Exploiting workload similarities for efficient scheduling in diverse asymmetric chip multiprocessing

Dani Shaket

Exploiting workload similarities for efficient scheduling in diverse asymmetric chip multiprocessing

Research Thesis

In Partial Fulfillment of The Requirements for the Degree of Master of Science in Electrical Engineering

Dani Shaket

Submitted to the Senate of the Technion - Israel Institute of Technology
Adar 5778 Haifa February 2018

The Research Thesis Was Done Under The Supervision of Professor Idit Keidar in the Faculty of Electrical Engineering at the Technion.

Acknowledgments

First, I would like to thank my advisor, Prof, Idit Keidar. Thank you for timely transforming me from an engineer to a researcher, for teaching me to differentiate between the wheat and the chaff, and for supporting me technically and mentally during this long, long journey. What I learned from you greatly exceeds any course, paper or research tasks.

I would also want to thank Uri Weiser, Yoav Etsion, Mattan Erez and Israel Koren for their contribution to this research, for taking the time to analyze raw, immature work, for sharing ideas, and for providing highly valuable constructive criticism.

I thank two teachers, Yoram Moses and Shmuel On, who did not have a direct impact on this work but their excellent courses "Knowledge and Games in Distributed Systems" and "Advanced Integer Programming" (respectively), enriched my toolbox and expanded my vision.

I'd like to thanks the wonderful Idit's research group team members: Kfir Lev-Ari, Sasha Spiegelman, Naama Kraus, Noam Shalev, Itay Tsabary, Hagar Porat, Alon Berger, Ittay Eyal, Stacy Patterson, Meni Orenbach, and Oved Itzchak. Thank you for expanding my horizon in so many exciting, innovative, creative and smart ways.

A special thanks to Alex Shraer for introducing me to Idit and her research group, for pushing me to do the masters, for expressing a definitive belief that I can do it, and for timely checking that I am.

Finally, I would like to thank my family. I thank my parents which never cease to support me and empower me. I thank my amazing wife Ravit, for understanding me and my studential needs, for supporting me all the way, and for working very hard to help me achieve my goals. I also thank my kids, Hila and Eyal, which in turns, allowed me to complete writing my thesis at nights, for being my muse, and for giving me powers to keep going.

Table of Contents

List of Figures

List of Tables

Abstract

Heterogeneous chip-multiprocessor (HCMP) incorporates different types of cores on a single die. Each core-type operates in a different way, provides different capabilities, and shows different performance attributes. Asymmetric chip-multiprocessor (ACMP) is an HCMP in which all core-types comply to the same instruction set architecture (ISA). In some ACMP design schemes cores share functional components, sometimes even mutually exclude each other.

ACMPs can increase power-efficiency by executing diverse workloads on suitable cores. Matching workloads to cores with ever-changing workload behaviors is a challenge. Practical solutions so far assume ordering of core strength from simple "small" energy-efficient cores to complex "big" ones which provide high performance for compute-heavy workloads. This approach allows relatively straightforward workload-to-core assignment.

While ARM-based big.LITTLE architectures already capture a huge market share, higher single-ISA diversity is avoided. In this thesis, we argue that restricting the cores design space to linear ordering limits the potential of ACMP. We examine the notion of using asymmetric specialized chip-multiprocessors (ASCMPs) to achieve better power-efficiency.

Managing the execution of a multi-core system with high core and workloads diversity is a complex task. In order to perform this task efficiently, there is a need to predict the power-performance attributes of a workload on the different cores.

For this purpose, we propose a centralized history-based lookup mechanism that shares execution information among cores. During workload execution, each core produces micro-architecture-independent signatures and corresponding performance attributes. The signatures are generated using workload behavior, and not from the impact on the core. This approach guarantees that the workload signature is identical regardless of the core it runs on.

The signature is created for performance characterization per core, where workloads that share the same signature are expected to have similar performance characteristics when executed on the same core. Thus, the execution management system exploits workload similarities to group workloads by behaviors and use execution information as a performance predictor.

The micro-architecture-independent signature structure should be small enough (bitwise) to enable practical mechanism, use execution characteristics which are feasible to extract in addition to fulfilling two opposing requirements:

1. Accuracy - The signature should be flexible enough to express different workload behaviors and provide a high correlation between the signature and the performance attribute on each core.

2. Aggregability - The signature space should be small enough to enable grouping workloads into categories.

In this thesis, we address the challenge of finding the sweet spot between these two opposing requirements and create a signature structure which is feasible to produce and practical to use.

We propose a six-core platform and evaluate it using GEM5 and McPAT. Our proposed system with perfect assignment shows up to 37% power-performance gain with 13% on the average for the SPEC2006 benchmark suite in comparison to big.LITTLE based ACMP. The same system with a trained history-based predictor shows up to 33.8% power-performance gain with 22.2% on average for a subset of six benchmarks selected from SPEC2006 benchmarks.

1 Introduction

Where labor is not differentiated and distributed like that, where everyone is a jack-of-all-trades, professions remain at an utterly primitive level.

Immanuel Kant, Groundwork of the Metaphysics of Morals, 1785

Asymmetric Chip Multiprocessors (ACMPs) are composed of different core types that comply with the same *Instruction Set Architecture (ISA)*. Sometimes different cores may be simultaneously active, while in other cases they are mutually exclusive. Either way, ACMPs increase power-efficiency by executing diverse workloads on suitable cores.

Matching workloads to cores with changing behaviors is a challenge. Practical solutions assume ordering of core strength from simple "little" cores, which are power-efficient, to complex "big" ones which perform compute-heavy workload efficiently. This approach allows relatively straightforward workload-to-core assignment.

In this thesis, we argue that restricting the cores design space to linear ordering limits the full potential of ACMPs. We examine the notion of using *Asymmetric Specialized CMP (ASCMP)* to achieve higher power-efficiency than state-of-the-art linear-ACMP platforms provide.

To enable workload-to-core matching, we propose a history-based lookup mechanism that shares execution information among cores. During workload execution, each core produces micro-architecture-independent signatures and corresponding performance attributes. Workloads that share the same signature are expected to have similar performance characteristics when executed on the same core. Thus, the execution management system exploits workload similarities in order to group workloads by behaviors and uses execution information as a predictor for performance characteristics.

The rest of this chapter is organized as follows: Section 1.1 gives a general overview of ACMP. In Section 1.2, we summarize the limitations of linear ACMP and discuss the benefits of using ACMPs. Section 1.3 introduces the basics of workload characterization, the metrics we use for power-performance and the concept of micro-architecture-independent signature. Section 1.4 explains our testing methodology and tools and Section 1.5 enumerates the contributions of this thesis.

1.1 Asymmetric CMP

A *Heterogeneous Chip Multiprocessor (HCMP)* is a system that incorporates different types of cores on a single die. Each core type operates differently, has different capabilities, and possibly a different ISA. A *Graphic Processing Unit (GPU)*, for example, is designed primarily for providing efficient graphics rendering capabilities. *Digital Signal Processors (DSPs)* are optimized for real-time data streaming and numeric calculations. Among other things, a general purpose CPU can do both graphics rendering and signal processing, but not as efficiently as a GPU or a DSP.

While explicitly executing workloads on different cores (e.g., CPU, GPU, DSP) is common practice, it has major disadvantages. First, it imposes a significant overhead on the development process. It requires the software engineers as well as frameworks, ecosystems, compilers, and applications to be familiar with different tool-chains, ISAs, micro-architectures, concepts of operations and APIs. Second, it usually lacks the flexibility to dynamically choose which core is used in case of input and run-time variations (e.g., resource availability, power profile). Third, it makes the design cumbersome and expensive by requiring core-specific resources like on-chip memories (e.g., GPUs). While these downsides motivated the industry and academia to create powerful offloading mechanisms (e.g., OpenCL, OpenACC, OpenHMPP), these solutions usually require knowing the target architectures, write the code in a specific way, handle the scheduling explicitly, pre-compile binaries for multiple architectures, which increases binary size and prevents on-the-fly migrations, and just-in-time compiling which can increase run-time.

ACMP is an HCMP in which all cores comply to the same ISA. Previous work [1][34] as well as state-of-the-art commercial products demonstrate that asymmetry, namely using different core types, yields significant improvements in power-efficiency of small-scale chip multiprocessors. ACMPs also include platforms in which different core types are implemented using shared resources or pipeline components [28, 12], possibly mutually excluding each-other at runtime [20].

By complying to the same ISA, ACMPs eliminate the downsides of HCMPs. The workload-to-core assignment is done (in most cases) automatically by the operating system or by the hardware.

In Chapter 3, we discuss workloads' behavioral properties, define our core model and explain the metrics we use to estimate execution efficiency.

1.2 A case for specialized cores

The main downside of ACMP is that the diversity of the cores is limited by the ability to predict performance. The implemented cores typically reside on a single line – from little "weak" to big "strong" cores.

Big cores have complex super-scalar *Out-Of-Order (O3)* execution mechanism, aiming to exploit *Instruction-Level-Parallelism (ILP)* and high data availability (or low data dependencies). Typically, big cores are capable of running at a high frequency. These cores usually provide high single thread performance for compute-heavy workloads.

