DESIGNING HETEROGENEOUS MANY-CORE PROCESSORS TO PROVIDE HIGH PERFORMANCE UNDER LIMITED CHIP POWER BUDGET

A Thesis
Presented to
The Academic Faculty

by

Dong Hyuk Woo

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2010

DESIGNING HETEROGENEOUS MANY-CORE PROCESSORS TO PROVIDE HIGH PERFORMANCE UNDER LIMITED CHIP POWER BUDGET

Approved by:

Dr. Hsien-Hsin S. Lee, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sung Kyu Lim
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Dr. Marilyn Wolf
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 23 September 2010

To my family.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all those who directly or indirectly helped me in completing my Ph.D. study.

First of all, I would like to thank my advisor, Dr. Hsien-Hsin S. Lee, who continuously motivated me, patiently listened to me, and often challenged me with critical feedback. I would also like to thank Dr. Sudhakar Yalamanchili, Dr. Marilyn Wolf, Dr. Sung Kyu Lim, and Dr. Milos Prvulovic for volunteering to serve in my committee and reviewing my thesis.

I would also like to thank all the MARS lab members, Dr. Weidong Shi, Dr. Taeweon Suh, Dr. Chinnakrishnan Ballapuram, Dr. Mrinmoy Ghosh, Fayez Mohamood, Richard Yoo, Dean Lewis, Eric Fontaine, Ahmad Sharif, Pratik Marolia, Vikas Vasisht, Nak Hee Seong, Sungkap Yeo, Jen-Cheng Huang, Abilash Sekar, Manoj Athreya, Ali Benquassmi, Tzu-Wei Lin, Mohammad Hossain, Andrei Bersatti, and and Jae Woong Sim. A special thanks to Mrinmoy Ghosh for his help in my early research, from which I was able to easily learn how to conduct academic research. Another special thanks to Nak Hee Seong for his help in my recent projects. Countless rounds of in-depth discussion with him helped me to better understand various design trade-offs of memory hierarchy.

On the other hand, I would like to express gratitude to all the people who actively participated in our weekly computer architecture meeting, *arch-beer:* Dr. Gabriel Loh, Dr. Hyesoon Kim, Dr. Nathan Clark, Dr. Samantika Subramaniam, and Dr. Guru Venkataramani, just to name a few. From the heated discussion with them, I learned how to find a new problem, how to develop a reasonable solution to a problem, and how to evaluate a proposed solution fairly.

I also appreciate Dr. Josh Fryman and Dr. Allan Knies of Intel Corporation for providing assistance and guidance during the summer internship in 2006. This internship project has later become a part of my thesis. Especially, Dr. Fryman taught me how to read a paper critically, which helped me a lot throughout my graduate study.

Furthermore, I would like to thank all members of our 3D-MAPS project, Krit Athikulwongse, Rohan Goel, Michael Healy, Mohammad Hossain, Moongon Jung, Dae Hyun Kim, Young-Joon Lee, Dean Lewis, Tzu-Wei Lin, Chang Liu, Brian Ouellette, Mohit Pathak, Hemant Sane, Guanhao Shen, and Xin Zhao. While designing a 3D-stacked many-core processor together for this project, they directly and indirectly helped me in many aspects.

In addition to those who I know personally, I would also like to thank anonymous conference and journal reviewers who volunteered to read my papers and provided many useful comments. Additionally, I would like to appreciate those who has developed and maintained the open-source SESC simulator, which was extremely useful throughout my graduate study.

Most importantly, I would like to appreciate my wife, Jung In Sung, for her devotion and endurance. Without her endless support, this thesis would not have been possible. I also thank my son, Nolan Hyunjae Woo, whose smile always cheers me up. Finally, I appreciate my parents, Sang Kyu Woo and Young Hee Shin, for their unconditional love and support not only during my graduate study but also throughout my life so far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis describes the efficient design of a future many-core processor that can provide higher performance under the limited chip power budget. To achieve such a goal, this thesis first develops an analytical framework within which computer architects can estimate achievable performance improvement of different many-core architectures given the same power budget. From this study, this thesis found that a future many-core processor needs (1) energy-efficient parallel cores and (2) a high-performance sequential core. Based on these observations, this thesis proposes an energy-efficient broad-purpose acceleration layer that can be snapped on top of a conventional general-purpose processor. In addition to such an energy-efficient parallel cores, this thesis also proposes different architectural techniques to further boost the performance of sequential computation while those parallel cores are idle. In particular, this thesis develops low-cost architectural techniques to enhance the memory performance of a host core by utilizing those idle parallel cores. This idea is evaluated in two different system architectures: one with the aforementioned acceleration layer and the other with an emerging integrated CPU and GPU chip.

CHAPTER I

INTRODUCTION

Unsustainable power consumption and ever-increasing design and verification complexity have driven the microprocessor industry to integrate multiple cores on a single die, called a multicore processor, as an architectural solution to sustaining Moore's Law [44]. With dual-core and quad-core processors on the market and an oct-core processor on the horizon, researchers are already a step ahead. They are investigating architectures, compilers, and programming models for a many-core processor with hundreds or even thousands of cores on a single platform [49, 57].

To integrate this number of cores on a single die, computer architects must address the problem of the fundamental physical limit, power consumption, to make a many-core architecture viable. For future many-core processors, low power will be not only a feature, but also a design constraint. In other words, achieving the goal of integrating a large number of cores onto one chip boils down to one issue, how to design many processing cores so that they do not consume more power than the chip power budget.

To address this question, we proposed an analytical framework within which computer architects can estimate achievable performance improvement of different many-core architectures given the same power budget [127]. This study provided us two interesting insights. The first insight is that building smaller, but more efficient parallel processors provides greater performance than replicating a state-of-the-art superscalar processor. The second insight is that sequential performance is not only important to improve overall performance, but also critical to achieve higher energy efficiency.

