores for the high-ILP phases and fuse a smaller number of cores when ILP drops to conserve energy. We have also illustrated the importance of fine-tuned code transformations to achieve sustained performance and increase the potential for fusing cores. The next section will study in more details the expected impact of fusion using an analytical model.

5.3 A Study of Core Fusion

In this section we study the performance limitations of fusing several cores into a single logical core (LC). This allows us to better understand what leads to good performance and how to determine regions of code that benefit from core fusion. The two major obstacles to gaining performance with core fusion are branch prediction and synchronization costs.

5.3.1 Branch Prediction

As discussed in section **??**, DMPs accelerate a single thread by executing instructions from the thread speculatively across several fused cores. Similarly, EDGE DMPs accelerate a single thread by speculatively executing instruction blocks [32, 34]. Fusion puts a strain on the branch predictor since efficiently using the fused cores depends on the misprediction rate. The branch predictor has to meet a different accuracy requirement depending on the size of both the LC and average size of a block being executed.

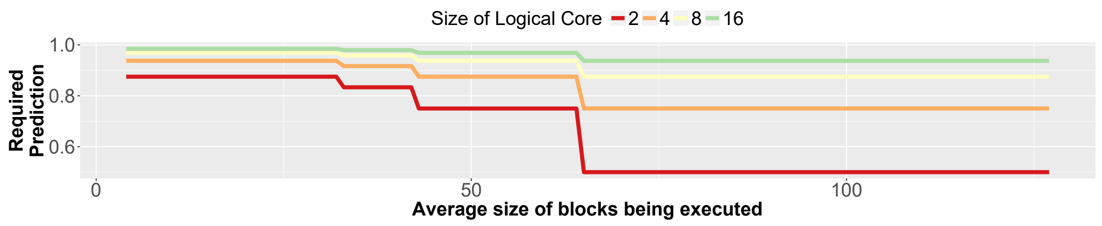

Figure 5.3: Required prediction accuracy for a logical core size to be efficient given an average block size.

Given a Logical Core *LC* of size *i*, denoted LC_i, we can determine the minimum branch prediction requirement using Formula 5.1 where *BlocksInFlight* represents the number of total blocks being executed on the *LC*.

$$min_{PredLC_i} = \frac{BlocksInFlight - 1}{BlocksInFlight} \tag{5.1}$$

BlocksInFlight will vary depending on the average size of the blocks, the maximum block size the architecture can execute (*MaxBlockSize*), and the number of blocks that are allowed to execute in parallel on a physical core (*NumOfBlocksPerCore*). The size of the instruction window is equivalent to *MaxBlockSize* multiplied by *NumOf-BlocksPerCore*. When a program is running on a LC, one of the blocks will always be unconditionally executed, which is why we require one less block to be predicted.

Figure 5.3 shows the expected prediction accuracy required to fully utilize a LC given the average block size in flight. In this figure, NumOfBlocksPerCore is equal to four and MaxBlockSize is 32. Adding extra physical cores to a LC requires an increasingly accurate branch predictor, especially when the size of a block is under 50 instructions. This informs us in two ways; first of all large LCs will need to run on code sections with less control flow as they are more sensitive to branch misspredictions. Second of all, branch prediction can be a simple method of evaluating the current effectiveness of a LC. Given a certain number of cores, if the prediction accuracy is under the limits presented in Figure 5.3 we can easily determine that the LC is sub-optimal.

5.3.2 Synchronization Cost

For a program to execute correctly, the cores in a logical core (LC) must communicate when they have finished executing a block. This ensures that the cores fetch blocks

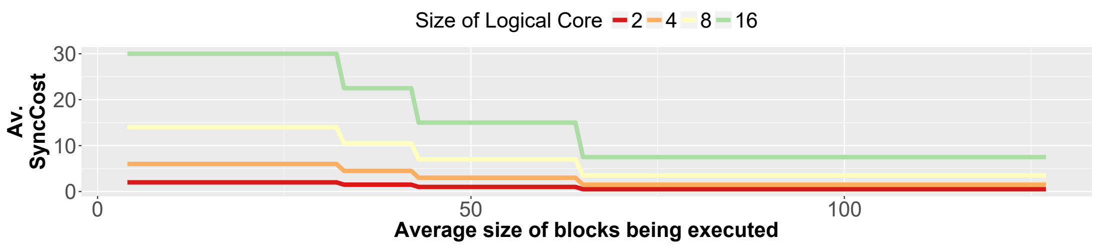

Figure 5.4: Synchronization Cost in cycles for a given number of cores in a composition and an average block size.

from the correct control paths and update memory consistently. A core may have to wait for other cores to commit before fetching a new block. We define the worst-case estimate of this stall as the **Synchronization Cost**.

Blocks commit in a sequential fashion with the non-speculative block committing first and the most recent speculative block committing last. If a core's instruction window is full then it must commit a block before it fetches a new one. The Synchronization Cost, in cycles, is defined in equation 5.2 and is measured by averaging the overall number of cycles each fused core waits until it can continue to fetch and execute new blocks. *AvBlocksInFlight* represents the average number of blocks in flight on a single core in the LC. This is a worst-case estimate as block sizes will fluctuate during the execution of a program.

$$SyncCost_i = \frac{\sum_{n=0}^{i-1} (AvBlocksInFlight) \times n}{i} \tag{5.2}$$

Figure 5.4 shows how many cycles the Synchronization Cost will be for a given LC and average block size. The larger the block size the lower the Synchronization Cost is since cores fetch fewer blocks and wait less for other fused cores to finish committing. We can see that large LCs executing small blocks have a high Synchronization Cost. This indicates that large LCs should be avoided when dealing with smaller blocks as the Synchronization Cost outweighs the code execution.

5.3.3 Summary

This section estimated the worst-case IPC for a logical processor using Average Block Size, Average Branch Prediction, and Synchonization Cost. Figure 5.5 presented a worst-case estimate of IPC performance assuming each core can sustain an IPC of

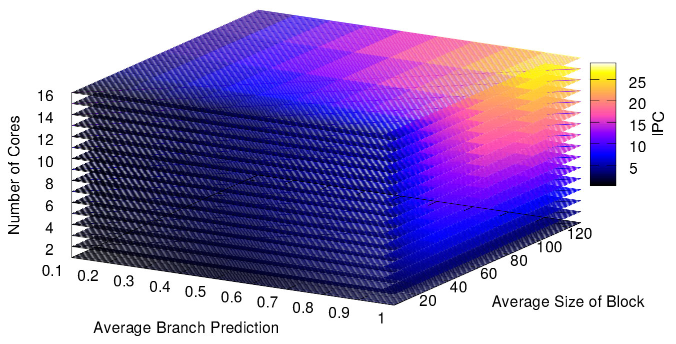

Figure 5.5: IPC estimate given a logical processor size, average branch prediction and average block size for a dual-issue core. A higher IPC means better performance.