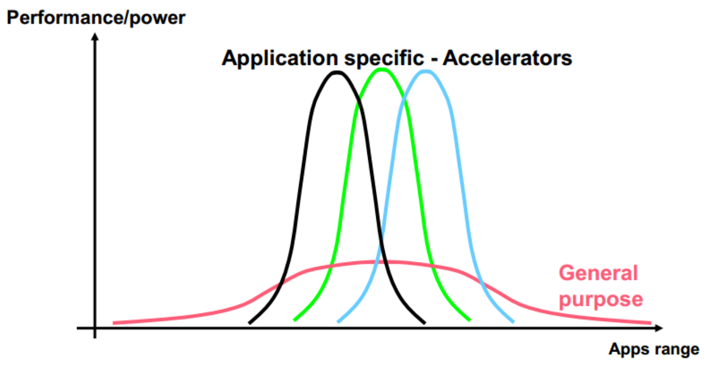

Figure 1.1: Application specific vs. general purpose design[35]

Little cores have simple in-order execution pipes and typically run at lower frequencies. These cores are more power-efficient than big cores, but execute slower for compute-heavy workloads. In cases where big cores demonstrate poor pipeline utilization (e.g., multi-threaded workloads with frequent synchronization, i/o, or memory dependencies), the execution times of the little core and the big core might be similar, but the execution might be significantly more power-efficient for the little core, and thus more energy-efficient.

The motivation for creating cores with higher diversity, as illustrated in Figure 1.1, stems from the fact that the more (application) specific the design is, the better performance, power-efficiency, and area-cost it has.

While ARM-based big.LITTLE [24] architectures already capture a market share (e.g., Apple A10, Qualcomm Krait [26], Samsung Exynos, MediaTek CorePilot [22], Huawei Kirin)[1], wider diversity is still avoided.

In these products, as well as in recent research, a linear ordering of core strength is assumed, limiting the benefit that can be achieved. This is partially due to a pessimistic estimation of the software efforts, complexity, and runtime overheads that may be required to manage more diverse architectures efficiently. Furthermore, it was not demonstrated that practical systems can benefit from more diverse asymmetry.

In Chapter 4, we explore the micro-architectural design space and show the potential benefit of more diverse asymmetry in ACMP.

1.3 Predicting performance via micro-architecture-independent signatures

Managing the execution of a multi-core system with core and workload diversity is a complex task. To perform this task efficiently, there is a need for an efficient mechanism that predicts power-performance attributes of a given workload on the different cores.

[1] The latest MediaTek Helio X20 takes this further by providing a tri-cluster small-medium-big architecture.

For that purpose, we use micro-architecture-independent signatures, which represent a workload by collecting partial information on how the workload uses the ISA. Using this approach, the workload generates the same signature regardless of the core it executes on. We avoid machine-dependent information, such as cache misses, TLB misses, or delays, which would result in different signatures for the same workload. We then associate each workload signature with execution characteristics such as power and performance for each particular core.

Thus, we make a clear distinction between workload characterization and execution characterization. When characterizing workload, we examine its inherent behavior: what instructions it runs, in which order, how it accesses memory, etc.

Execution characterization is the impact of executing a workload on a particular core: How much energy it consumes, how much time it takes, the utilization of the core, cache/TLB misses, etc. It provides us with power-performance attributes the system needs in order to make educated workload-to-core assignments.

To enable performance predictions from the history of other cores using micro-architecture-independent signatures, we need it to be structured in a way that fulfills two opposing requirements:

1. Accuracy - The signature should be flexible enough to express different workload behaviors and provide a high correlation between the signature and the corresponding workload's performance attributes on each core.

2. Aggregability - The signature should be small enough to enable grouping of different workloads under the same signature.

On the one extreme, if the signature is too accurate, it creates a unique signature for each workload. Therefore it is useless for predicting the performance of a workload on a core it did not run on. Moreover, it requires allocating and managing large memory space. At the other end, if the signature is too small and aggregates workloads with different behaviors and performance attributes into the same group, it also cannot be used for predicting performance. The challenge is finding the sweet spot between these two opposing requirements.

In Chapter 5, we find a micro-architecture-independent signature structure that enables efficient aggregation of different workloads into groups and provides a high correlation between signatures and performance attributes on each core. We then use this structure to create a performance predictor.

1.4 Simulator and tools

We use GEM5 [6] to simulate the execution of workloads on our cores. We use ARM ISA with an O3 processor and GEM5's classic memory system. We altered the simulator to provide all the information that might be useful to produce micro-architecture-independent signatures as well as *Instructions per Second(IPS)*, instruction types histograms and other information required for calculating power. Using this simulator, we execute programs from the SPEC2006 benchmark suite [9], which provides compute-intensive applications from diverse fields. Since programs usually demonstrate different behaviors during execution[2], we generate the information per window of execution. For power and area estimation we use McPAT [19]. In addition, we have implemented several Python scripts that aggregate data from GEM5 and McPAT, and produce statistics and data visualization.

1.5 Summary of contributions

This thesis makes the following contributions:

- We show that having a non-linear set of asymmetric cores might increase power-efficiency.
- We show that micro-architecture-independent signatures can be used to identify workload behavior in a way that correlates well with power-performance.
- We propose a mechanism for efficient performance prediction based on a signature lookup table.

[2] Sometimes referred to as program-phases or thread-phases.

2 Background and related work

Today it is impossible to estimate performance: you have to measure it. Programming has become an empirical science.

Joshua Bloch, Google Inc., Performance Anxiety: Performance analysis in the new millennium

The case for ASCMP. Since the rise of *general purpose computing on GPUs (GPGPU)* in the mid-2000's, the case for specialized cores with different ISA was repeatedly made, increasingly turning into HCMPs that also include DSPs and FPGAs.

Kumar et al. [16] were the first to demonstrate the potential of improving performance using ACMP. They analyzed the execution of several benchmarks on four different alpha-based cores scaled from little core to big core. Since then, numerous studies have been conducted on various aspects of linear ACMPs [23, 17, 10, 2], most of them relying on Pollack's rule [25] by assuming that $Performance \propto \sqrt{die\ area}$.

A variety of ACMP design schemes was suggested: Ipek et al. [12] presented *core fusion*, a reconfigurable CMP where several little cores can dynamically morph into a larger core to accommodate workload diversity. Lukefahr et al. [20] proposed *Composite Cores*, a big.LITTLE-like architecture that mitigates workload migration penalty by sharing context-related resources among the cores, thus allowing only one to be active at any time. Rodrigues et al. [28] suggested a *Dynamic Morphing Core (DMC)*, which enables reconfiguring cores depending on computational demands. Their baseline configurations included two cores, one with strong integer units and one with strong floating point capabilities. Several works [36, 37, 21, 27] discussed solutions for accommodating unpredictable core diversity due to manufacturing variants and errors.

Besides from DMC, which examined two non-linear asymmetric cores, we are not aware of any previous work that aimed specifically to show the benefit of single-ISA specialized cores beyond linear diversity. We make this case by implementing ASCMP, executing benchmarks, and comparing the results to a big.LITTLE platform. Our different core types can be incorporated into a single die as distinct cores, may share resources like DMC, or execute in a mutually exclusive fashion like Composite Cores.

Workload characterization and micro-architecture-independent signatures. Sherwood et al. [32] analyzed behavioral changes in programs and demonstrated how different program phases result in different power-performance. They described a mechanism that identifies a phase based on the program counter (PC). While their approach works well in the context of one program, our signature-based mechanism identifies behavioral similarities across multiple workloads.

Hoste and Eeckhout [11] presented a comprehensive study of micro-architecture-independent workload characterization including possible parameters and calculation mechanisms. In addition, they compared their method to micro-architecture-dependent methods and showed interesting insights regarding workloads characterization and classification. In our work, we share the notion of characterizing workloads using micro-architecture-independent parameters. We reduce the problem from performance reasoning to identifying groups of workloads with similar behaviors and so our signatures are much more succinct. Also, we focus on parameters and extraction methods that are feasible to achieve at runtime whereas they extract as much information as they can to thoroughly analyze the behaviors and corresponding performances.

Performance prediction in asymmetric systems. Kumar et al. [16] proposed a sampling-based scheduling scheme using four core types. Every so often, the system initiates a sampling phase, in which the scheduler permutes the assignments of threads to cores. During this period, performance statistics are gathered. These statistics are then used to create a new assignment, which is then employed during a longer phase of execution.

Another sampling-based approach argues that greedy algorithms can significantly reduce the search space. Becchi and Crowley [3] presented a greedy algorithm with individual sampling and steady phases for each thread. They used two core types (alpha-based EV5 and EV6). Whenever significant performance reduction is detected, a thread initiates a local sampling phase in which it migrates among cores, each of which measures its performance. The thread then migrates to the core that provided the highest performance.

Winter and Albonesi [36], and later on Winter et al. [37], presented comprehensive studies of sampling-based scheduling algorithms and their tradeoffs, with regard to many core types for both predictable and unpredictable core diversity, which is a result of manufacturing variants and errors.

While sampling-based solutions are fast and accurate for a small number of cores (2 - 4), we aim to handle many cores and core types. Our workload-to-core mapping scheme is based on exploiting workloads' behavioral similarities, thus eliminating the need for exhaustive sampling of all threads on all cores.

Adaptive schedulers based on analytic prediction schemes presented in [15, 1, 33, 34] showed that hardware-assisted prediction can provide a good compromise between accuracy, adaptation, and scalability. In these works, scheduling is initially arbitrary. During execution, the core collects micro-architecture-dependent information. Performance prediction is calculated locally, using an analytic model. This approach assumes that each core also includes logic that enables selecting the appropriate core type for a given workload behavior. Furthermore, since prediction is done locally, the logical block that performs the prediction is not aware of the dynamic states of the other cores. In contrast, our work does take into account the dynamic state of the cores and does not require cores to predict each other's performance. Moreover, our prediction mechanism does not assume any relation between the cores' micro-architectures; therefore, we can support more diverse core types.