To achieve both energy efficiency and best-of-class single-thread performance, we also proposed a 3D-integrated broad-purpose acceleration layer that can be snapped on top of a conventional superscalar processor die [125]. Building such an efficient acceleration layer with a conventional superscalar processor poses several challenging problems. The first problem is binary compatibility between the host processor and the acceleration layer. Because the host superscalar processor is a complex instruction set computer (CISC) superscalar processor such as Intel Core and each parallel core in the acceleration layer is an energy-efficient, very long instruction word (VLIW) processor, the host processor should have an efficient mechanism to solve the problem of binary heterogeneity. The second problem is efficient interaction between the host processor and the parallel cores. As the host processor issues instructions in an out-of-order fashion by nature, either the parallel cores should be able to roll-back from misspeculation, or the host processor should be able to issue instructions in a different way. The third problem is designing an efficient interconnection network. As an interconnection network shares the same area and power budget with all other computation resources, the interconnection network should be as efficient as possible so that we can implement more computation resources instead. In this research, we designed the acceleration layer focusing on these three problems.

Although such an efficient asymmetric, heterogeneous many-core processor can provide high performance when running well-parallelized applications, such an architecture has a certain drawback: parallel cores are underutilized when the processor runs a single-thread application or the sequential code of a multi-thread application. During such a sequential computation phase, parallel cores of an asymmetric many-core processor remain idle while consuming area and power if not properly turned off. Instead of leaving them idle, we envision that we can utilize them to achieve higher single-thread performance by virtualizing them as a last-level cache and a data prefetcher. In the meantime, the overhead of such architectural techniques should be

low enough that they do not compromise the efficiency of the baseline asymmetric many-core processor. Thus, in this research, we propose low-overhead architectural techniques that can virtualize these idle resources for higher single-thread performance and evaluate their hardware and power overheads [126].

In addition, we also design and evaluate such an idea in a CPU and GPU integrated platform, which many industry leaders plan to commercialize [128]. From this study, we found that we have lots of interesting challenges in such an integrated platform. For example, a conventional GPU does not have large cache or scratch pad memories. Furthermore, the GPU pipeline is optimized for throughput, not for latency, which makes our goal, reducing memory latency, even more challenging. In this study, we address these challenges by exploiting properties of a conventional GPU with negligible hardware overhead so that our new proposal does not harm the performance of conventional 3D graphics or general-purpose GPU applications.

The remainder of this document is organized as follows. Chapter II enumerates prior studies related with this thesis. Chapter III proposes an analytical framework within which computer architects can estimate the energy-efficiency of different many-core designs. Chapter IV proposes a 3D-integrated broad-purpose acceleration layer that provides a power-scalable parallel execution engine. Moreover, Chapter V proposes low-overhead architectural techniques to reuse such an accelerator, when it is idle, as a virtualized last-level cache or a virtualized data prefetcher. Chapter VI further extends the idea of reusing idle heterogeneous cores for boosting the single-thread performance of central processing units (CPUs) on the same die; It proposes architectural techniques to reuse idle graphics processing unit (GPU) cores for the same purposes. Chapter VII concludes this dissertation.

CHAPTER II

ORIGIN AND HISTORY OF THE PROBLEM

2.1 Amdahl's Law

Amdahl's law [5] is used for predicting the maximum performance improvement when a new technique for improving the performance of a system applies to a subset of the system. A very good usage of it is parallel software. Parallel software programmers use Amdahl's law to estimate the theoretical maximum speedup of their parallelized software. Recent studies use Amdahl's law to estimate the performance of future many-core processors. Cho and Melhem extended the original Amdahl's law to find the optimal clock frequencies for serial and parallel code of a parallel program [24]. Hill and Marty proposed an extended Amdahl's law to show how much speedup can be achieved with different many-core processor designs given a fixed chip area [51].

2.2 Massively Parallel Processors

While this research revisits the massive single-instruction, multiple-data (SIMD) concept of massively parallel processors (MPPs), this research expands and interprets the SIMD concept with modern requirements and technological constraints. The closest architectures are Maspar MP-1 [10, 82, 93] and Thinking Machines CM-2 [122]. Both of these machines centered the concept of a very large number of processing tiles for parallel calculations. This style of SIMD machine was broadly characterized as having a stand-alone front-end system that consists of a host processor. Both machines take advantage of the relatively free wire latency compared to the transistor switching speed and have a reasonable communication latency between nearest-neighbor

processing elements (four cycles for CM-2 and eight cycles for Maspar). The nearest-neighbor connections are supplemented with sophisticated networks, which allow all-to-all, unstructured, and long-distance communication. The high number of processing elements (PEs) led to a sophisticated global network design with many layers of switches and crossbars. In the case of CM-2, it provided an n-way hypercube interconnect that linked clusters of PEs to other clusters. MP-1 utilized a multi-stage, multi-pass network that enabled arbitrary communications patterns albeit with substantially higher latency and setup costs.

Other important SIMD machines include Solomon [111], IBM GF-11 [69], and Illiac IV [13]. Some hybrid efforts between a multiple-instruction, multiple-data (MIMD) and a SIMD architecture were undertaken [115] but remain on the fringe. Systolic arrays [12, 70] resemble SIMD machines but were tailored for applications whose computations fit a narrower structure. In most cases, for higher efficiency, programmers need to map execution and data flow of an application in details to the target machine. Tarantula [33] extended Alpha instruction set architecture (ISA) with a slew of new instructions. EV8 understands an entire vector ISA for a renaming, a retirement, and a speculation issue and supports deep conditionals via masks. However, it does not support communication among vector units except for gather-scatter operations. VIRAM1 [38] is a vector processor with embedded dynamic random access memory (DRAM) for mobile multimedia devices.

Common SIMD use in modern processors is most evident in streaming SIMD extension (SSE) and 3DNow! extension as well as AltiVec or VMX Power Architecture extensions. These instructions were added to baseline ISAs to enable programmers and compilers to exploit data parallelism in media applications.

Imagine [4, 64, 106] is a stream-model processor but acts as a co-processor. Imagine uses a 128 KB stream register file to contain data while each attached arithmetic cluster has a local register file and several dedicated arithmetic logic units (ALUs)

to operate on stream data in a producer-consumer model. Imagine capitalizes on an eight-wide SIMD ability inside each full ALU tile. The drawback is that each of the eight sub-ALU blocks is wired with a crossbar to full streaming register files, and applications must be ported to a stream-based model for exposing the parallelism.