2. From what we previously learned, Figure 5.5 shows us that to obtain optimal performance requires a high branch prediction accuracy and large blocks. It shows that larger logical processors can easily under-perform; for example we can see that 16 fused cores often have IPCs under 15, meaning that each core has an IPC under 1.

5.4 Experimental Setup

The previous section studied the performance potential for core fusion using an analytical model. We now present the experimental setup used for the remaining parts of the paper where we conduct a thorough evaluation of core fusion with a cycle-level simulator.

5.4.1 Benchmarks

For this paper we study the performance of our Dynamic Multicore Processor (DMP) on a set of Vision Benchmarks designed for hardware and compiler research [47]. The San Diego Vision Benchmark suite (SD-VBS) is composed of nine single-threaded C benchmarks ranging from image analysis to motion tracking. These benchmarks represent state-of-the-art applications in image and vision recognition which are prevalent in embedded systems.

Vision applications typically have regular and simple control flow which enables the formation of large blocks of instructions. Our processor relies on the ability to form large blocks to exploit ILP which makes these applications particularly well suited. As the results will show, the phase length has minimal impact on energy savings when the reconfiguration overhead is low.

5.4.2 Architecture and Simulator

We use a cycle-level simulator of an EDGE-based Dynamic Multicore Processor [37] whose core pipeline is verified against an RTL implementation within 5%. This validation is done by running workloads on RTL and comparing the traces cycle-by-cycle with the software simulator. The architecture and core fusion mechanics are similar to the work described in [25, 34]. We configure the simulator to model a 16 core multiprocessor, with 32 KB private L1 caches, and allow each core to fetch up to 4 blocks of instructions, and issue up to 2 instructions per block for a maximum of 64 blocks in flight.

5.4.3 Compiler

Each benchmark is compiled with the Microsoft C++ compiler for EDGE [37], with -O2 optimisations and using instruction predication for hyperblock formation [38].

5.4.4 Measuring Performance and Power

We run five simulations per benchmark, one for each LC size: 1, 2, 4, 8 and 16. For each LC we record the IPC of the LC at an interval of 640 committed blocks. We selected 640 committed blocks as it allows each core in a LC to execute enough blocks before taking the measurement. This is due to the fact that the highest LC of 16 cores can execute up to 64 blocks at a time, thus recording performance after 640 blocks allows each core to have executed at least 10 blocks. Using committed blocks as an interval allows us to easily compare each simulation as the total number of committed blocks does not change even if the LCs are different.

Due to the fact that we study an EDGE ISA [38], we cannot use McPAT to model power consumption as it differs from traditional CISC/RISC cores modeled in McPAT. Instead we use a coarse grained power model where either a core is turned on or or it is off.

5.5 Code Optimizations

This section describes optimizations focused on reducing control flow and expanding block sizes which is necessary for high performance as seen in section 5.3.

```
for(int i = 0; i < 1000; i++)
  for(int j = 0; j < 1000; j++)
     for(int k = 0;k < 5; k++)
         a[i][j] = a[i][j] * b[k][j];
```

Figure 5.6: Example of an inner-most loop which should be completely unrolled.

```
for(int i = 0; i < 1000; i++)
  for(int j = 0; j < 1000; j++)
      a[i][j] = a[i][j-1]
                          * b[i][j];
```

Figure 5.7: Example of a data dependency which can be removed by interchanging the loops.

5.5.1 Loop Unrolling

Loop unrolling is a common optimization used to reduce the overhead of the loop header and to better expose Instruction Level Parallelism (ILP). When dealing with tightly-knit loops, logical cores may perform poorly due to the fact that they execute many small blocks, thus increasing the Synchronization Cost. Unrolling loops will both reduce the number of blocks required to execute the loop and increase the size of the blocks, thus reducing the Synchronization Cost and increasing ILP. For example, the innermost loop in Figure 5.6 should be completely unrolled and its outer loop unrolled partially to increase the block size. There are certain factors which can limit the usefulness of loop unrolling which we examine later on. In the architecture we evaluate, we may not have more than 32 load or store instructions per block. Therefore, if we unroll memory intensive loops, we must ensure we do not go above this threshold. Going above this threshold leads to creating a new block which will put a strain on the branch predictor. Another issue is that unrolling loops with conditional statements may not help improve the size of the block as the conditional branches might still segment the new blocks. So we should avoid unrolling such loops.

5.5.2 Loop Interchange

When dealing with nested loops there is one reason we have determined for interchanging the loops. The case arises when interchanging the loop removes dependencies in the inner-most loop. The dependency in Figure 5.7 can be removed by interchanging the loops. This allows us to unroll the inner loop efficiently, but also remove any kind of dependency between blocks. Since two blocks from the same loop may execute on different cores, we want to reduce any kind of data dependency, minimizing core communication.

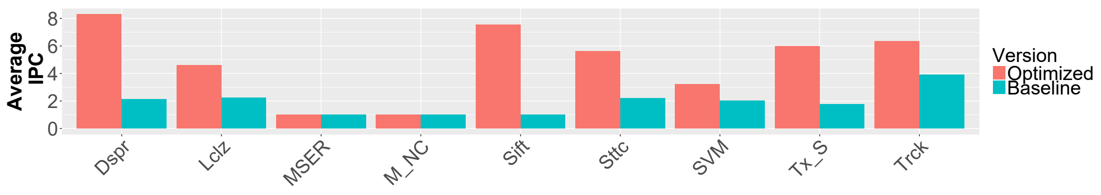

Figure 5.8: Average IPC using the optimal sized logical-core, with and without optimizations. Higher is better.

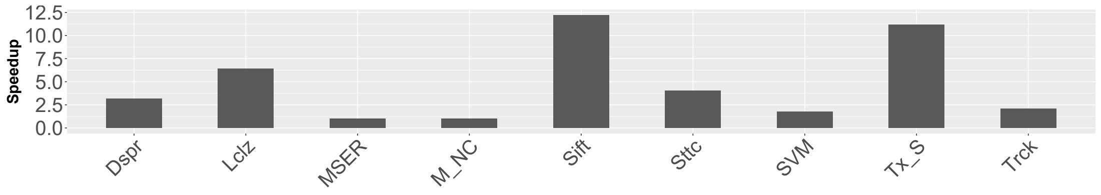

Figure 5.9: Speedup from using code-optimizations over baseline source code using the same optimal sized logical-core.