Shelepov and Federova [29] and Shelepov et al. [30] proposed profiling threads offline for generating application signatures that represent thread behavior. An analytic model uses these signatures to predict the different cores' runtime performance. This scheme enables workload-to-core mapping with very little overhead. While this solution scales very well, it does not apply to applications with high runtime variability due to input variation or behavioral changes. Moreover, it is based on an analytic model that assumes isolated execution. In our work, we adapt to both program phase changes and core state changes dynamically. We do share a similar notion of micro-architecture-independent signatures, but in our scheme, it is used for behavior identification and not as a base for analytic performance prediction.

3 Model

Two elements are needed to form a truth - a fact and an abstraction.

Remy de Gourmont

In this chapter, we explain the model we use throughout this thesis. The remainder of this chapter is as follows. In Section 3.1, we explain the properties we use to describe workload behavior. Next, in Section 3.2, we define the core model we use. Since the design space of cores is huge, we limit ourselves to several simple, configurable parameters and assume the general structure of the cores remains the same. In Section 3.3 we define and explain our efficiency metrics.

3.1 Workload properties

In this section, we explain the three main properties we use to classify workload behavior. These properties are inherent characteristics of the workload, defined only by the executed instructions and input.

3.1.1 Instruction level parallelism

Instruction Level Parallelism(ILP) is a widely used concept that describes the possibility of executing instruction belonging to the same sequential workload in parallel. Given an ideal system, with an infinite number of infinitesimally fast execution units and pipeline components, the number of instructions that can be executed in parallel defines the ILP of the workload.

Ideally, ILP is defined only by the program and not by its impact on the micro-architecture. In practice, quantifying ILP is an elusive task. There are some existing schemes for measuring ILP. Hoste and Eeckhout [11] define the level of ILP by instructions-per-cycle achievable for an idealized O3 processor (with perfect caches and branch predictor) for window sizes of 32, 64, 128, and 256 in-flight instructions. This simulation-based method requires a lot of computation. Thus it is too cumbersome to perform at runtime. Xu et al. [38], detect if ILP is high or low by measuring effective instructions-per-cycle, which uses micro-architecture-dependent parameters.

Instructions can be executed in parallel if they are not data-dependent. If an instruction A uses data or a register which is created by instruction B, instruction B must be executed before instruction A; thus we say that A is dependent on B. This is, of course, a transitive relation. If A is dependent on B and B is dependent on C, then A is dependent on C. The dependency DAG encapsulates the data dependencies of the program execution. Long paths in the DAG indicate low ILP and vice-versa. In our work, we build such a DAG during the execution in order to estimate the level of ILP.

3.1.2 Memory usage pattern

Memory usage pattern has a huge effect on performance. Access to memory external to the core is slow, contention prone (bus, controller), and expensive energy-wise. Thus, cache utilization is crucial to system efficiency.

A core can execute the same instructions with different memory addresses (or the same addresses in a different order) and demonstrate extremely different performance. The frequency, repetition, pattern, and locality of the memory accesses determine the benefit that can be achieved by caches. A common micro-architecture-dependent performance measure is the miss/hit ratio of the cache. A high miss ratio can imply that a bigger cache is needed. It can also indicate that the program only reads new memory or that the cache uses a bad placement strategy.

A widely used technique to estimate the required cache size is calculating the *memory reuse distance (MRD)* [5, 4]. MRD measures the distances (in instructions) between consecutive accesses to the same memory reference. A low MRD indicates high locality. Since accesses to new memory (in cache line granularity) always cause a (compulsory) cache miss, these accesses are counted separately.

3.1.3 Instruction types histogram

Instructions Type Vector (ITV) [14] is a histogram of instruction types executed by the program in a certain window. In this context, we classify the different ARM instructions into Load (LD), Store (ST), ALU (integer arithmetic), VFP (vector/floating-point), and control. ITV's implication on suitable micro-architectures is (almost) immediate. For example, ITV with a high number of ALU operations requires faster or multiple ALU units (depending on its ILP). ITV which shows a large number of memory operations will probably prefer micro-architectures with fast memory units, a big store/load queue, or a big cache, depending on its MRD and ILP.

3.2 CPU model

In this section, we explain the model of the core used in our experiments and the design space derived from this model. We use GEM5's O3 CPU model, which is loosely based on Alpha 21264 [18, 13].

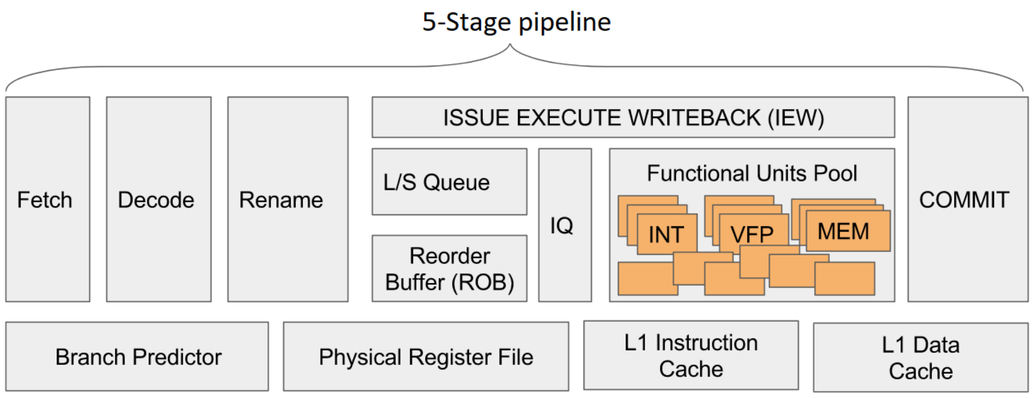

Figure 3.1: Core model.
Each functional block has configurable parameters.

Figure 3.1 shows the model of the core. Each block represents a configurable functionality or resource. Obviously, the core model can be more detailed or abstracted. This level of abstraction corresponds with the simulator we use and some of its configurable parameters.

There are five stages in the pipeline: *fetch*, *decode*, *rename*, *issue-execute-writeback(IEW)*, and *commit*. The O3 model allows instructions to enter the IEW phase in a different order than that in which they are fetched. This allows effectively higher utilization of the functional units by executing independent instructions (ILP).

Each stage has a configurable *width* which defines the number of instructions it is capable of handling in parallel. The *O3 window* is the number of instructions the pipe can hold in the *reorder buffer (ROB)* and the *Load/Store (LS)* Queue. The utilization of the ROB depends on the availability of physical registers (i.e., physical register file size) and the workload's level of ILP. The width of the different stages in the pipe, the sizes of the different queues (load/store/instruction), and the number and speed of the different functional units – together define the effective instruction execution frequency with regard to the executed workload.

The following summarizes the various configurable resources and parameters:

1. Stage width – the number of instructions the stage can handle in parallel.
2. *Branch predictor (BP)* – assists with pre-fetching instructions in case of a branch. Both the size and type of the branch predictor can be selected.
3. Number of entries in ROB.
4. Number of entries in the LS Queue. LS Queue holds the Load/Store instructions that have reached the IEW stage. This number determines the O3 window for load/store instructions.

5. Number of entries in *instruction queue (IQ)*. IQ stores the issued instructions. Instructions reaching this stage will be dispatched upon required functional unit availability.

6. Functional units – Each functional unit is responsible for a different operation. The standard parameters for configuring these are latency and throughput per operation. Each core must have at least one of each type, to comply with the ISA. Adding more than one of each type enable executing more instructions of the same kind in parallel. The units we consider are:

   SimpleInt – Simple integer operations (such as add/sub).

   ComplexInt – Complex integer operations (e.g., div/mult).

   FpSimd – Floating point operation and vector-operations.

   Load – Loading data from memory or cache into register/s.

   Store – Storing data to memory.

7. *Level 1 (L1)* instruction cache size.

8. L1 data cache size.

9. Physical Register file size – number of actual hardware registers that can be used.

3.3 Metrics

We measure execution efficiency using two basic elements. E denotes the energy, in *Joules*, consumed by the core to execute the workload. D denotes the delay, or time, usually in seconds, it takes to complete the execution. Efficiency metrics combine both. For example, ED (energy delay product) gives the same weight for energy efficiency and speed. A higher value of ED means lower efficiency. In our work, we use the common ED^2 efficiency metric – energy delay squared product, which puts more emphasis on the speed of execution.

To get E and D we measure the following core performance counters:

1. *Instructions per Second (IPS)* - measures how many instructions reach the commit phase (i.e., excluding squashed speculative instructions) per cycle multiplied by the frequency (cycles per second).

2. Power (P) – measures the energy (Joule) per time(seconds) that the core consumes.

We get the E consumed by the workload execution by multiplying P by D.

4 The case for ASCMP

In an increasingly divided world, we need a common language.

Prabhu Guptara

General purpose cores, as the name suggests, are designed to do everything. Targeting a core's design specifically to handle a particular class of workloads increases its efficiency. In other words, the narrower the target application is, the more specific the core design can be, and thus more efficient. Workloads that run on general-purpose cores can be very different from each other. They differ in the type and number of instructions they run, their order, predictability, memory access patterns and more.

In principle, each workload can have an optimal core for running it, meaning that in whatever efficiency measure we choose, this core will provide the most efficient execution. Clearly, it is not feasible to design a core for each workload. The workloads should instead be classified and grouped to enable creating a feasible number of cores.

A common conception is that the most important property of a workload is how compute-heavy versus how memory-heavy it is. A compute-heavy workload performs a lot of arithmetic operations while accessing relatively small amounts of input data. On the other end of this spectrum are memory-heavy workloads, which access a lot of input data relative to the number of arithmetic calculations they execute. The latter will not be hampered by running on a small core with less functional/arithmetic units, a simpler pipe, lower frequency (due to i/o latencies), and a smaller cache.