2.3 Tiled Architecture

Non-SIMD tile-based architectures, e.g., MIT RAW [117, 116] and UT-Austin TRIPS [108], have also been proposed. The RAW processor provides local instruction and data caches, contains 64KB of random access memory (RAM) in each processor tile for programming a dynamic and static router, and has its ALU bypass network tied directly into the interconnection network. To support its programming model, the RAW also has large memory space and additional modes dedicated to routing logic based on application needs for either static or dynamic routing. This approach requires extra logic and power compared to a pure SIMD design. TRIPS is tile-based, but it uses more sophisticated ISA mechanisms relying on a compiler analysis and static placement of computation. It also provides separate interconnection networks for data and instruction movement, and each tile has a complete CPU. The whole design is intended to support highly speculative parallelization techniques.

2.4 Accelerator Architecture

IBM Cell Broadband Engine [52], an alternative to tile-based designs, hosts eight synergistic processor elements (SPEs) on a PowerPC host. These SPEs are complete with instruction fetch, decode, and branch control and work in a MIMD fashion. Each SPE also provides 256KB of static random access memory (SRAM) for local program and data. SPEs are connected to a planar ring network and perform direct memory access (DMA) operations to transfer information from and to memory.

Recently, a general-purpose graphics processing unit (GPGPU) [77] has been attracting more interest from the high-performance computing market. Tesla [95] is a

GPGPU-based on a multi-threaded architecture for high-performance computing. It has 128 multi-threaded processors and supports up to 76.8 GBps of off-chip memory bandwidth but supports only IEEE 754 single-precision floating point operations. PhysX [3] is a specialized accelerator dedicated only to real-time physics to enrich game playing. ParallAX [129] was recently proposed as a future physics processor with detailed behavioral analysis of physics simulation.

2.5 Snap-On 3D Stacked Layers

In addition to its various advantages, 3D IC stacking technology also allows computer architects to snap on a new feature on top of a conventional product with reasonable cost. Mysore *et al.* proposed integrating specialized monitoring hardware stacked on top of the processor die using the 3D IC stacking technology [90]. On the other hand, Madan *et al.* proposed integrating a redundant checker processor on a second die to improve reliability of a processor [79].

2.6 Software Caching

Many systems have used software caching techniques. To improve the programmability of synergetic processing elements (SPEs) on IBM Cell/BE, IBM provides a software cache library as a part of the IBM Cell software development kit [7, 32]. The MIT Raw processor used software caching to emulate both instruction [84] and data cache [86]. The Stanford VMP multiprocessor handled cache misses using software techniques [22]. Furthermore, prior studies developed advanced compiler techniques to better manage a software cache memory [123, 62, 20]. While the goal of all these prior studies is to improve the programmability or the performance of a processor with their local scratch-pad memory, our proposed design is to provide a virtualized last-level cache to a host processor to improve the single-thread performance of the host processor.

2.7 Shared Cache Management

To use on-chip memory resources more efficiently, researchers have focused on managing a shared cache [67, 130, 18, 47, 53, 102, 42]. Zhang and Asanovic proposed a new cache management policy called victim replication, which combines the advantages of private and shared schemes in a tiled CMP [130]. Chang and Sohi used private caches for both dynamic sharing and performance isolation [18]. Harris proposed a synergetic caching policy that groups neighboring cores into a cluster to have shared memory space among them [47]. While these prior studies attempt to manage shared cache more efficiently, our work addresses the underutilization issue of heterogeneous multicore processors by virtualizing unused PEs.

2.8 Non-Uniform Cache Architecture

To tackle the long latency problem of the last-level cache, researchers proposed a non-uniform cache architecture (NUCA). Kim *et al.* proposed an adaptive, non-uniform cache structure [65]. On the other hand, Huh *et al.* studied an optimal degree of cache sharing in a NUCA [56]. Unlike the original NUCA proposal, NuRAPID decouples the tag and data array to enable flexible data placement [23]. Although our research adopts the idea of a NUCA, we propose an architectural solution called *time-zoning* to provide non-uniform cache access latency on a SIMD engine with a strictly synchronized execution model.

2.9 Prefetching Techniques

Hardware prefetching is widely investigated by prior studies [19, 60, 92, 63, 92, 74, 26]. These hardware prefetchers use dedicated hardware blocks to predict and bring in the cache lines in advance whereas COMPASS is designed to reuse the existing GPU hardware and leverage their programmability and flexibility so that one (*e.g.*, software vendors) can write or select the most appropriate COMPASS shader to improve the

performance of their applications. On the other hand, software prefetching was also investigated by prior studies [16, 21], and various data prefetch instructions were added in almost every commercial microprocessor. These techniques basically insert software instructions into a code so that a future memory fetch can be initiated in advance. COMPASS is different from a conventional software prefetching technique because of the following. First, COMPASS does not consume any compute bandwidth of the main processor, rather, it leverages the idle GPU to perform the task. Second, it emulates a hardware prefetcher and its associated hardware (*e.g.*, prefetch table). As we demonstrated, the emulated table oftentimes is much larger than the practical size of a real hardware table. Third, COMPASS does not require re-compilation of a code.

Helper thread techniques [30, 25, 78, 6, 73, 89] were proposed to prefetch cache lines by precomputing load addresses. They often relied on another thread running on the same processor (*e.g.*, SMT or Multicore) to achieve their goal. As such, they could diminish the return if they compete the same resources needed by their master thread. Similar to software prefetching, these techniques usually run a stripped-down slice of the original program to precompute miss addresses and bring in those data in time. An event-driven helper thread [37] is proposed to emulate a hardware prefetcher on an idle CPU.

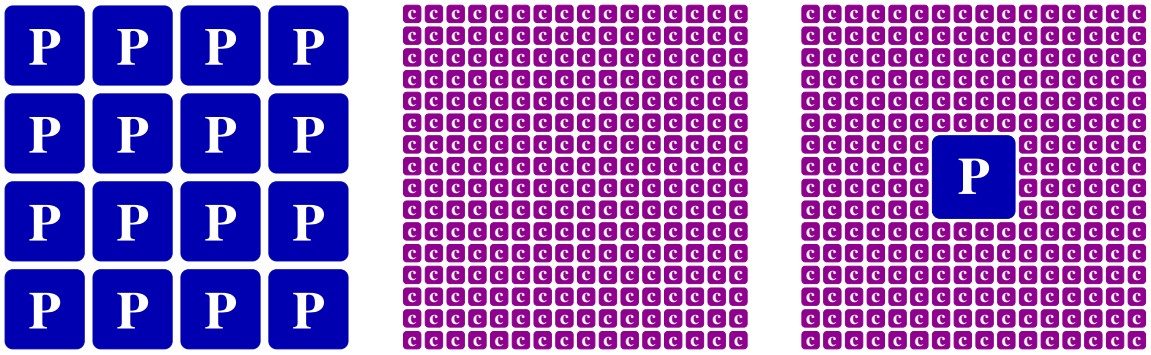
CHAPTER III

EXTENDING AMDAHL'S LAW FOR ENERGY-EFFICIENT COMPUTING IN THE MANY-CORE ERA

In 1967, Gene Amdahl proposed an often overlooked law of scaling: Sequential computation of a program largely limits the maximum achievable speedup [5]. This implies that any nonparallel execution or intercore communication will rapidly diminish the performance scalability for parallel applications regardless of the amount of additional computation resources. A simple, yet insightful, observation, Amdahl's law continues to serve as a guideline for parallel programmers to assess the upper bound of the attainable performance.