5.5.3 Predication and Hyperblock Formation

EDGE compilers must split blocks whenever control-flow is present [38]. If a loop contains a conditional statement, the loop body has to be split in two unless if-conversion is applied. Hyperblock formation aims to reduce branching and increase block size by combining two or more blocks into a single predicated block [38]. Hyperblocks reduce both synchronization cost and branch prediction requirements as discussed previously. This is especially important in control-flow intensive loops where unrolling increases the number of conditional statements.

5.5.4 Results

While the optimizations described above and their tuning would be easy to implement in a compiler, we did not have access to the compiler's source code. We therefore modified the source code of our benchmarks by manually interchanging or unrolling loops. In the case of predication and hyperblock formation, we converted simple if-then-else statements into ternary operators whenever possible. We also tried to reorder statements within the body of a loop to avoid having control flow in the middle of the body. We then verified that our source code modification had the intended effect by dissembling the binary produced by the compiler. We modified between 0 and 12 loops depending on the benchmark.

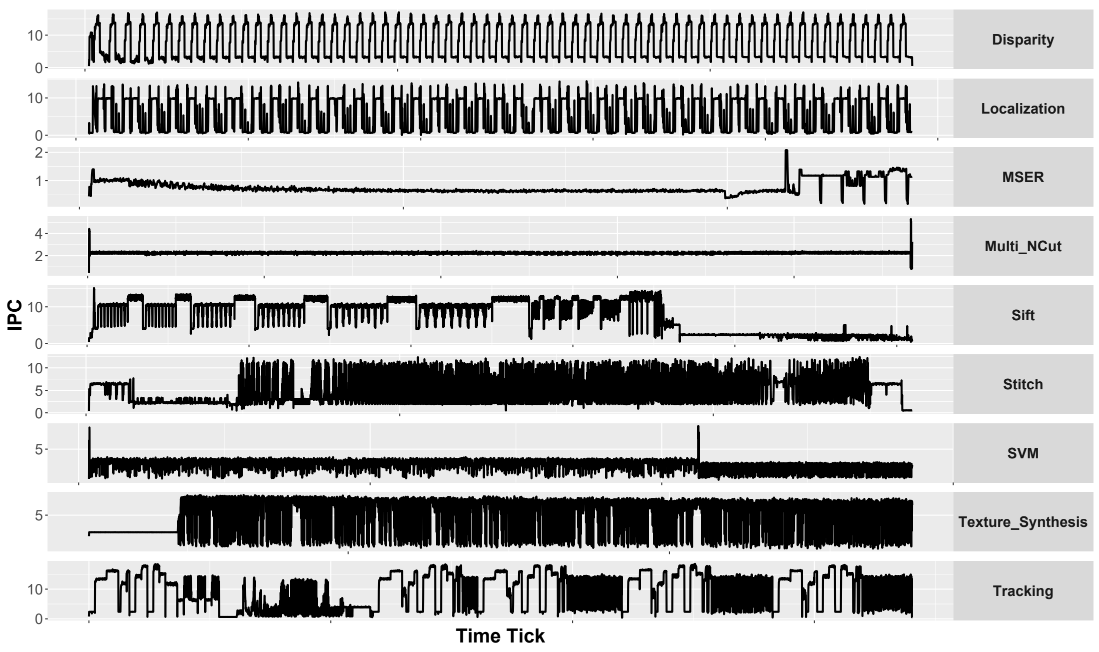

Figure 5.10: IPC as a function of time for each benchmark when run on 16 fused cores.

We compare the best static core fusion using the optimized code with the unmodified code, both version compiled with `-O2`. Figure 5.8 shows the resulting IPC for the baseline case and the optimized benchmarks when run on a core with the optimal number of fused core to maximize performance. The IPC of the baseline is very low for the majority of the benchmarks which might give the impression that core fusion is rather inefficient. However, after applying the simple optimizations described above, the average IPC is significantly increased in many cases.

Since optimizations change the total number of instructions, we also show the actual speedup obtained using cycle count in Figure 5.9. As we can see, benchmarks *MSER* and *Multi-NCut* do not perform any differently. This is due to the fact that none of these optimizations can be successfully applied on these benchmarks. For the other benchmarks we see significant improvements of up to 12$\times$ for *Sift* when the optimizations are applied.

5.5.5 Summary

Overall, this section shows that classical loop transformations can have a large impact on the performance of fused cores. Without these optimizations, it would be more difficult to motivate the use of core fusion even at a static-level as the IPC does not deviate enough from a single core.

5.6 Benchmark Exploration

This section explores how core fusion affects the performance of the SD-VBS benchmarks. We first perform a phase analysis, followed by a study of the IPC variation for static core fusion. We then motivate the use of dynamic core fusion using the information gathered.

5.6.1 Phase Detection

Figure 5.10 presents the IPC performance through time for all the benchmarks when using a logical core (LC) composed of 16 cores. The IPC is calculated for each time tick, which is set at interval of 640 blocks committed. The IPC varies a lot for some of the benchmarks such as *Disparity* or *Localization* where we expect dynamic fusion to be especially good. For other, such as *Multi_NCut*, the execution is dominated by a single long phase with constant IPC, which will clearly show no benefit from using dynamic fusion.

To better understand how dynamic core fusion improves performance, either by improving speedup or reducing energy, we first study how each benchmark features different phases during their execution. For every benchmark we regroup the IPC results of 16,8,4,2,1 fused cores and use kMeans clustering to determine phases. This process is only done for the purpose of exploring this set of benchmarks. Intervals that exhibit similar IPC values when run on different core counts are classified in the same cluster. In order to find the correct number of clusters we plot the Sum of Square Errors (SSE) for a given cluster size from 1 to 15 and determine the optimal cluster to be in the elbow in the plot [12].

Figure 5.11 shows us the optimal number of clusters for each benchmark and the frequency of each cluster. The data can be corroborated with the information found in Figure 5.10. For example, benchmarks *MSER* and *Multi_NCut* feature two phases, with one dominating phase. This means that it will be impossible to obtain any kind of performance improvements through dynamic reconfiguration. For all the other benchmarks, they each have at least two dominant phases. Since each phase is a cluster of similar IPC values, having two or more clusters will result in a higher chance of benefiting from dynamic core fusion.

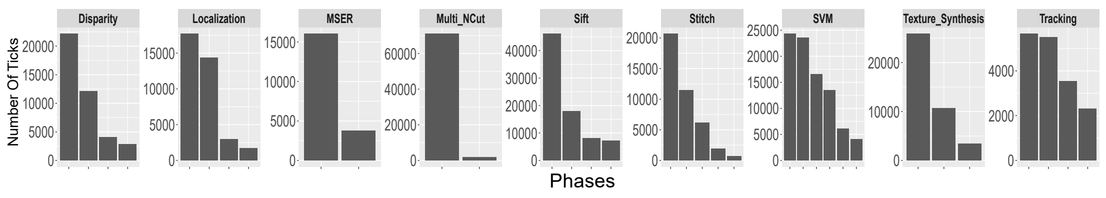

Figure 5.11: Number of phases determined for each benchmark using kMeans clustering and their distribution.