This conception leads to a linear design space of micro-architectures. Ranging from small energy-efficient cores to big energy-consuming ones. Nevertheless, there are different kinds of computations, and not all of them should be treated the same way. Ferdman et al. [8] exemplified this point as they demonstrated how powerful modern servers are underutilized and energy-wasting when running emerging scale-out workloads.

In this chapter, we show that using an ASCMP can have significant positive impact on efficiency. Note that in this chapter we ignore the workload-to-core matching problem. We will get back to this problem in Chapter 5.

The remainder of this chapter is as follows. In Section 4.1, we explain the workload properties we use and how we obtain them. In Section 4.2, we explain our core selection. In Section 4.3, we demonstrate the benefit that can be achieved using ASCMP by statically assigning benchmarks to suitable cores. In Section 4.4, we further allow on-the-fly zero-cost migrations to show the benefit that can be achieved for the tested benchmarks on our selected cores.

4.1 *Profiling workloads*

We examine inherent workload behavior. Thus, we can not rely on any information besides the instructions that compose the workload. Moreover, we only use instructions that reach the commit phase without being squashed due to misspeculation. We have tweaked GEM5 [6] to enable several capabilities:

1. Obtaining ITV – Collecting executed instructions per type.

2. Obtaining MRD – Measures data locality of the executed code. we use the average and standard deviation of memory reference access rates and the number of single accesses to memory locations. Figure 4.1 shows the mechanism we implemented for extracting these

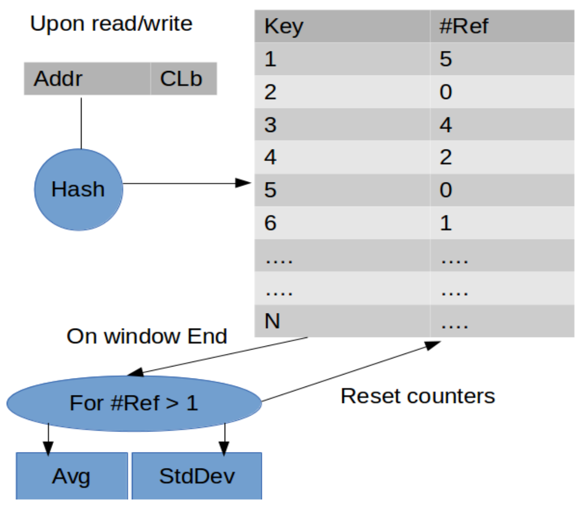

Figure 4.1: MRD extraction mechanism.

parameters. During a window of execution, each memory reference (read or write operation) updates its corresponding cell in the hash-table. The average and standard deviation are calculated for all table elements larger than 1 in order to detect recurring accesses. The number of single references is counted separately. We ignore address bits from the same cache line (CLb in Figure 4.1), thus, storing memory references in cache-line granularity. Although the cache line size is a part of the micro-architecture, a reasonable design choice

```
**Constants:** N – window size, M – Number of architectural registers
**Input:** inst – list of all instructions in a window
**Result:** Average instruction dependency chain
$R \leftarrow \{0xM\}$    // keeps dependency chain length per register
$G \leftarrow \{0xN\}$    // keeps dependency chain per instruction
for $i$ to $N$ do
    $longest = max(R[s] | s \in inst.src \cup \{0\})$
    for $d$ in $inst[i].dst$ do
        $R[d] = longest + 1$
    end
    $G[i] = longest$
end
**Output:** Average of G
```

Algorithm 1: Calcuating AIDC

for ACMP is using the same cache line size for all cores[1]. Since access to different cache lines is what determines memory usage and performance, this optimization allows major reduction of the hash table size without losing valuable information.

3. Average instruction dependency chain (AIDC) length –

   For the purpose of calculating AIDC length, we refer to instructions as a tuple of (*source (src)* registers, *destination (dst)* registers). The registers are the architectural registers defined by ISA. We measure the AIDC for blocks of 256 instructions. We do so by keeping track of the dependency chain length for each architectural register. The dependency chain length for each instruction is the maximum dependency length of src registers. Algorithm 1 shows our implementation in pseudo-code. Array R represents the dependency chain length for each one of the architectural registers. At the beginning of the window, R is initialized to all zeros. For each instruction, the algorithm finds the src register with the longest dependency chain, it then updates R at the dst register with the length of that dependency chain incremented by 1. If there is no source register (e.g., constant assignment or load instruction), R[dst] is updated to 1. The dependency chain length in R at index dst is also stored in G at index i, representing the length of dependency chain for instruction i. At the end of the execution window, the algorithm returns the average of G, which is our AIDC value.

We have compiled the SPEC2006 benchmarks using ARMv7 cross compiler with full optimizations, auto-vectorize compilation flags, and full NEON support.

We first ran the benchmarks using standard core model for 3.5 billion instruction. We have collected our ITV, MRD, and AIDC of each one of the 22 benchmarks. The results are shown in 4.1.

[1] See "A tale of impossible bug: big.LITTLE and caching", https://news.ycombinator.com/item?id=12481700.

Table 4.1: Benchmark behaviors

	LD%	ST%	ITV ALU%	VFP%	CTRL%	MRD AVG	Single Ref	AIDC AVG
bzip2	22	6	58	0	12	45.69	20	3.58
gcc	15	3	61	1	18	41.91	7	3.22
mcf	22	12	49	0	14	61.27	6	3.22
hmmer	33	18	42	1	3	62.77	1	2.14
sjeng	22	5	57	0	14	50.35	12	2.60
libquantum	4	3	65	8	17	18.80	0	3.80
h264ref	38	10	46	0	4	74.59	7	3.05
gromacs	21	9	53	0	15	46.14	17	2.31
omnetpp	23	9	50	2	13	26.57	11	4.63
astar	36	8	46	0	9	60.04	13	3.31
bwaves	31	5	36	20	6	60.58	0	1.72
milc	31	11	11	42	2	35.83	0	1.73
zeusmp	11	7	26	52	2	35.67	5	2.15
cactusADM	37	19	6	36	0	20.43	0	1.08
leslie3d	13	5	32	39	9	96.89	0	1.82
namd	19	4	35	32	6	33.47	0	2.36
dealII	20	2	53	8	14	69.04	2	3.39
soplex	22	7	55	1	11	16.27	11	4.01
calculix	14	7	50	13	13	51.91	2	2.89
GemsFDTD	16	5	39	28	11	53.97	0	2.77
tonto	27	7	47	9	8	40.62	4	2.41
lbm	24	11	14	49	1	33.52	0	1.00

4.2 Selecting cores

Each member of the ITV affects the performance requirements of the corresponding functional unit. The MRD AVG has a direct effect on the cache size; the AIDC has a direct effect on the width and O3 window of the suggested core.

Using GEM5, McPAT, and a self-implemented analysis tool, we have explored several different core configurations. Table 4.2 shows the configurations we have selected.

Small is expected to handle i/o and memory heavy workloads efficiently. It has small caches, small branch predictor, and a minimum number of functional units. *Big* is a balanced core which performs all tasks with reasonable efficiency. *Wide/Int* has a wide pipe, big data cache and the high number of integer functional units. Suitable workloads are ones with high ILP and a high number of integer operations. *Narrow/VFP* has a narrow pipe and two vector/SIMD/FP units. Suitable workloads are ones with a high number of vector operations. *Narrow/Int* has a narrow pipe and runs at high frequency. This core is suitable for workloads with low ILP and a high number of integer operations. *Control* has a big instructions Cache and a Huge BP. This core is expected to execute control-heavy (a lot of conditions, branches, etc.) workloads efficiently.

Table 4.2: Core configurations

	Small	Wide/Int	Narrow/VFP	Narrow/Int	Control	Big
ICache	24k	32k	32k	32k	64k	32k
DCache	24k	64k	64k	32k	32k	48l
BP	Small	Med	Med	Big	Huge	Med
Alu Simple	1	6	2	3	4	3
Alu Complex	1	2	1	1	1	1
Mem	1	3	2	2	3	3
Simd FP	1	1	2	2	1	1
Width	1	5	2	2	4	3
Freq	1Ghz	2Ghz	2Ghz	2.4Ghz	1.8Ghz	2Ghz
Core Area	1.15mm	1.27mm	1.28mm	1.2mm	1.23mm	1.23mm

4.3 Results: static assignment

We demonstrate the improvement potential of ASCMP by assigning benchmarks to suitable cores. We use four of the six core types: Wide/Int, Narrow/VFP, Narrow/Int, and Big. Control and Small are omitted since both of them are efficient in small portions of each one of the benchmarks.

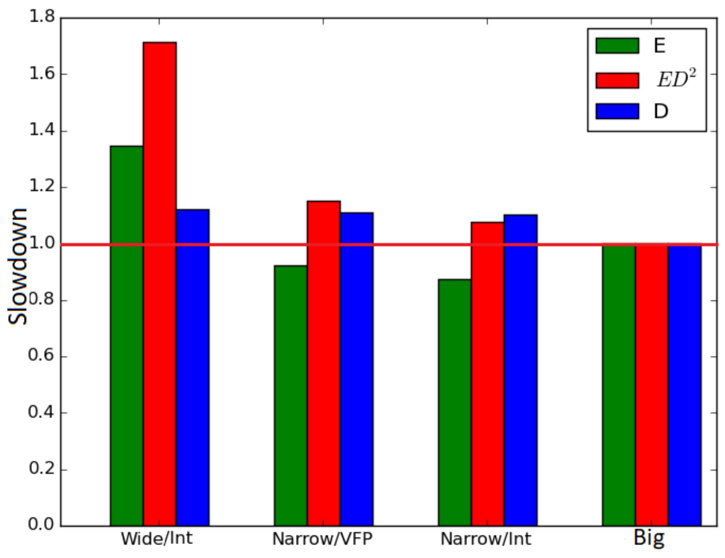

Figure 4.2: The average performance over all benchmarks on each core type. All bars are normalized to Big.