Unfortunately, beyond performance, computer architects face another grand challenge: energy efficiency. Architects should carefully design a future many-core processor so that its power consumption does not exceed its power budget [87]. For example, a 16-core processor with each core consuming an average of 20 watts will lead to 320 watts total power when all cores are active. This level of consumption can easily exceed a single-chip power budget. In other words, the amount of power that each core consumes will dictate the number of cores that architects can integrate on-die. Apparently, power is becoming more critical than performance in scaling up many-core processors. Thus, before integrating a large number of cores on-chip to provide desired performance and throughput, architects must maximize the power efficiency of each core.

Tackling these new design challenges requires extending Amdahl's law to account



(a) P^* (b) c^* (c) P+c^*

Figure 1: Many-core design styles.

for implications of power scalability in the coming many-core era. As the original Amdahl's law demonstrates, a simple analytical model can provide computer architects with useful insights. By using simple analytical models at the early design phase, architects can easily understand energy-efficiency limits, some feasible many-core design options, and future directions for making many-core processors more scalable.

3.1 Many-Core Design Styles

For this study, future many-core architectures are classified into three types. The first is a symmetric many-core processor that simply replicates a state-of-the-art superscalar processor on a die, as in Figure 1(a). High-end multicore processor vendors such as Intel and AMD use this approach. It is flexible and general enough to run different processes simultaneously while providing the best single-thread performance. Additionally, it can run independent threads spawned from one process to improve the performance of a single application. We use P to represent a single state-of-the-art superscalar processor and P^* to represent this type of many-core design style.

As Figure 1(b) shows, the second design style is a symmetric many-core processor that replicates a smaller, yet more power-efficient, core on a die. Embedded many-core processors, such as PicoChip [29], Connex Machine [43], and TILE64 [120], use

this approach. The performance of a processing core using this approach is not as high as that of a state-of-the-art superscalar processor. However, architects can integrate more processing cores on a die using this approach; thus the aggregate on-chip performance might be comparable to $P*$. We use c to denote a smaller, more power-efficient processing core and $c*$ to represent this many-core design style.

The third design style, shown in Figure 1(c), is an asymmetric many-core processor that contains many efficient cores ($c*$) and one full-blown processor (P) as the host. The Sony-Toshiba-IBM (STI) Cell Broadband Engine [52] is an example of such a heterogeneous implementation. This design style lacks the flexibility to run different processes simultaneously. Nevertheless, the single-thread performance on the host processor should be high because it guarantees state-of-the-art sequential performance for certain applications. Moreover, it provides highly parallel performance when the efficient cores are in use. We use $P+c*$ to represent this design style.

3.2 Augmenting Amdahl's Law

While Amdahl mainly focused on performance scalability back in the 1960s, we are more interested in the power scalability or energy efficiency of future many-core processors. Here, we develop analytical power models of each design and formulate metrics to evaluate energy efficiency on the basis of performance and power models.

3.2.1 Models for $P*$

According to Amdahl's law, the formula for computing the theoretical maximum speedup (or performance) achievable through parallelization is as follows:

$$Perf = \frac{1}{(1-f) + \frac{f}{n}}, \tag{1}$$

where n is the number of processors, and f is the fraction of computation that programmers can parallelize ($0 \leq f \leq 1$).

To model the power consumption for a P^* many-core processor, we introduce a new variable, k, to represent the fraction of power the processor consumes in idle state ($0 \leq k \leq 1$). We assume that one superscalar processor in active state consumes a power of 1. By definition, the amount of power one full-blown processor consumes during the sequential computation phase is 1, while the remaining $(n-1)$ full-blown processors consume $(n-1)k$. Thus, during the sequential computation phase, P* consumes $1+(n-1)k$. For the parallel computation phase, n full-blown processors consume n amount of power. Because it takes $(1-f)$ and f/n to execute the sequential and parallel code, respectively, the formula for average power consumption (denoted by W) for a P^* is as follows:

$$\begin{aligned} W &= \frac{(1-f) \times \{1+(n-1)k\} + \frac{f}{n} \times n}{(1-f)+\frac{f}{n}} \\ &= \frac{1+(n-1)k(1-f)}{(1-f)+\frac{f}{n}}. \end{aligned} \tag{2}$$

Now, we can model *performance per watt* (*Perf/W*), which represents the performance achievable at the same cooling capacity, based on the average power (W) in Equation (2). This metric is essentially the reciprocal of energy, because *performance* is defined as the reciprocal of execution time. Because *Perf/W* of single-core execution is 1, *Perf/W* benefit of a P^* is expressed as

$$\begin{aligned} Perf/W &= \frac{1}{(1-f)+\frac{f}{n}} \times \frac{(1-f)+\frac{f}{n}}{1+(n-1)k(1-f)} \\ &= \frac{1}{1+(n-1)k(1-f)}. \end{aligned} \tag{3}$$