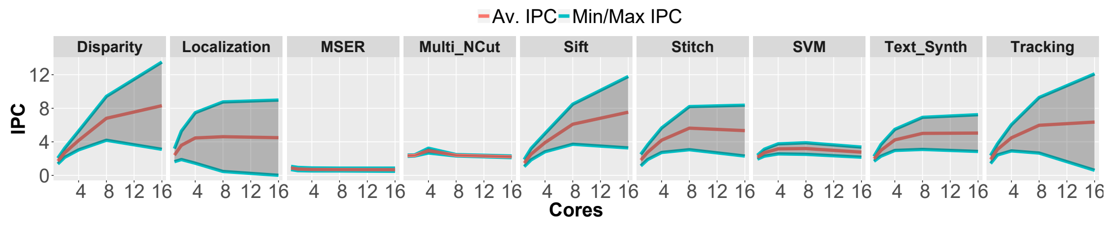

Figure 5.12: Comparing average, smallest and greatest IPC for each SD-VBS benchmark using logical-core size of 16.

5.6.2 Static Core Fusion Exploration

Figure 5.12 shows how the average Instructions Per Cycle (IPC) changes as we increase the size of a LC, going in powers of 2 from 1 to 16 fused cores. We see that, for most benchmarks, fusing more cores provides an increase in IPC performance. Benchmarks *Disparity*, *Localization*, *Sift*, *Stitch*, *Texture Synthesis* and *Tracking* all at least observe a speedup of 2x when using core fusion.

However increasing the size of a LC is not always beneficial as can be seen in benchmarks *Localization*, *MSER*, *Multi_NCut*, *Stich*, and *SVM*. For benchmarks *Localization* and *Stitch* the performance increases when fusing up to 8 cores, where-as *MSER* and *Multi_NCut* never benefit from core fusion. Referring back to Figures 5.10 and 5.11, we can see that *MSER* and *Multi_NCut* feature one dominating long phase, both performing poorly. This explains the lack of scaling for these two benchmarks.

Figure 5.12 also shows the standard deviation of the IPC for each given LC size represented by the grayed out areas. For example, running the *Disparity* benchmark on a LC of 16 cores, we have an average IPC of 8.3 with a standard deviation of 5.2. The standard deviation for 16 cores shows that the performance can drop down to 2.5. An IPC of 2.5 when using 16 cores is very inefficient as this represents 0.1 of an instruction per cycle for each core. We can observe that when using a LC of size 4 for the *Disparity* benchmark we achieve an average of 4.1 with a standard deviation of 1.2.

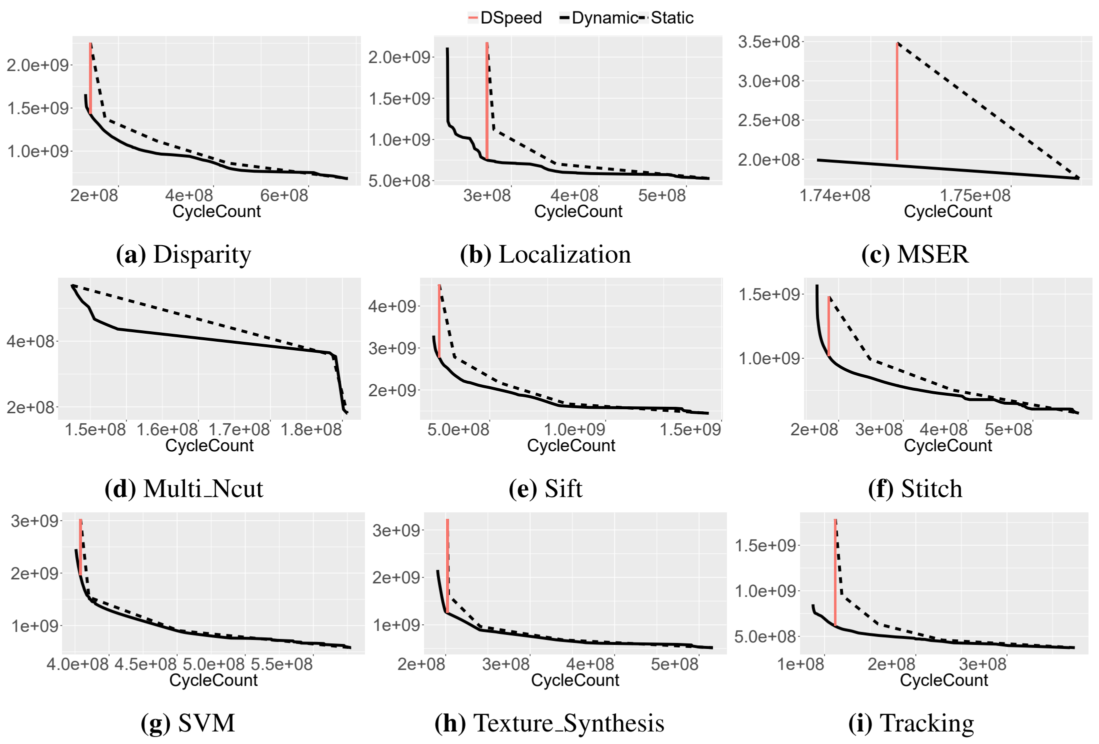

Figure 5.13: Time (x-axis) vs. Energy (y-axis) tradeoffs using Static and Dynamic Composition Schemes.

Thus, if the logical-core could change size, there is a possibility that we could reduce the overall energy consumption of the system by switching from 16 to 4.

Overall, most benchmarks that benefit from large logical-cores will also be met with important standard deviations of IPC performance. The high standard deviation is evidence of performance phases found in each application which are likely to benefit from dynamic adaptation.

5.7 Dynamic Core Fusion

Having studied the behavior of our program under a fix number of cores, we now study the impact of varying the number of fused cores throughout program execution. We first describe how we generate traces for the dynamic core fusion schemes. Before we begin the analysis we define two types of static core fusion:

- **Static Benchmark**: A fixed fused-core which is optimal for the benchmark at hand (SB).

- **Static Suite**: A fixed fused-core which represents the average best for the entire suite of benchmarks. This represents our baseline for the paper (SS).

We then compare the static fused-core scheme with the results obtained for the dynamic one for the SD-VBS benchmarks. This is followed by a closer analysis for two dynamic core fusion objectives: one that optimizes speed and another that optimizes for efficiency.