Figure 4.2 shows the performance average over all benchmarks on each core for energy, delay, and ED^2. We can see that on average, all three specialized cores perform worse than Big for ED^2 metric. The Narrow architectures show better energy consumption, whereas Wide/int shows the worst delay and energy. Big is the best when averaging over all benchmarks, both in delay and ED^2.

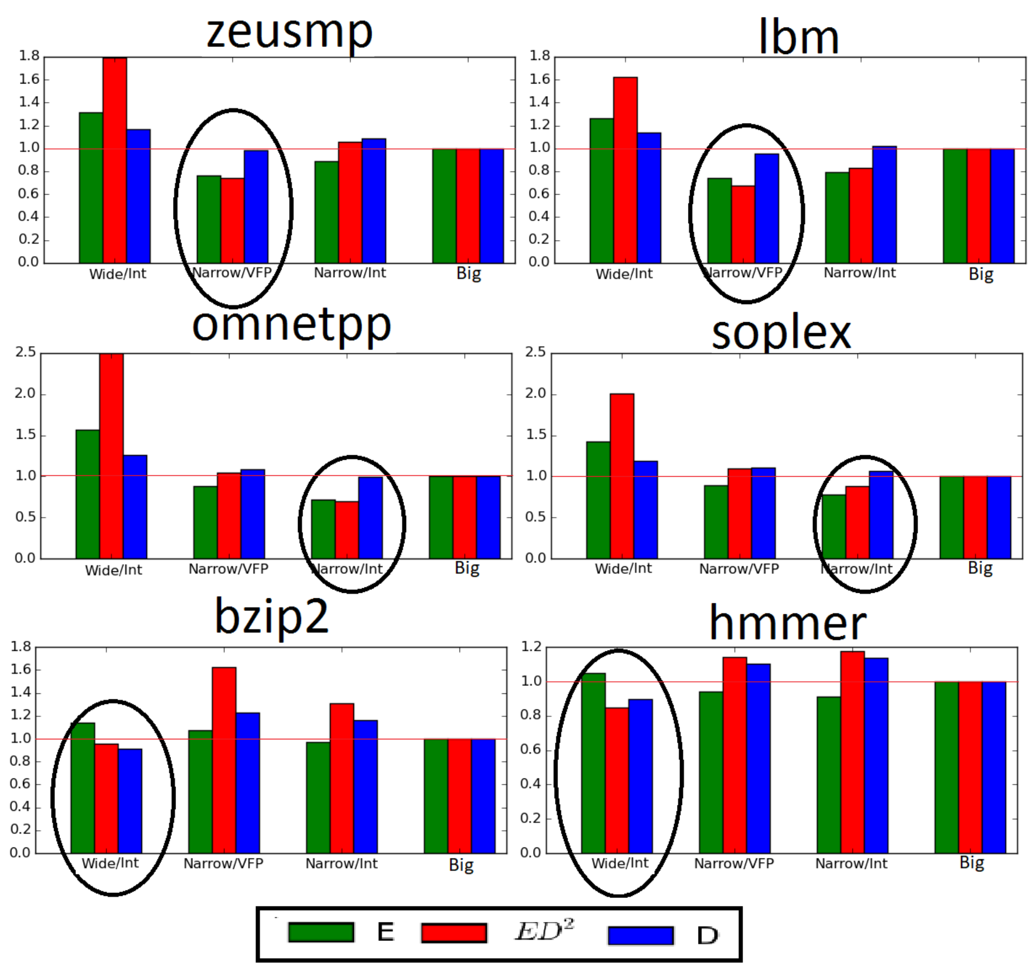

Figure 4.3: Performance under static assignment of different benchmarks to all core types.
All bars are normalized to Big. The best performing core is encircled.

Figure 4.3 shows per benchmark results. The core achieving the best ED^2 per benchmark is marked by a black circle. Here, we see a different picture. Big, which has the best ED^2 in the average case, is not the best in any particular benchmark. In other words, Big is a "jack-of-all-trades but a master of none".

In zeusmp and lbm, which are vector-operation-heavy (52% and 49% respectively) we see that the best core is Narrow/VFP. Big's performance is almost 35% worse, and Wide/Int core is almost 3x worse than Narrow/VFP.

For omnetpp and soplex, Narrow/int is the best core, while others perform between 1.2x to 5x worse. As we can see in Table 4.1, both have relatively high values of AIDC which, indicates low ILP.

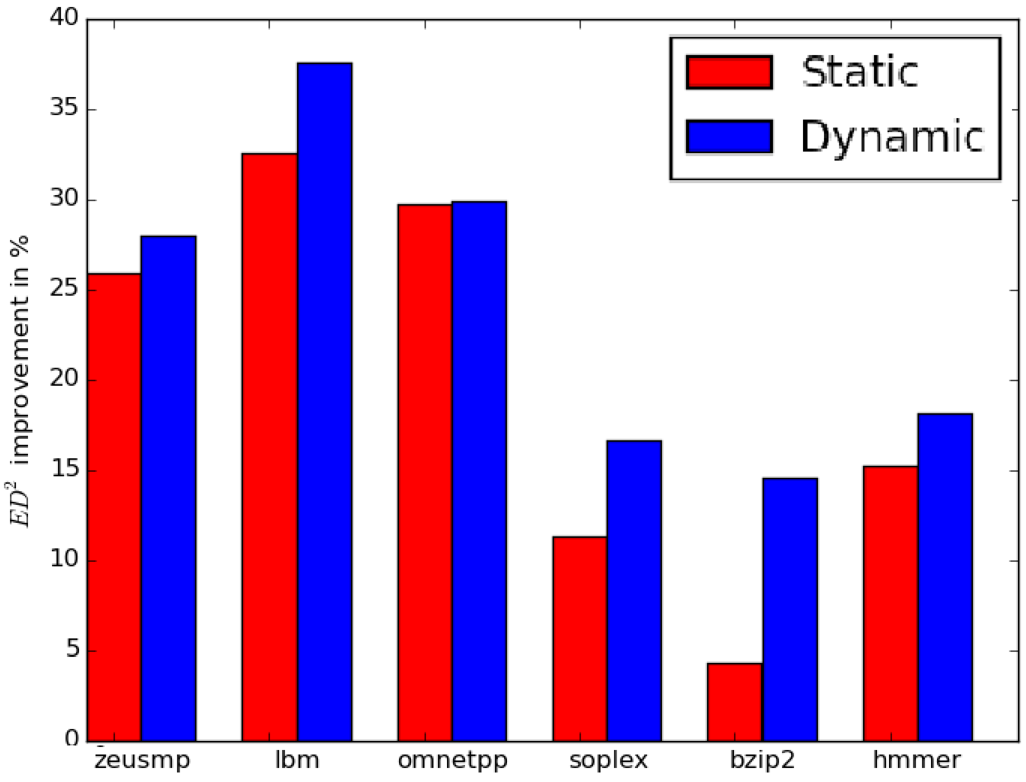

Figure 4.4: Improvement over Big, dynamic vs. static assignment.

In the bottom row, bzip2 and hmmer show mild performance gain vs. Big (10%-20%) when executed on Wide/Int.

4.4 Results: dynamic assignment

We now further allow on-the-fly migration of workloads between cores. For this purpose, we have generated an optimal execution per benchmark, where each portion of the workload is executed on the most suitable core, selected from our six core described in Table 4.2.

Figure 4.4 shows improvements of ED^2 in percentages over the static scheduling for the test cases from the previous section. As shown in the graph, most of the benchmarks show mild improvement, indicating the benchmark behavior does not change often. But a few, most notably bzip2, do improve substantially.

We take a closer look at the two extremes: omnetpp which show almost no improvement, and bzip2 which show almost 15% with migrations and less than 5% without migrations.

Figure 4.5 shows performance over time for all cores running omnetpp. As seen in the graph, the behavior is constant throughout the whole benchmark (except for initialization), and Narrow/Int is the most suitable core from beginning to end.

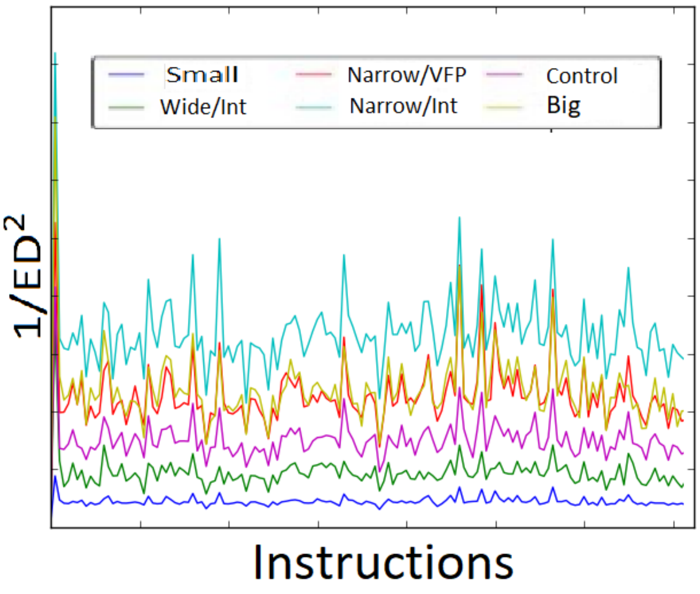

Figure 4.5: omnetpp execution on all cores.