In addition to *Perf/W*, we can model *performance per joule (Perf/J)*, a metric to evaluate the performance achievable within the same battery life cycle or, more specifically, energy. *Perf/J* is equivalent to the reciprocal of *Energy-Delay Product* [40]. Using Equation (1) and Equation (3), *performance per joule* is as follows:

$$Perf/J = \frac{1}{(1-f) + \frac{f}{n}} \times \frac{1}{1 + (n-1)k(1-f)}.$$

3.2.2 Models for c^*

The performance model of a c* many-core processor has been a topic of Mark Hill and Michael Marty's recent research [51]. This model assumes that one larger core consumes the same amount of die area that several smaller cores consume. We slightly modified this performance model to accommodate arbitrarily sized cores. To model the performance difference between a full-blown processor (P) and an efficient core (c), we introduce a new variable, s_c. This variable represents the performance of an efficient core, normalized to that of a full-blown processor ($0 \leq s_c \leq 1$). Since the performance of each efficient core is s_c, the formula for calculating the performance model of a $c*$ is as follows:

$$Perf = \frac{s_c}{(1-f) + \frac{f}{n}}.$$

To model the power consumption of a $c*$, we need two new variables: w_c and k_c. The first variable represents the relative power consumption of an active efficient core to that of an active full-blown processor ($0 \leq w_c \leq 1$); the second represents the fraction of idle power of an efficient core normalized to the overall power consumption of the same core ($0 \leq k_c \leq 1$). During the sequential computation phase, one efficient core in active state consumes w_c, and all idle cores consume $(n-1) \times w_c \times k_c$. During the parallel computation phase, all efficient cores consume $n \times w_c$. Because it takes

$(1-f)/s_c$ and $f/(n \times s_c)$ to perform sequential and parallel computation, respectively, the average power consumption by a c^* is

$$W = \frac{\frac{1-f}{s_c} \times \{w_c + (n-1)w_c k_c\} + \frac{f}{ns_c} \times nw_c}{\frac{1-f}{s_c} + \frac{f}{ns_c}}$$
$$= \frac{w_c + (n-1)w_c k_c (1-f)}{(1-f) + \frac{f}{n}}.$$

Thus, the following equations can represent *Perf/W* and *Perf/J*:

$$Perf/W = \frac{s_c}{w_c + (n-1)w_c k_c (1-f)}$$

and

$$Perf/J = \frac{s_c}{(1-f) + \frac{f}{n}} \times \frac{s_c}{w_c + (n-1)w_c k_c (1-f)}.$$

3.2.3 Models for $P{+}c^*$

Hill and Marty have also studied the performance model of a $P{+}c^*$ many-core processor [51]. We slightly modify this performance model. Executing the sequential code at the host processor (one P) takes $(1-f)$, whereas executing the parallel code using the efficient cores takes $f/(n-1)s_c$. (A $P{+}c^*$ many-core processor contains one P and $(n-1)$ c cores.) Note that we assume the host processor to be idle while the efficient cores are executing the parallel code. Thus, the formula for computing performance improvement using a $P{+}c^*$ is as follows:

$$Perf = \frac{1}{(1-f) + \frac{f}{(n-1)s_c}}.$$

During the sequential computation phase, the amount of power the full-blown processor consumes is 1, and the amount the efficient cores consumes is $(n-1)w_c k_c$. During the parallel computation phase, its full-blown processor consumes k, while the efficient cores consume $(n-1)w_c$. Because executing sequential and parallel code takes $(1-f)$ and $f/(n-1)s_c$, the average power is

$$W = \frac{(1-f)\{1+(n-1)w_c k_c\} + \frac{f}{s_c}\left\{\frac{k}{n-1} + w_c\right\}}{(1-f) + \frac{f}{(n-1)s_c}}. \tag{4}$$

Consequently, the *Perf/W* of a *P+c** is expressed as

$$\begin{aligned} & Perf/W \\ &= \frac{1}{(1-f)\{1+(n-1)w_c k_c\} + \frac{f}{s_c}\left\{\frac{k}{n-1} + w_c\right\}} \end{aligned} \tag{5}$$

and the *Perf/J* of a *P+c** as

$$\begin{aligned} & Perf/J \\ &= \frac{1}{(1-f) + \frac{f}{(n-1)s_c}} \times \\ & \frac{1}{(1-f)\{1+(n-1)w_c k_c\} + \frac{f}{s_c}\left\{\frac{k}{n-1} + w_c\right\}}. \end{aligned} \tag{6}$$

3.2.4 Power-Equivalent Models

Because the limited power budget is one of the most critical design constraints, comparing different designs without considering the single-chip power budget is meaningless.

Two main factors limit power growth on a single chip: power supply and power density. Power supply is proportional to the energy cost for sustaining machines in data centers as well as a concern for the battery life of portable devices. Power density pertains to the extra complexity and cost of thermal control mechanisms. From the power budget perspective, take, for example, a full-blown processor and an efficient core that consume 20 W and 5 W, respectively. Given a 160-W maximum power budget, we can integrate only eight full-blown processors or 32 efficient cores on a single die. Thus, to perform an apples-to-apples comparison for a given power budget envelope, we developed power-equivalent models by converting the number of cores of a c^* or $P + c^*$ to an equivalent number of full-blown processors of a P^*.

Let W_{budget} be the single-chip power budget and n_{P^*} be the maximum number of full-blown processors we can implement on a P^* die. Because the amount of power consumption by a full-blown processor is modeled as 1, n_{P^*} full-blown processors on a die can consume up to n_{P^*}. Therefore, the maximum number of full-blown processors on a P^* is $n_{P^*} = W_{budget}$.

Conversely, n_{c^*} cores of a c^* consume power up to $n_{c^*} \times w_c$, which should be less than or equal to W_{budget}. So, the maximum number of efficient cores on a c^* is $n_{c^*} = W_{budget}/w_c$.

Similarly, n_{P+c^*} cores of a $P+c^*$ consume power up to $1 + (n_{P+c^*} - 1)w_c$. Again, a single-chip power budget, W_{budget}, constrains the number of cores that an architect can implement on a chip. Consequently, the maximum n_{P+c^*} is

$$n_{P+c^*} = 1 + \frac{W_{budget} - 1}{w_c}.$$

Using these equations, we can uniformly represent and compare *performance*, *Perf/W*, and *Perf/J* of each many-core style with respect to a single-chip power budget.