5.7.1 Creating Dynamic Core Fusion Traces

With dynamic core fusion, we have the ability to change the number of cores for each time tick (an interval of 640 blocks) during program execution. In order to explore the different performance and energy trade-off that is possible to achieve with this technique, we collect traces of execution for the whole application. We run the whole application on 1,2,4,8 and 16 fused cores and record for each time tick the cycle count. Using these 5 traces, we can then reconstruct any arbitrary dynamic execution and generate dynamic traces.

To simplify the exploration process, time ticks of the same phase will always be attributed the same number of cores. This is done to reduce the search space as on average we have 48494 ticks which would result in an average of $5^{48,494}$ different possible executions. Since the maximum number of clusters found is 6 (for SVM), we only build a maximum of $5^6 = 15625$ different dynamic execution traces. When we switch the size of the logical core (LC), we use the performance of that LC from its respective trace file and add an extra 100 cycle penalty for switching the size of a LC. With all these different dynamic core fusion traces, we can now find the optimal schemes for maximizing speed or maximizing efficiency.

5.7.2 Dynamic Core Fusion

Figure 5.13 shows the trade off between time and energy using either a static scheme fixed once at the beginning of the program or a dynamic scheme. The dotted line represents the static core fusion scheme whilst the solid line represents the Pareto Front of all the dynamic core fusion traces. The vertical line represents the amount of energy that can be saved from using a dynamic core fusion scheme that matches the same speed as the best static scheme.

Figure 5.13 demonstrates how static core fusion fails to maintain good energy efficiency as we improve speed. For example, *Disparity* (Figure 5.13a) is fastest on 16

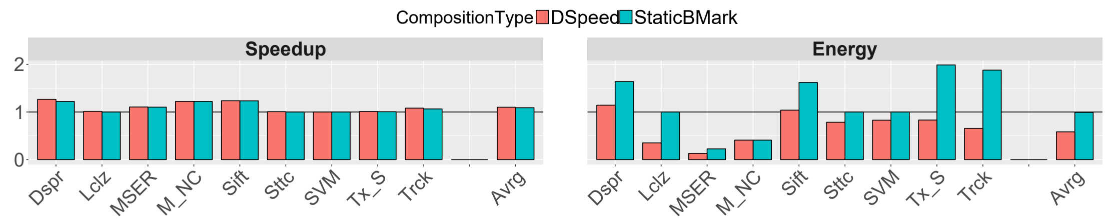

Figure 5.14: Maximizing speed for all the SD-VBS benchmarks. For Speedup, higher means better, for Energy, lower is better.

fused cores, but has an 1.63x increase in energy consumption for a 1.22x improvement in speed. When using the dynamic scheme, it is clear that energy consumption increases at a slower rate when increasing speed. In this case the number of cores is adapted to the current program phase, using just enough cores to maintain high performance without wasting energy.

5.7.3 Optimizing for Speed

In this section we define our dynamic scheme to be one that matches the same speed performance as the fastest static core fusion for the benchmark: **DSpeed**. This is equivalent to the vertical line found in Figure 5.13. This scheme enables us to maintain good performance whilst reducing energy consumption drastically.

Figure 5.14 shows the speedup of **DSpeed** and SB and the respective energy consumption. The results are normalized against the performance of SS, which is 8 cores fused. The SS core count is obtained by averaging the number of cores for each benchmark using the SB scheme. The speed performances are the same for SB and **DSpeed** as the dynamic scheme is designed to match the static speed. We can see that some benchmark perform better when using benchmark specific core fusions rather than SS. Both *Disparity* and *Sift* obtain a 1.25x speedup when using the SB scheme whilst *Tracking* benefits from a 1.10x speedup.

When looking at the Energy graph of Figure 5.14, we can clearly notice where the SS scheme fails. Benchmarks *MSER* and *Multi_NCut* feature very little improvements when using core fusion, therefore the SS will perform very poorly when it comes to energy consumption for the benchmarks. SB does not always perform well neither; as we can see, for the benchmarks *Disparity*, *Sift*, *Texture_Synthesis* the energy consumption is much higher. This is due to the fact that these benchmarks perform best on a 16-core system, however as we saw in Figure 5.12, the variation in performance always increases when fusing this many cores. The **DSpeed** scheme always performs

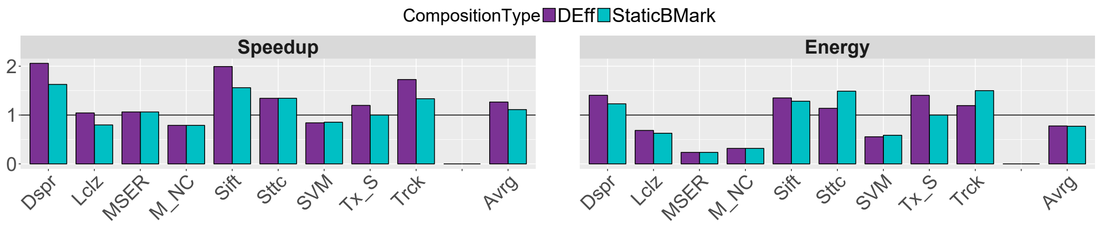

Figure 5.15: Maximizing efficiency for all the SD-VBS benchmarks. For Speedup, higher means better, for Energy, lower is better.

better than the SB scheme and can even match the SS scheme on energy consumption whilst improving speed such as in the *Sift* benchmark. For the *Localization* benchmark, the **DSpeed** matches the performance of both the SB and SS whilst reducing energy consumption by 65%.

Overall, by using **DSpeed**, we can reduce energy consumption by 42% compared to both SB and SS without impacting performance. This illustrates the greatest advantage of using a DMP since the number of fused core can be adapted continuously depending on the amount of ILP available for each phase.

5.7.4 Optimizing for Efficiency

In this section we define our dynamic scheme to maximize the efficiency metric EDD, which is defined as $Energy \times Delay \times Delay$ where Delay is the execution time. This metric attempts to optimize speed whilst remaining energy efficient; we call the scheme **DEff**. Figure 5.15 shows the speedup performance of **DEff** and SB and their respective energy consumption. The results are normalized against SS which is a fixed-composition of 4 fused cores.

Unlike the previous results in Figure 5.14, we can see that there are differences in the speedup obtained by **DEff** and SB. For benchmarks *Disparity*, *Sift* and *Tracking* the **DEff** scheme is 1.30x faster than the SB scheme and at least 1.75x faster than the SS scheme. It is important to note that this extra speedup does not incur great increases in energy consumption compared to SB: only 1.10x for *Disparity* and *Sift*. In fact, for *Tracking* **DEff** saves 20% in energy compared to SB. When comparing to SS **DEff** is 1.75x times faster for only 19% more energy for the *Tracking* benchmark.