In contrast, Figure 4.6 shows that bzip2 repeatedly transitions between behaviors, and the most suitable core alternates between Narrow/Int, Wide/Int, and Big core.

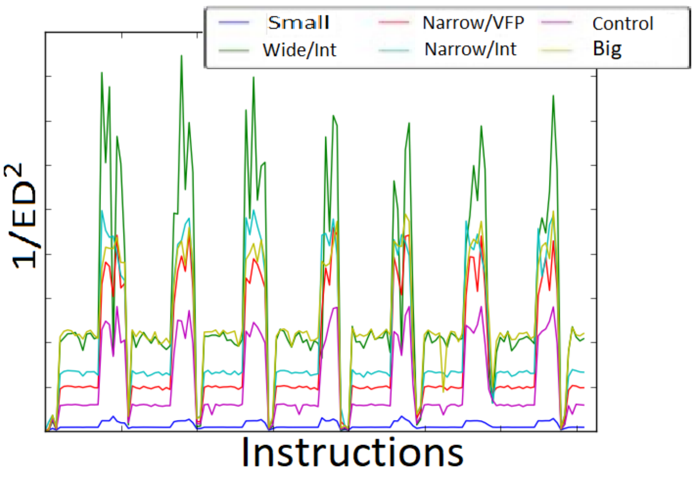

Figure 4.6: bzip2 execution on all cores.

To complete the picture, Figure 4.4 provides all tested benchmarks' potential improvements using ASCMP with prefect scheduling. The improvement over using Big ranges between 2% and 37%, with an average of 13%.

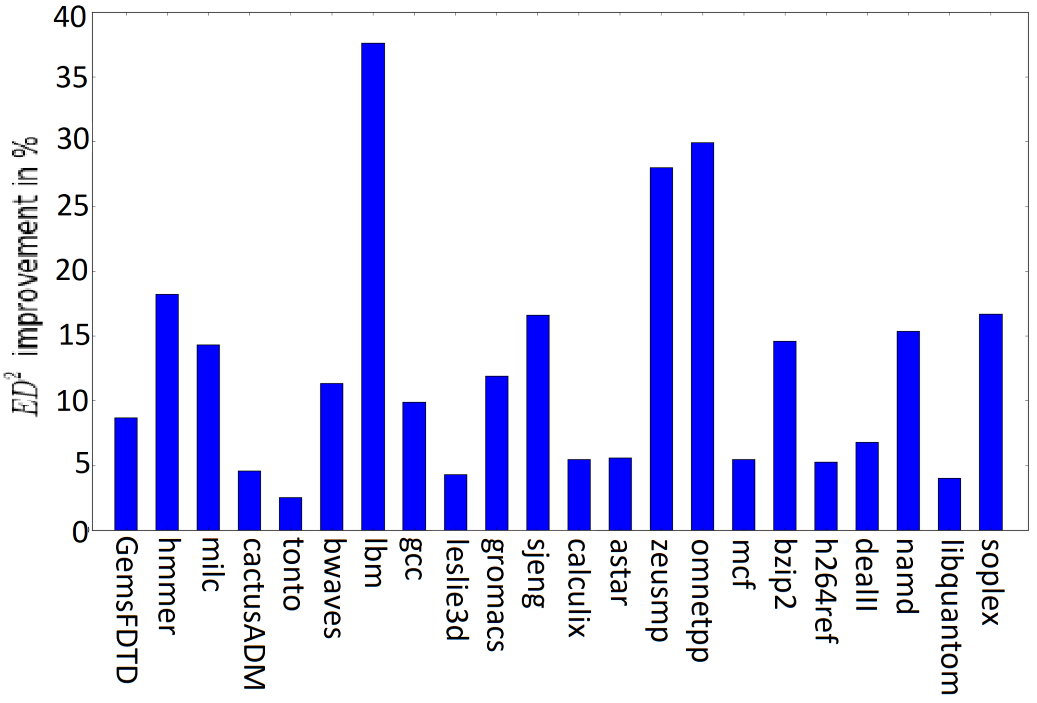

Figure 4.7: Improvement over Big with a perfect scheduler, for all benchmarks.

5 History-based prediction using signatures

I know whatever it is
I've not seen one before
But here comes another one
And here comes a bunch of 'em

Monty Python, Here Comes Another One

In this chapter, we create a method for predicting performance based on previous experience of executing workloads with similar behaviors. We use workloads' behavioral properties defined in the previous chapter as candidates for elements in micro-architecture-independent signatures that represent behaviors. We use these signatures for online performance prediction.

The remainder of this chapter proceeds as follows. In Section 5.1, we define basic concepts and discuss workload similarities. Following that, in Section 5.2, we describe the process we use for building a micro-architecture-independent signature based on behavioral properties. In Section 5.3 we describe our predictor implementation. Finally, in Section 5.4 we test our prediction scheme.

5.1 Workload similarities

A core can run multiple programs concurrently. OS processes are one abstraction for handling such concurrency; virtualization is another. In our work, we use the term *process* to describe instructions belonging to the same context from the core's perspective. For example, multiple threads belonging to the same process (e.g., using POSIX threads library without multicore support) are considered sequential even through threads context-switch.

The same process can go through phases of execution in which it performs different tasks. Thus, its behavior can change over time [31]. Our terminology of a workload behavior classification corresponds with the program phase concept in [7], but our detection scheme is not dependent on performance or instructions working set. Our classification and detection rely on changes in the workload behavior.

Roughly speaking, we say that workloads' behaviors are similar if their interaction with the cores is similar. By interaction, we mean executed instructions (histogram and order), working data set, resource dependencies, etc.

We denote by BV our behavior vector, which is composed of some measurable characteristics of a workload in a given window of execution. To be able to find similarities among workloads running on different cores, it is crucial to select BV elements that are micro-architecture-independent (similar to architectural signatures in [30]). Also, these elements should have a meaningful impact on performance, i.e., workloads with similar BVs should demonstrate similar performance when executing on a particular core.

Although we aim to create micro-architecture-independent signatures, in order to classify and group workloads that resemble each other we have to make some reasonable assumptions regarding the micro-architecture. For example, when categorizing instruction into types, we assume that executions of ADD and SUB instructions result in very similar operations by the core, and therefore are performance-equal, so we can use one counter to measure both.

5.2 Building the signature

In this section, we describe the process we used to reach a practical micro-architecture-independent signature. The main requirements are that workloads that share a signature, i.e., belong to the same behavioral similarity group, will have similar performance characteristics on all cores. The goal is to find a signature structure that is both accurate and usable, so we able to predict performance accurately based on the history of workloads that share the same signature.

We have used our altered version of GEM5 to produce the elements we use as candidates for the signature, namely, ITV, MRD, and AIDC. We have executed all the benchmarks described in Table 4.1 on all cores in Table 4.2, collected our signature candidates, raw IPS, and average power per window of execution, which we set to 25K.

The signature space includes a coefficient and *number of bits (bit-width)* for each of the properties we capture in the execution phase. Using an automated script, we performed an educated search within the signature space and gave a score to each of the signature structures. The process was to generate random bit-widths and coefficients within reasonable limits and check the following:

1. Accuracy: Let M be a set of collected $\langle \sigma, x \rangle$ pairs where σ is a signature and x is an ED^2 measurement for a workload with signature σ. We denote by $M(\sigma) \triangleq \{x | \langle \sigma, x \rangle \in M\}$. For a signature σ, denote $\bar{x}_\sigma$ the average of all measurements in $M(\sigma)$:
$$\bar{x}_\sigma \triangleq \frac{\sum_{x \in M(\sigma)} x}{|M(\sigma)|}.$$

We then define the average error and accuracy as follows:

$$Avg.Error \triangleq \frac{\sum_{\langle\sigma,x\rangle\in M} |(x-\bar{x}_\sigma)|}{|M|}; \quad \text{and}$$

$$Accuracy \triangleq 100\cdot(1-Avg.Error).$$

2. Usability is the percentage of the signatures shared among multiple benchmarks.

3. Size is the overall bit count of the signature.

The signature structure score is computed as $score = \frac{Usability\cdot Accuracy^2}{Size}$. We run the script several times, each time limiting the search space according to the limits of the few a best results of previous iteration. The best signature structure we reached, as shown in Figure 5.1 is composed of the following:

- 2 bits for memory operations (coef=142);
- 1 bit for int operations (coef=78);
- 1 bit for SimdFP instructions (coef = 173);
- 2 bits for control instructions (coef = 67);
- 1 bit for MRD number of references (coef = 160);
- 3 bits for MRD average distance (coef = 200); and
- 1 bit for AIDC average distance (coef = 440).

This 11-bit signature strucutre has an Accuracy of 95.55% and a Usability of 83%.

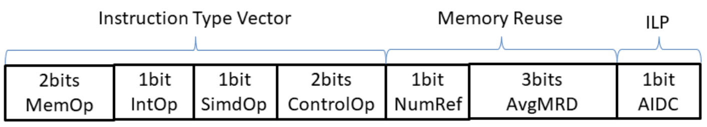

Figure 5.1: signature structure.

5.3 The predictor

The predictor works as part of an ASCMP platform in which each core periodically provides a measured pair $\langle\sigma,x\rangle$, where σ is a signature generated by our signature generation scheme described in Section 5.2 and x is an ED^2 measurement for the given window.

The predictor maintains two data structures: *LEARN* contains the last measurement x obtained for each core c and signature σ. *PRED* maps a signature σ to the highest performing core for σ.