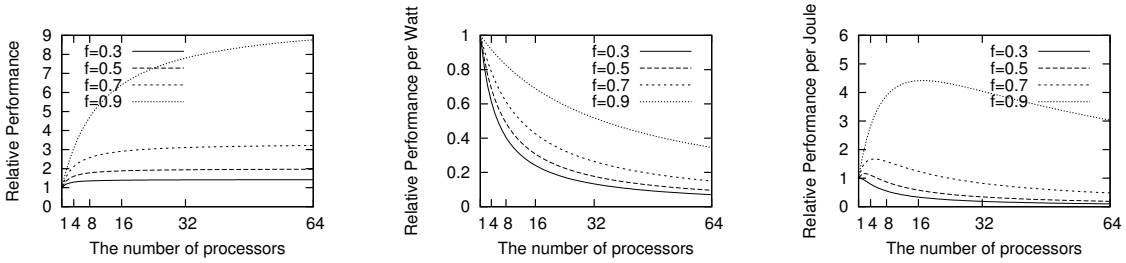
3.3 Evaluation

3.3.1 Evaluation of a P^*

Figure 2(b) shows the *Perf/W* of a P^*. Unfortunately, parallel execution on a P^* consumes much more energy than sequential execution to complete the task. In the ideal case of $f = 1$, in which we can parallelize the entire code, P^* can achieve the maximum *Perf/W* — that is, 1. In other words, a sequential execution and its parallel execution version will consume the same amount of energy only when the performance improvement through parallelization scales linearly. Otherwise, a P^* must dissipate more energy to finish the same task. This occurs because performance does not scale linearly, as Figure 2(a) shows, but the amount of idle power does scale linearly with the number of cores.

Another interesting implication of this outcome addresses battery life. If one wants to optimize the system for a longer battery life, it is better to run several processes on different cores rather than parallelize each process and time-multiplex multitask them. Although the number of processes is less than the number of cores, spawning as few threads as possible so that different processes can run simultaneously is more power efficient. This improved efficiency is because *Perf/W* becomes worse as the number of cores increases. Furthermore, this result implies that maximizing and balancing parallelization among processors is also important, not only for higher performance but also for power-supply efficiency and extended battery life. However, no matter how well the code is parallelized or its performance scales, parallelization on a P^* many-core processor will always consume more energy unless the parallel performance scales perfectly linearly.

Figure 2(c) shows the *Perf/J* of a P^*. The evaluation result demonstrates that, if the performance of a parallelized application scales well, one can expect performance improvement at the same energy budget. In other words, a P^* can extract greater performance when running embarrassingly parallel applications given the same amount



(a) *Performance* (b) *Performance per watt* (c) *Performance per joule*

Figure 2: Performance, performance per watt, and performance per joule of a P^* ($k = 0.3$).

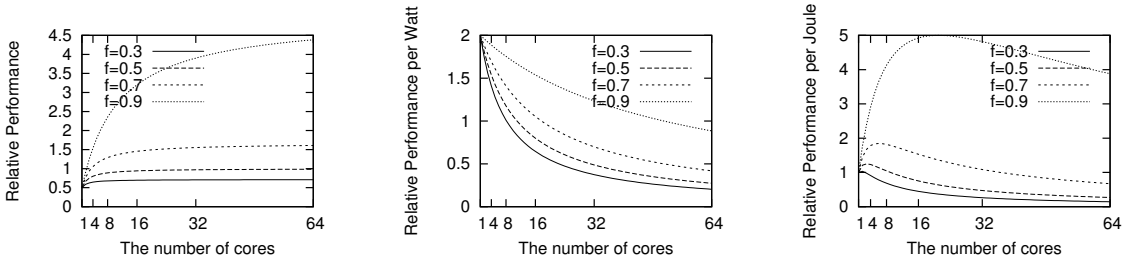
of energy. For example, when $f = 0.9$ and $k = 0.3$, a 16-core P^* can achieve more than four times speedup than a single-core processor using the same amount of energy.

However, parallelization on a P^* does not always lead to better *Perf/J*, as Figure 2(c) shows. For example, an application, half of which we can parallelize ($f = 0.5$), loses energy efficiency if it runs on eight full-blown processors. This means that, from both the *Perf/W* and *Perf/J* perspectives, efforts to parallelize applications that can not be parallelized well might not be useful at all.

Another interesting observation is the existence of an optimal number of cores to achieve the best possible *Perf/J*. So, if one is particularly interested in tuning a system for this metric, dynamic monitoring and adaptively adjusting the system will be helpful. For example, given a 32-core P^*, it is wise to enable only 17 full-blown processors when running an application with $f = 0.9$ — that is, 90 percent of it can be parallelized. In this case, it is best to completely shut off the remaining 15 full-blown processors to suppress unnecessary idle energy consumption.

3.3.2 Evaluation of a c*

To evaluate the performance and power consumption of a c^*, we must model the relationship between the performance and the size of a core. To do this, we use Fred Pollack's performance efficiency rule [101]. It states that, given the same process



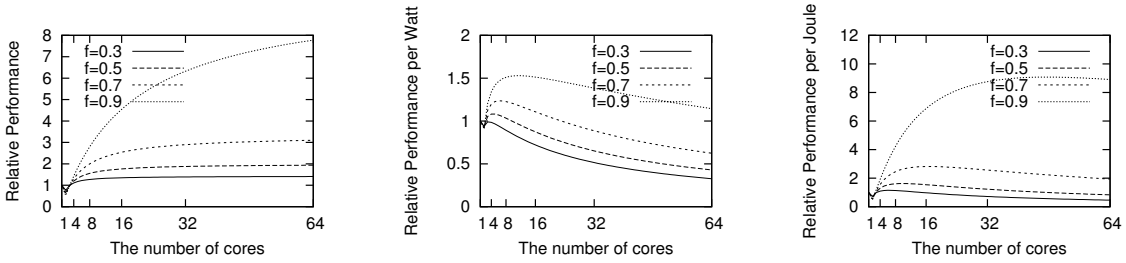
(a) *Performance* (b) *Performance per watt* (c) *Performance per joule*

Figure 3: Performance, performance per watt, and performance per joule of a c^* ($s_c = 0.5, w_c = 0.25$, and $k_c = 0.2$).

technology, the state-of-the-art processor provides 1.5 to 1.7 times higher performance and consumes 2 to 3 times the die area compared with its previous-generation counterpart. This means that a processor that consumes T times more transistors can provide only $\sqrt{T}$ times higher performance. On the other hand, the rule also implies that the processor is $\sqrt{T}$ times less efficient in terms of area. Another rule of thumb used in this evaluation is that the amount of power consumption of a core is proportional to the number of transistors it contains.