Overall, **DEff** results in a 1.25x speed increase compared to SB and SS whilst consuming 25% less energy than SS. This shows how dynamic core fusion's flexibility allows us to get better speedups whilst not drastically increasing our energy consumption.

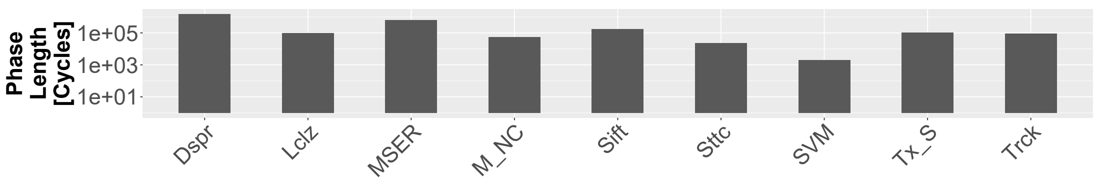

Figure 5.16: Average number of cycles without switching.

5.7.5 Reconfiguration Latency

Up until now, the paper has assumed a reconfiguration latency of 100 cycles whenever dynamic reconfiguration occurs as explained in section **??**. This section studies the impact of a larger reconfiguration overhead on energy savings. First, figure 5.16 shows the average phase length for each benchmarks when maximizing energy savings while maintaining performance (**DSpeed**). As can be seen, the majority of the benchmarks run for long period of several ten of thousands cycles before any switching occurs. Therefore, we expect that even if the reconfiguration latency would be increased to larger value (e. g. 1,000 cycles), its impact might be minimal.

Furthermore, we always have the option to reconfigure less often, in the case where a change in configuration only brings marginal reduction in energy. In such case it might be more beneficial to keep running on the slightly less optimal configuration than paying a cost for reconfiguration. Figure 5.17 illustrates perfectly this scenario, showing how energy behaves as a function of the reconfiguration overhead (averaged across benchmarks). For each latency value, we determine the best trace of reconfiguration to keep performance constant while minimizing energy (**DSpeed**). The left y-axis expresses the energy savings relative to the static scheme, while the right y-axis shows the total number of switches. The energy savings remains high up to a latency of 1,000 cycles, with a noticeable decrease in the number of switches. For latency values over 1,000 cycles, the energy savings drop considerably, with very few switching occurring. This data shows that even if the reconfiguration overhead is 1,000 cycles, average energy savings of 38% are possible compared to 42% when the overhead is 100 cycles.

5.7.6 Summary

Overall, we have seen that whether we optimize for speed or efficiency, dynamic core fusion will always lead to higher speedup or lower energy consumption than a fixed configuration. This is due to the presence of phases in applications that the dynamic scheme can exploit to reduce wasting energy in low ILP phases. We have shown that

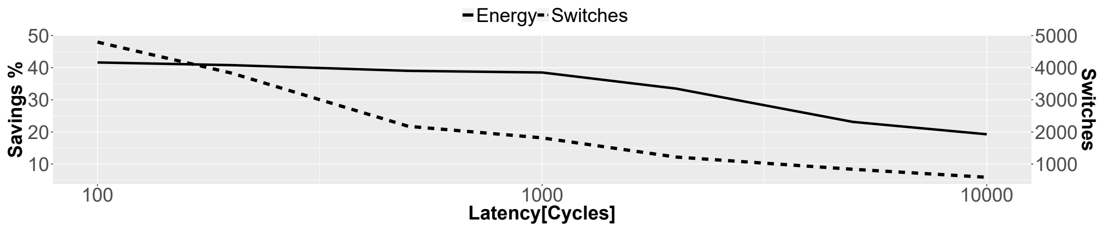

Figure 5.17: Energy savings and number of switches as a function of the switching latency in cycles.

maximizing speed can be highly energy inefficient when using a static LC and that a dynamic scheme can help reduce energy consumption by 42% on average. When optimizing for efficiency, we have shown that a dynamic scheme can help improve both speed and energy consumption, for example in the case of *Tracking*, and overall we can improve speed by 1.25x whilst saving 25% energy.

5.8 Linear Regression Model

Having shown the potential that a DMP has to offer, we now present a simple scheme that is used to exploit the large energy savings available. The main idea is to monitor at runtime some performance counters and make a decision at a regular interval on how to reconfigure the cores. For this purpose, we train a model offline using the data collected and presented earlier in the paper. Once trained, the model predicts the optimal number of cores based on the performance counters from the previous time interval and reconfiguration occurs if it is different from the current number of cores.

5.8.1 Model

We use a linear regressor which makes predictions using a simple weighted sum of the input features, which is very lightweight and easy to integrate in hardware. The model is trained offline using the traces gathered from our prior analysis for the **DSpeed** scenario which maximizes energy savings while maintaining performance. The dataset consists of a set of four input features (average block size, and percentage of integer, floating point and load operations) and the optimal number of cores for each time tick for each program. These features were chosen as they are easy to extract from the hardware. To speedup the learning process, we create a single data point per phase,

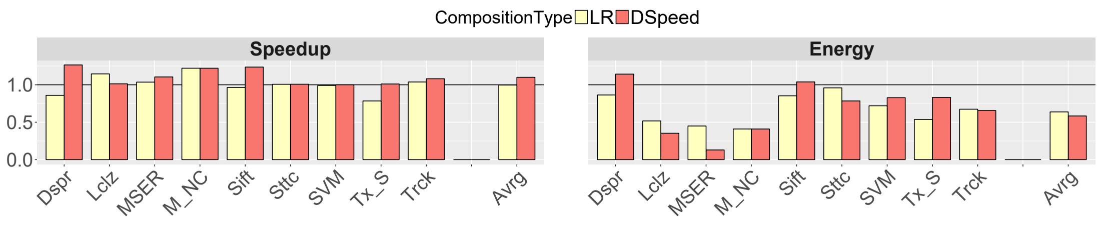

Figure 5.18: Performance results for maximizing speed for the SD-VBS benchmarks using our linear regressor (LR) model.

averaging the features of all the ticks in a phase, resulting in a total of 34 pairs of optimal core number and features.