The predictor reacts to prediction requests by the cores. For each incoming pair $\langle\sigma,x\rangle$, the predictor first tries to find σ in *PRED*. In case it is found (i.e., *predictor hit*), the predictor returns $PRED[\sigma]$. In case of σ is not in *PRED* (i.e., *predictor miss*), the predictor stores the $\langle\sigma,x\rangle$ pair in *LEARN*. In case *LEARN* already contains an entry for each core in the system, the predictor adds an entry $\sigma \rightarrow C$ to *PRED*, where C is the core providing the highest performance for σ among the entries in *LEARN*. In case *LEARN* does not contain an entry for each core in the system, the predictor returns an arbitrary core that was not yet recorded in *LEARN* for the given σ. There is no limit on the number of signatures that are stored in *LEARN*, but any signature that was sampled on all cores is garbage-collected and a corresponding entry is created in *PRED*.

5.4 Experiment

We now implement a simplified predictor. The predictor acquires the prediction data by executing a training set of benchmarks on all cores in a training phase. We set the execution window to 25K instructions.

We use our collected signatures and performance data obtained in Section 5.2 for a training subset of 16 benchmarks: astar, mcf, libquantom, h264ref, bwaves, milc, gromacs, cactusADM, leslie3d, namd, dealII, calculix, gemsFDTD, sjeng, tonto, and gcc. Each of the benchmarks is executed on each of the cores in our ASCMP, storing the power and IPS parameters per each signature in our predictor data structure. The number of recognized signatures at the end of the training was 576, covering 28% of the 2^{11} signature space. There were no apparent redundant bits (i.e., bits that had constant value throughout the training).

Following that, we used the trained predictor to run each of the remaining benchmarks on its own: zeusmp, lbm, omnetpp, soplex, bzip2, and hmmer. We did not update the trained predictor between executions.

Throughout this chapter, we assume the migration penalty is 1000 cycles. This correspond to hardware-based migrations or core reconfiguration models [28, 20]. We note that in case of operating system involvement, where the penalty can be much higher, a different granularity of behavior sampling (i.e., execution window size) should be used.

We do not simulate a multi-core environment using GEM5 or execute the benchmarks concurrently. Rather, we emulate running each of the benchmarks individually on a six-core system as described in Table 4.2. This execution model is applicable in various architectures, e.g., Composite Cores [20].

Our migration decision algorithm is simple; if the predictor returns the highest score to the same core for four execution windows in a row (a total of 100K instructions), we migrate the workload to that core. In case of a predictor miss, the workload is assigned immediately to the core suggested by the predictor. This is done in order to shorten new signature learning phase in operational mode.

Table 5.1 summarizes our prediction behavior. To all benchmarks but lbm, training was sufficient, showing a low number of misses. For lbm, the predictor suffers from extremely high miss rate and a large number of migrations, indicating high learning penalty and rapid changes of behaviors.

Figure 5.2 compares perfect assignment presented in the last chapter with the emulated predictor, by showing the percentage of improvement in ED^2. When comparing lbm predictor performance to perfect, we see significant degradation, which corresponds to the high miss rate. The rest of the benchmark shows a very close performance of our predictor to a perfect one.

Table 5.1: Learning predictor hits, misses and migrations.

	Hits	Miss	Num Migrations
bzip2	142823	1	1312
omnetpp	143061	5	40
zeusmp	143427	94	359
soplex	148118	143	784
hmmer	154371	0	328
lbm	158743	1046	4509

The results for the six benchmark using the trained predictor range between 12.5% and 33.8% with an average of 22.2%.

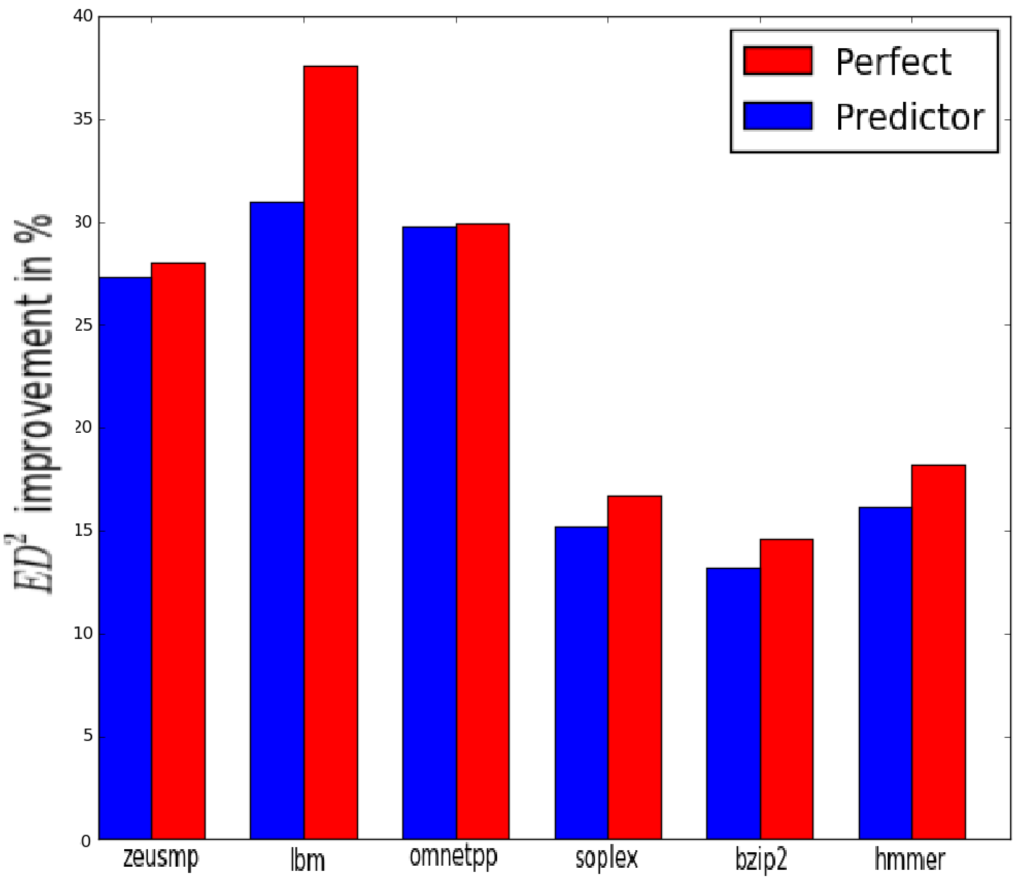

Figure 5.2: Performance improvements over Big – Perfect vs. Predictor.

6 Conclusion

Making the simple complicated is commonplace; making the complicated simple, awesomely simple, that's creativity.

Charles Mingus

In this thesis, we have argued that using a linear set of cores, ranging from little to big, limits the potential of ACMPs. We have suggested using specialized cores which comply to the same ISA to achieve higher power efficiency. To manage the complexity of running diverse workloads on diverse cores, we have created a mechanism, based on sampling and micro-architecture-independent signatures. We have shown that due to a high correlation between workloads that share signatures and their performances, we can group different workloads into similarity groups and build a reliable history-based performance predictor.

We evaluated our proposed system using the GEM5 architectural simulator. We altered GEM5 so it provides all the information we need to produce micro-architecture-independent signatures as well as IPS, instruction histograms, and other information that is required for calculating power. For power and area estimation we used McPAT. We have executed the SPEC2006 CPU benchmark suite, which provides compute-intensive applications from diverse fields.

Our ideal system, which consists of six cores with perfect assignment and zero migration cost, shows up to 37% power-performance gain with 13% on average for the SPEC2006 benchmarks in comparison to big.LITTLE based ACMP.

Our proposed predictor, using an 11-bit signature structure shows an average prediction error of less than 4.5%. The same system with a trained history-based predictor, with a migration penalty of 1000 cycles, shows up to 33.8% power-performance gain with 22.2% on average for a subset of six benchmarks selected from SPEC2006 suite.

References

[1] Arunachalam Annamalai, Rance Rodrigues, Israel Koren, and Sandip Kundu. An opportunistic prediction-based thread scheduling to maximize throughput/watt in amps. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 63–72. IEEE Press, 2013.

[2] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 506–517. IEEE, 2005.

[3] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*, pages 29–40. ACM, 2006.

[4] Erik Berg and Erik Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pages 20–27. IEEE, 2004.

[5] Kristof Beyls and Erik DHollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360, 2001.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[7] Ashutosh S Dhodapkar and James E Smith. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 217. IEEE Computer Society, 2003.

[8] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak

Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.

[9] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[10] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

[11] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3), 2007.

[12] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 186–197. ACM, 2007.

[13] Richard E Kessler, Edward J McLellan, and David A Webb. The alpha 21264 microprocessor architecture. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD'98. Proceedings. International Conference on*, pages 90–95. IEEE, 1998.

[14] Omer Khan and Sandip Kundu. A self-adaptive scheduler for asymmetric multi-cores. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 397–400. ACM, 2010.

[15] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*, pages 125–138. ACM, 2010.

[16] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92. IEEE, 2003.

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

[18] Daniel Leibholz and Rahul Razdan. The alpha 21264: A 500 mhz out-of-order execution microprocessor. In *Compcon'97. Proceedings, IEEE*, pages 28–36. IEEE, 1997.

[19] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and

manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[20] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 317–328. IEEE Computer Society, 2012.

[21] Jason Mars and Lingjia Tang. Whare-map: heterogeneity in homogeneous warehouse-scale computers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 619–630. ACM, 2013.