Figure 3 shows the analytical results of a c^*. In this analysis, we assume that each efficient core c has one-fourth the number of transistors of a full-blown processor P. We then model the amount of power consumption of this efficient core as one fourth of that of a full-blown processor ($w_c = 0.25$). We also assume the performance of the efficient core as one half of that of a full-blown processor ($s_c = 0.5$) and its fraction of power to be 20 percent ($k_c = 0.2$).

Figure 3(a) shows that the maximum speedup of this c^* is not as high as that of P^*. The primary reason is that the sequential performance of an efficient core is lower. As Amdahl's law says, sequential performance strictly limits the maximum speedup, and a c^* design quickly levels off the speedup. Figure 3(b) shows that, when the number of cores is small, a c^* consumes less energy than a single-core, full-blown



(a) *Performance* (b) *Performance per watt* (c) *Performance per joule*

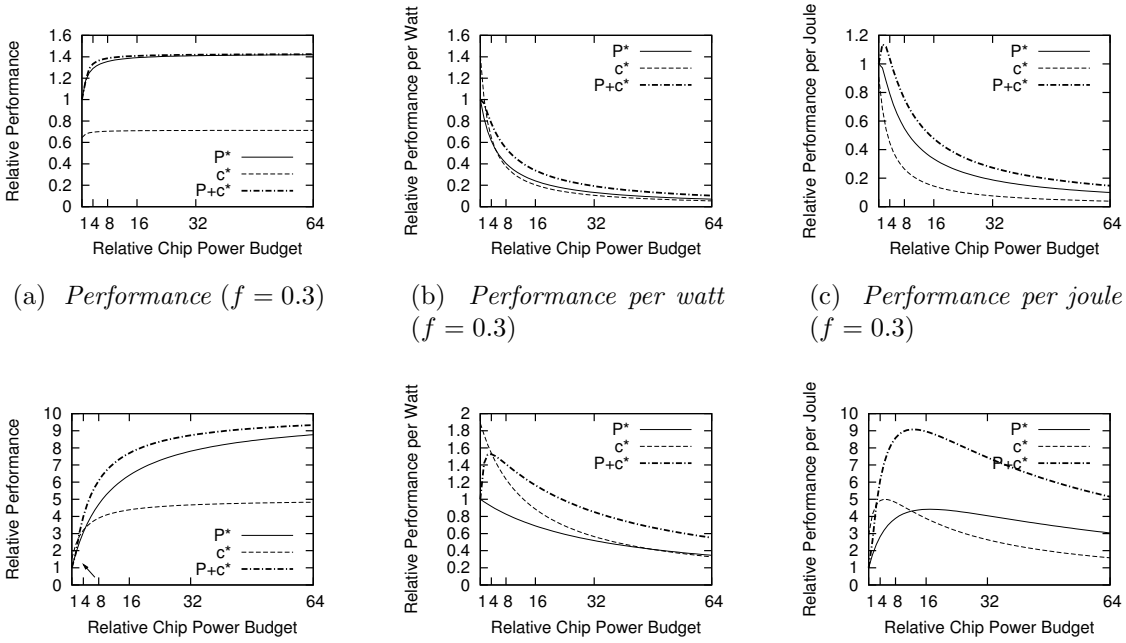
Figure 4: Performance, performance per watt, and performance per joule of a $P{+}c^*$ ($k = 0.3, s_c = 0.5, w_c = 0.25$, and $k_c = 0.2$).

processor baseline. This occurs mainly because the performance-to-power ratio of an efficient core is better than that of a full-blown processor. Unfortunately, as the number of cores increases, the amount of energy consumption becomes higher than that of a single-core full-blown processor baseline. Furthermore, Figure 3(c) shows that the *Perf/J* of a c^* is not good either, unless the application is embarrassingly parallel — that is, it has high f values. This means that performance saturation is the major contributor that leads to a low *Perf/J*.

3.3.3 Evaluation of a $P{+}c^*$

Figure 4(b) shows the *Perf/W* of a $P{+}c^*$, where s_c, w_c, and k_c are modeled as 0.5, 0.25, and 0.2, respectively. Unlike a P^* or c^*, whose *Perf/W* monotonically decreases, an optimal number of cores exists that consume the least amount of energy to execute an application. For example, we can improve the *Perf/W* of an embarrassingly parallel application ($f = 0.9$) by about 50 percent when eight cores execute it.

However, *Perf/W* becomes worse than that of a one-core baseline processor when the number of cores exceeds a certain peak. There are two reasons for this result: relative power efficiency of efficient cores and performance saturation. When the number of cores is small, the additional performance benefit gained by adding one efficient core to the host processor dominates additional power overhead, so *Perf/W*



(a) *Performance* ($f = 0.3$) (b) *Performance per watt* ($f = 0.3$) (c) *Performance per joule* ($f = 0.3$)

(d) *Performance* ($f = 0.9$) (e) *Performance per watt* ($f = 0.9$) (f) *Performance per joule* ($f = 0.9$)

Figure 5: Power-equivalent models ($k = 0.3, s_c = 0.5, w_c = 0.25$, and $k_c = 0.2$).

increases. However, once performance improvement starts to saturate, as Figure 4(a) shows, additional power overhead dominates. Thus, *Perf/W* decreases, as in Figure 4(b). In an energy-constrained environment such as embedded systems, how to spawn the optimal number of threads and turn off unused cores is an interesting topic of investigation.

Figure 4(c) shows the *Perf/J* of a *P+c**. Because of its low-latency sequential execution and energy-efficient parallel execution, a *P+c** achieves the best *Perf/J* compared with the two previous designs.

3.3.4 Evaluation of Power-Equivalent Models

In addition to evaluating each many-core design style on its own, we use power-equivalent models to perform cross-design comparisons. Because the power budget is the major design constraint, the amount of power one core consumes determines the

number of cores architects can implement on a single die. So, to compare different many-core designs, it is better to study performance and energy efficiency with the same power budget, rather than with the same number of cores.

Figure 5 shows the evaluation results with power-equivalent models. We assume each efficient core to consume one-fourth the power of a full-blown processor ($w_c = 0.25$) and its performance to be half that of a full-blown processor ($s_c = 0.5$). As Figure 5(a) and Figure 5(d) show, the power-equivalent performance of a $P{+}c^*$ is found to be highest in most cases. The power-equivalent performance of a P^* approaches that of a $P{+}c^*$ when f is small. As f increases, the difference between them grows, because a $P{+}c^*$ can have more cores at the same power budget. The power-equivalent performance of a c^* improves as f increases, as Figure 5(a) and Figure 5(d) show, but it is still the lowest among the three in most cases.