The training consists of finding the weights that minimize the error when predicting the optimal number of cores to use across all time ticks and benchmarks. Since we have only considered core configurations which use a power of two number of cores, the linear model is built to predict the logarithm (base 2) of the number of cores. The prediction is rounded up to the nearest integer in the interval $[0,4]$. The following equation represents the trained linear model which can be used to make prediction:

$$log_2(\textbf{\#cores}) = -7.7 + 0.028 \cdot \textbf{avgBlkSze} + 0.075 \cdot \textbf{\%int\_ops} +$$
$$0.069 \cdot \textbf{\%fp\_ops} + 0.21 \cdot \textbf{\%ld\_ops}$$

For instance, if we observe at runtime an average block size of 6 instructions, and 77%, 1% and 18% of integer, floating point and load operations, respectively, then the predicted value will be 2.092. Rounded up to the nearest integer value, 2, the optimal number of cores predicted will, therefore, be 4. As can be seen, the largest weight is on the percentage of loads operations. This is due to the fact that loads can be fired independently to the Load-Store Queue. Unlike stores that depend on previous memory instructions blocks being committed, loads can be fired with less overhead. As data can be speculatively fetched, loads instructions can receive data from other cores before the data is stored, speeding up the load instruction. By increasing the core count on load heavy blocks this will improve performance more reliably due to cores being able to issue loads in parallel.

5.8.2 Results

To evaluate the performance of our model, we use leave-one-out cross-validation, a standard machine-learning methodology which tests the model using not seen during training. For instance, if we want to test the model for one program, let say *Disparity*, we train the model using the dataset from all the other programs combined. We then use the resulting trained linear model to predict the optimal core number for each time tick of the disparity program and report the performance achieved.

Figure 5.18 shows the performance in terms of speed and energy that is achieved using our linear model normalized by a fixed static configuration. The fixed configuration maximized performance across all the benchmarks using 8 cores and is the same as in the previous results presented in figure 5.14. On average, our linear regressor model is able to consume 37% less energy compared to the 8 cores fixed configuration and is able to exactly match its speed.

The performance is also compared with the best possible choice of dynamic reconfiguration, **DSpeed** which acts as an Oracle. As can be seen, the linear model is able to exploit similar energy savings to the **Dspeed** scheme in most cases. On average it reduces energy by 37%, which is within 5% of the 42% achievable by the **Dspeed** scheme. These results show that it is possible to implement a simple realistic lightweight scheme which offers large energy savings.

5.9 Related Work

5.9.0.0.1 Reconfigurable Processors ElasticCore [39] proposes a morphable core that uses dynamic voltage and frequency scaling (DVFS) and microarchitectural modifications such as instruction bandwidth and capacity. They propose a linear regressor model to determine reconfiguration, which uses more runtime information than ours, such as branch prediction and cache misses. Overall Tavana et al's architecture is 30% more energy efficient than a big.LITTLE architecture.

In [10] they also propose a similar core architecture that modifies microarchitectural features. They provide extensive analysis of SPEC 2000 benchmarks and demonstrate that machine learning and dynamic adaptation can double the energy/performance efficiency compared to a static configuration.

MorphCore [24] focuses on reconfiguring a core for thread level parallelism. It switches between out-of-order (OoO) when running single threaded applications and an in-order core optimised for simultaneous multi threading (SMT) workloads. This provides an opposite solution to our DMP: providing a large core made for ILP that can be modified to better fit TLP workloads. MorphCore outperform a 2-Way SMT OoO core by 10% whilst being 22% more efficient.

All these projects focus on uni-core modifications, and traditional CISC/RISC like architecture which differs from our work.

5.9.0.0.2 Dynamic Multicore Processors Previous work on Dynamic Multicore Processors includes CoreFusion [22] and Bahurupi [32, 33]. These architectures use a standard ISA and either fetch fixed sized instruction windows [22] or entire basic blocks [32]. Other DMPs such as TFlex [25] and E2 [37] use an hybrid-dataflow EDGE ISA [6]. In TFlex, instructions from a block are executed on different fused cores. In E2, a block is mapped to a fused core and all instructions from that block execute locally.

5.9.0.0.3 Dynamic Core Fusion In the work of Pricopi et al. [33], they show how dynamic reconfiguration is beneficial when it comes to scheduling tasks. However, they do not discuss any method of automatically deciding the optimal configuration beyond a 4 core fusion. Instead they use speedup functions determined from profile executions of applications to determine how to schedule tasks. They do not discuss what software characteristics help determine when to reconfigure the cores, or how to optimise software.

Work on using machine learning to automatically choose a composition was achieved in [28]. This work does not involve changing the core fusion dynamically during the execution of the benchmark. The machine learning model focuses on using high-level information from StreamIt's [41] language constructs.

5.9.0.0.4 Voltage Scaling Voltage scaling is another method of reducing energy consumption [31], however this approach is orthogonal to DMPs [17]. Whilst both methods adapt to programs phases, DMPs can also be used to speed up the execution of programs.

5.10 Conclusion

In this paper we have shown that whilst static core fusion already demonstrates promising results, it becomes harder to be efficient when increasing the size of logical cores. We explained theoretical limitations of static core fusion; without high branch prediction and large blocks, it under-performs. This was followed by a study of a suite of benchmarks, showing how performance varies greatly depending on the size of logical cores.

We then created two dynamic schemes: **DSpeed** that matches the speed of the fastest static core fusion and **DEff** that maximizes efficiency. Using these schemes we saw that **DSpeed** saves on average 42% energy compared to the optimal static logical core for a given benchmark. We also showed that **DEff** can improve performance by up to 1.30x and reduce energy consumption by 1.20x on some benchmarks. Finally, we developed a simple linear regression model to decide on the number of cores to fuse at runtime to optimize for performance, leading to a 37% reduction in energy while maintaining the same level of performance as a static scheme.

Bibliography

[1] J. Auerbach, D. Bacon, I. Burcea, P. Cheng, S. Fink, R. Rabbah, and S. Shukla. A compiler and runtime for heterogeneous computing. In *DAC, 2012*, pages 271–276, June 2012.

[2] L. Bauer, M. Shafique, S. Kreutz, and J. Henkel. Run-time system for an extensible embedded processor with dynamic instruction set. In *the Conference on Design, Automation and Test in Europe*, DATE '08. ACM, 2008.

[3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *ISSCC 2008. IEEE International*, pages 88–598, Feb 2008.

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

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004*, pages 777–786, New York, NY, USA, 2004. ACM.

[6] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, X. Chen, R. Desikan, S. Drolia, J. Gibson, M. S. Govindan, P. Gratz, H. Hanson, C. Kim, S. K. Kushwaha, H. Liu, R. Nagarajan, N. Ranganathan, R. Reeber, K. Sankaralingam, S. Sethumadhavan, P. Sivakumar, and A. Smith. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, July 2004.

[7] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES '09*, pages 57–66, New York, NY, USA, 2009. ACM.

[8] J. Chen, M. I. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand. A reconfigurable architecture for load-balanced rendering. In *HWWS '05*, pages 71–80, New York, NY, USA, 2005. ACM.