[22] MediaTek. *MediaTek CorePilot^{TM} - Heterogeneous Multi-Processing Technology*, 2013. URL `http://cdn-cw.mediatek.com/WhitePapers/MediaTek_CorePilot.pdf`.

[23] Tomer Y Morad, Uri C Weiser, A Kolodnyt, Mateo Valero, and Eduard Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Computer Architecture Letters*, 5(1):14–17, 2006.

[24] Peter Greenhalgh, ARM. Big. LITTLE Processing with ARM Cortex-A15 & Cortex-A7, 2011.

[25] Fred J Pollack. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address). In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, page 2. IEEE Computer Society, 1999.

[26] Qualcomm. *Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age*, 2012. URL `https://developer.qualcomm.com/download/qusnapdragons4whitepaperfnlrev6.pdf`.

[27] Krishna K Rangan, Michael D Powell, Gu-Yeon Wei, and David Brooks. Achieving uniform performance and maximizing throughput in the presence of heterogeneity. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 3–14. IEEE, 2011.

[28] Rance Rodrigues, Arunachalam Annamalai, Israel Koren, and Sandip Kundu. Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(1):5, 2013.

[29] Daniel Shelepov and Alexandra Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

[30] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[31] Timothy Sherwood and Brad Calder. Time varying behavior of programs, 1999.

[32] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *Micro, IEEE*, 23(6):84–93, 2003.

[33] Sadagopan Srinivasan, Ravishankar Iyer, Li Zhao, and Ramesh Illikkal. Heteroscouts: hardware assist for os scheduling in heterogeneous cmps. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 149–150. ACM, 2011.

[34] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 213–224. IEEE Press, 2012.

[35] Uri Weiser. Vlsi architecture design:asymmetric platform approach. Retrieved from http://eecourses.technion.ac.il/046853/lectures/Winter2010/03 VLSI Asymmetric Platform approach 2011.pdf, 2011.

[36] Jonathan A Winter and David H Albonesi. Scheduling algorithms for unpredictably heterogeneous cmp architectures. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 42–51. IEEE, 2008.

[37] Jonathan A Winter, David H Albonesi, and Christine A Shoemaker. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 29–40. ACM, 2010.

[38] Shikang Xu, Israel Koren, and CM Krishna. Improving processor lifespan and energy consumption using dvfs based on ilp monitoring. In *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*, pages 1–6. IEEE, 2015.

מבנה החתימה צריך להיות קטן מספיק (מבחינת מספר ביטים) כדי לאפשר מנגנון אפקטיבי, ולהשתמש רק במאפיינים שאפשרי והגיוני להפיק בזמן ריצה. בנוסף, החתימה צריכה למלא שתי דרישות שלעיתים מנוגדות זו לזו:

1. דיוק – החתימה צריכה להיות גמישה מספיק כדי לבטא התנהגויות שונות ולספק תאימות גבוהה בין החתימה לבין מאפייני הביצועים ביחידות העיבוד השונות.
2. יכולת אגרגציה – החתימה צריך להיות קטנה מספיק כדי לאפשר קיטלוג של תוכנות שונות לאותה קבוצה.

בתזה זו, אנו מתמודדים עם האתגר של מציאת עמק השווה בין שתי דרישות מנוגדות אלו ויוצרים חתימה שהיא גם הגיונית למימוש וגם פרקטית לשימוש.

אנו בוחנים את המערכת בעזרת סימולטור ארכיטקטוני שנקרא GEM5. אנו משתמשים בסט פקודות של ARM, במעבד out-of-order ובמערכת הזיכרון הקלאסית. אנו משנים את GEM5 כך שיפיק את המידע לו אנו זקוקים כדי לייצר את החתימות, ואת מאפייני הביצועים (פקודות בשנייה) ומאפיינים נוספים הנדרשים כדי לחשב הספק. על מנת לחשב שטח והספק אנו משתמשים בכלי נוסף הנקרא McPAT. התוכניות שבהן אנו משתמשים לשם בחינת התזה הן מחבילת SPEC-CPU-2006 benchmark suite אשר מספקת תוכניות מרובות חישובים מתחומים שונים.

סט יחידות העיבוד שאנו בוחנים, במודל ביצוע אופטימלי וללא עלויות הגירת תהליכים מראה שיפור ביצועים של עד 37% וממוצע של 13% עבור 22 תוכניות שנבחנו.

המערכת שאנו בוחנים, המורכבת מסט יחידות העיבוד בנוסף לחזאי שאומן על 16 תוכניות, במודל שבו עלות הגירת התהליכים היא 1000 מחזורי שעון, מראה שיפור ביצועים של עד 33.8% וממוצע של 22.2% לשש תוכניות שנבחנו.

תרומות התזה:

1. אנו מראים ששימוש במעבדים "מומחים" משפר ביצועים.
2. אנו מראים שחתימות שאינן תלויות מיקרו-ארכיטקטורה יכולות לזהות התנהגות של תוכניות ולתת אינדיקציה טובה לביצועים עבור תוכניות שחולקות אותה חתימה.
3. אנו מציעים שיטה לחיזוי ביצועים שמבוססת על היסטוריה של תוכניות דומות.

תקציר

במסגרת מחקר זה נבחנה האפשרות לשילוב מספר יחידות עיבוד בעלות ממשק זהה ומימוש פנימי שונה.

מולטי מעבדים הטרוגניים מכילים מספר סוגים של יחידות עיבוד על שביב אחד. כל יחידת עיבוד עובדת בצורה שונה ולכן מספקת יכולות וביצועים מגוונים. מולטי מעבד אסימטרי הוא מולטי-מעבד הטרוגני, שבו כל יחידות העיבוד תואמות לאותו סט פקודות (ISA).

מולטי מעבדים אסימטריים משפרים ביצועים על ידי הרצת תוכניות על יחידות עיבוד מתאימות. התאמה של תוכניות ליחידות עיבוד, כאשר התנהגות התוכניות משתנה, היא אתגר. פתרונות מעשיים עד כה הניחו, שניתן לסדר את יחידות העיבוד על פי חוזק: מיחידות ביצוע קטנות ופשוטות, שחוסכות בהספק, ועד ליחידות ביצוע גדולות ומורכבות, שמיועדות לתוכניות מרובות חישובים. גישה זו מאפשרת התאמה יחסית פשוטה בין תוכנית ליחידת עיבוד.

בעוד שארכיטקטורות מבוססות big.LITTLE של ARM תופסות נתח שוק עצום (Apple A10, Nvidia Kal-el, Qualcomm Krait, Samsung Exynos, MediaTek CorePilot, Hauwei Kirin, ואחרים), התעשייה עדיין נמנעת ממערכות אסימטריות מגוונות יותר. אחת הסיבות היא ההנחה הגורסת כי החישוב הנוסף שיידרש לניהול מערכות כאלה הוא גדול.

בתזה זו, אנו טוענים שהגבלת מרחב התכן לסידור לינארי (מקטן עד גדול) מגביל את הפוטנציאל של מולטי-מעבדים אסימטריים. אנו בוחנים את הרעיון של שימוש במספר מעבדים "מומחים" בשביל להשיג ביצועים טובים יותר מאשר מולטי-מעבדים אסימטריים לינאריים. המוטיבציה לשימוש במעבדים "מומחים" נובעת מהעובדה שככל שהתכן ממוקד יותר עבור תוכנית מסוימת, כך ניתן להגיע לביצועים טובים יותר, להספק נמוך יותר ולשטח קטן יותר.

ניהול ביצוע של תוכנות מגוונות במולטי-מעבדים אסימטריים מגוונים הוא משימה מורכבת. על מנת לבצע משימה זו ביעילות יש צורך במנגנון, שיחזה ביצועים ויתאים את התוכניות למעבדים המתאימים. לשם מטרה זו, אנו מציעים טבלת חיפוש מבוססת היסטוריה, שיכולה לחלוק מידע בין יחידות העיבוד השונות. בזמן הרצת תוכנית כל אחת מיחידות העיבוד מפיקה חתימות שאינן תלויות במיקרו-ארכיטקטורה ודוגמת את ביצועי יחידת העיבוד המתאימים לחתימה. החתימה נבנית מנתונים הקשורים לתוכנית בלבד, ולא מהשפעת התוכנית על יחידת העיבוד. לדוגמה, היסטוגרמת פקודות ודפוס גישה לזיכרון הם תכונות של התוכנית. החטאות זיכרון מטמון אינן תכונות של התוכנית. בכך שאנו משתמשים בנתונים שאינם תלויים במיקרו-ארכיטקטורה בלבד, אנו מבטיחים שלכל תוכנית תהיה אותה חתימה בלי שום קשר ליחידת העיבוד שמבצעת אותה תוכנית.

החתימה נוצרת למטרת אפיון ביצועים עבור כל יחידת עיבוד, כאשר תוכניות שחולקות אותה חתימה צפויות להראות ביצועים דומים כאשר הן מבוצעות על אותה יחידת עיבוד. באופן זה מערכת ניהול הביצוע מנצלת דמיון על מנת לחזות ביצועים של תוכנית המראה התנהגות מסוימת, לתוכנית אחרת המראה התנהגות דומה.

המחקר נעשה בהנחיית פרופסור עדית קידר בפקלוטה להנדסת חשמל בטכניון.

ניצול דמיון בין התנהגות תהליכים לשם החלטות השמה במערכות אסימטריות מגוונות מרובות מעבדים

חיבור על מחקר לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים בהנדסת חשמל

דני שקט

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

באדר תשע"ח, חיפה, פברואר 2018

ניצול דמיון בין התנהגות תהליכים לשם החלטות השמה במערכות אסימטריות מגוונות מרובות מעבדים

דני שקט