[9] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. *SIGPLAN Not.*, 47(4):261–272, Mar. 2012.

[10] C. Dubach, T. M. Jones, and E. V. Bonilla. Dynamic microarchitectural adaptation using machine learning. *ACM Transactions on Architecture and Code Optimization*, 10, 2013.

[11] C. Dubach, T. M. Jones, and M. F. P. O'Boyle. Exploring and predicting the effects of microarchitectural parameters and compiler optimizations on performance and energy. *ACM Transactions on Embedded Computing Systems*, 11S(1), June 2012.

[12] L. Everitt, Landau. *Cluster Analysis*. 2001.

[13] S. Eyerman and L. Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. *SIGARCH Comput. Archit. News*, 38(3):362–370, June 2010.

[14] C. Fallin, C. Wilkerson, and O. Mutlu. The heterogeneous block architecture. In *IEEE 32nd International Conference on Computer Design*, ICCD, pages 386–393, Oct 2014.

[15] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Profile-guided deployment of stream programs on multicores. LCTES '12, pages 79–88, New York, NY, USA, 2012. ACM.

[16] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. *SIGARCH Comput. Archit. News*, 30(5):291–303, Oct. 2002.

[17] M. Govindan, B. Robatmili, D. Li, B. Maher, A. Smith, S. Keckler, and D. Burger. Scaling power and performance via processor composability. *IEEE Transactions on Computers*, 63(8):2025–2038, Aug. 2014.

[18] M. Govindan, B. Robatmili, D. Li, B. Maher, A. Smith, S. W. Keckler, and D. Burger. Scaling power and performance via processor composability. *IEEE Transactions on Computers*, 63(8):2025–2038, 2014.

[19] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, and D. Burger. Multitasking workload scheduling on flexible core chip multiprocessors. *SIGARCH Comput. Archit. News*, 36(2):46–55, May 2008.

[20] U. Gupta, C. A. Patil, G. Bhat, P. Mishra, and U. Y. Ogras. Dypo: Dynamic pareto-optimal configuration selection for heterogeneous mpsocs. *ACM Trans. Embed. Comput. Syst.*, 16(5s):123:1–123:20, Sept. 2017.

[21] H. Homayoun, V. Kontorinis, A. Shayan, T. Lin, and D. M. Tullsen. Dynamically heterogeneous cores through 3d resource pooling. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12. IEEE Computer Society, 2012.

[22] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 186–197. ACM, 2007.

[23] I. Jibaja, T. Cao, S. M. Blackburn, and K. S. McKinley. Portable performance on asymmetric multicore processors. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '16, pages 24–35. ACM, 2016.

[24] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, 2012.

[25] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 381–394. IEEE Computer Society, 2007.

[26] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *MICRO '07*, pages 381–394, Washington, DC, USA, 2007. IEEE Computer Society.

[27] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, June 2008.

[28] P. J. Micolet, A. Smith, and C. Dubach. A machine learning approach to mapping streaming workloads to dynamic multicore processors. In *Proceedings of the 17th ACM Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES '16, pages 113–122. ACM, 2016.

[29] S. Mittal. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Computing Surveys*, 48(3), Feb. 2016.

[30] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett. Design and evaluation of a compiler for embedded stream programs. In *LCTES '08*, pages 131–140, New York, NY, USA, 2008. ACM.

[31] S. Pagani, A. Pathania, M. Shafique, J. J. Chen, and J. Henkel. Energy efficiency for clustered heterogeneous multicores. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1315–1330, May 2017.

[32] M. Pricopi and T. Mitra. Bahurupi: A polymorphic heterogeneous multi-core architecture. *ACM Transactions on Architecture and Code Optimization*, 8(4), Jan. 2012.

[33] M. Pricopi and T. Mitra. Task scheduling on adaptive multi-core. *IEEE Transactions on Computer*, 63(10):2590–2603, Oct. 2014.

[34] A. Putnam, A. Smith, and D. Burger. Dynamic vectorization in the e2 dynamic multicore architecture. *SIGARCH Comput. Archit. News*, 38(4):27–32, Jan. 2011.

[35] R. Rodrigues, I. Koren, and S. Kundu. Performance and power benefits of sharing execution units between a high performance core and a low power core. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 204–209, Jan 2014.

[36] P. Santos, G. Nazar, F. Anjam, S. Wong, D. Matos, and L. Carro. A fully dynamic reconfigurable noc-based mpsoc: The advantages of total reconfiguration. In *HiPEAC '13*, Berlin, Germany, January 2013.

[37] A. Smith and A. Bakhoda. Microsoft research development kit for edge architectures. https://www.microsoft.com/en-us/research/project/e2/, 2017. Accessed: 2017-07-14.

[38] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. Burrill. Compiling for edge architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, 2006.

[39] M. K. Tavana, M. H. Hajkazemi, D. Pathak, I. Savidis, and H. Homayoun. Elasticcore: Enabling dynamic heterogeneity with joint core and voltage/frequency scaling. In *ACM/EDAC/IEEE Design Automation Conference*, DAC '15, pages 1–6, June 2015.

[40] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT '10*, pages 365–376, New York, NY, USA, 2010. ACM.

[41] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376. ACM, 2010.

[42] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *CC*, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[43] E. Tomusk, C. Dubach, and M. O'Boyle. Four metrics to evaluate heterogeneous multicores. *ACM Transactions on Architecture and Code Optimization*, 12(4), Nov. 2015.

[44] A. Venkat, S. Shamasunder, H. Shacham, and D. M. Tullsen. Hipstr: Heterogeneous-isa program state relocation. *SIGPLAN Not.*, 51(4):727–741, Mar. 2016.

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

[46] A. Venkat and D. M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 121–132, 2014.

[47] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. Sd-vbs: The san diego vision benchmark suite. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 55–64. IEEE Computer Society, 2009.

[48] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, 2003. AAI3121741.

[49] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, Sep 1997.

[50] Z. Wang and M. F. P. O'boyle. Using machine learning to partition streaming programs. *ACM Trans. Archit. Code Optim.*, 10(3):20:1–20:25, Sept. 2008.

[51] Y. Watanabe, J. D. Davis, and D. A. Wood. Widget: Wisconsin decoupled grid execution tiles. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 2–13. ACM, 2010.

[52] P. M. Wells, K. Chakraborty, and G. S. Sohi. Dynamic heterogeneity and the need for multicore virtualization. *SIGOPS Oper. Syst. Rev.*, 43(2):5–14, Apr. 2009.

[53] Y. Zhou and D. Wentzlaff. The sharing architecture: Sub-core configurability for iaas clouds. *SIGPLAN Not.*, 49(4):559–574, Feb. 2014.