When $f = 0.9$ and the relative power budget is very low, the power-equivalent performance of a c^* is the highest (a pointer highlights this area in Figure 5(d)). In other words, in terms of performance itself, a c^* is preferable only when applications contain a huge amount of parallelism, and the system is extremely power-limited. Embedded devices designed for multimedia or data-streaming applications fall into this category.

Figure 5(b) and Figure 5(e) show power-equivalent *Perf/W*. When the relative chip power budget is small, a c^* consumes the least amount of energy to finish a task. However, when the budget is reasonably large, a $P{+}c^*$ always consumes the least amount of energy. We explain these relationships as follows: When the power budget is small, a c^* can finish the task quickly owing to more processing power. As the power budget increases, this benefit diminishes because of the performance saturation resulting from its low sequential performance. This effect continues to degrade the c^* as the budget increases and eventually causes the *Perf/W* of a c^* to become even worse than that of a P^*.

Similarly, Figure 5(c) and Figure 5(f) show that the *Perf/J* of a c^* is the highest only when the power budget is low and the task is embarrassingly parallel ($f = 0.9$). However, as the power budget increases, the *Perf/J* of a c^* many-core processor is worse than that of the other designs. Instead, a $P{+}c^*$ is the most power-scalable. Thanks to its high sequential performance along with energy-efficient parallel computation capability, it achieved the highest *Perf/J*. To better understand the design spectrum, we also performed several sensitivity studies with different sizes of c and with different relationships between the performance and the power using these models. These studies showed similar trends.

3.4 Summary

Extending Amdahl's law to take power and energy into account, this study clearly demonstrates that a symmetric many-core processor can easily lose its energy efficiency as the number of cores increases. To achieve the best possible energy efficiency, this study suggests a many-core alternative featuring many small, energy-efficient cores integrated with a full-blown processor. This study also show that by knowing the amount of parallelism available in an application prior to execution, one can find the optimal number of active cores for maximizing performance for given cooling capacity and energy in a system. To further optimally control the number of active cores in an adaptive manner, a future many-core runtime needs to have the capability of dynamic per-core power profiling and a feedback mechanism to manage thread dispatch.

CHAPTER IV

A 3D-INTEGRATED BROAD-PURPOSE ACCELERATION LAYER

As shown in Chapter III, to provide high performance given a fixed power budget, computer architects should design a many-core processor that consists of a host processor designed for the highest single-thread performance and smaller but more cores designed for power-efficiency. One approach to achieve this goal is to incorporate on-die special-purpose accelerators with general-purpose cores. From an area-efficiency standpoint, however, it is impractical to specialize and accelerate all possible applications of interest. Furthermore, implementing several application-specific accelerators on a general-purpose platform has many drawbacks, such as longer design turnaround time, higher nonrecurring engineering cost, inflexibility, and lack of backward or forward binary compatibility.

To address these issues, this study proposes a broad-purpose accelerator design called parallel- on-demand (POD). Based on complex instruction set computer (CISC) superscalar processors (such as the Intel 64, formerly known as the Intel EM64T, processor), POD revisits several massively parallel single instruction, multiple data (SIMD) designs yet focuses on novel challenges including backward and forward binary compatibility, on-chip wire delay, and efficient interaction with a superscalar host processor. Furthermore, with emerging 3D die-stacking technology, a processor can flexibly snap a POD layer on top of a conventional general-purpose processor die with the ability of virtualizing different generations of a POD design. The snap-on feature of POD also lets processor vendors optionally upgrade a product during the packaging phase.

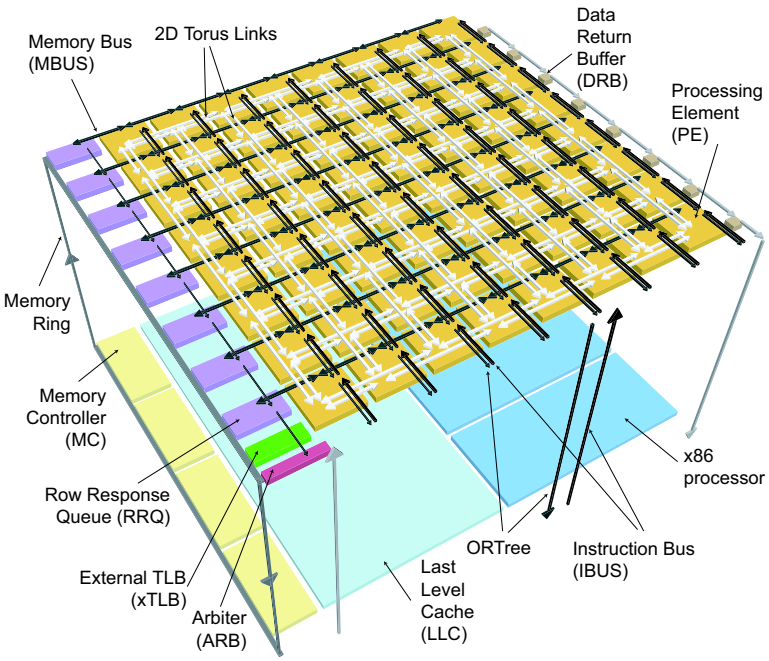


Figure 6: POD architecture in stacked die arrangement.

4.1 POD Architecture

Figure 6 is a high-level block diagram of the POD architecture, which consists of two die layers: a general-purpose processor (an Intel 64 processor) as the bottom layer and a snap-on accelerator, the POD layer. The general-purpose processor layer can be a die of a conventional multicore processor with design hooks. Depending on the target market segment, one can stack a POD layer on top at packaging time to improve performance per joule for certain applications such as high-definition multimedia, 3D games, or scientific computing. (The performance in performance per joule is defined as the inverse of execution time.)

During operation, the host Intel 64 processor fully boots a normal OS and runs every legacy application under that OS without deviating from current performance. At the instruction decode stage, the host processor might encounter a block of code written for POD to accelerate. In such a case, instructions within the block are broadcast to the POD layer through an instruction bus (IBus), of which each instruction

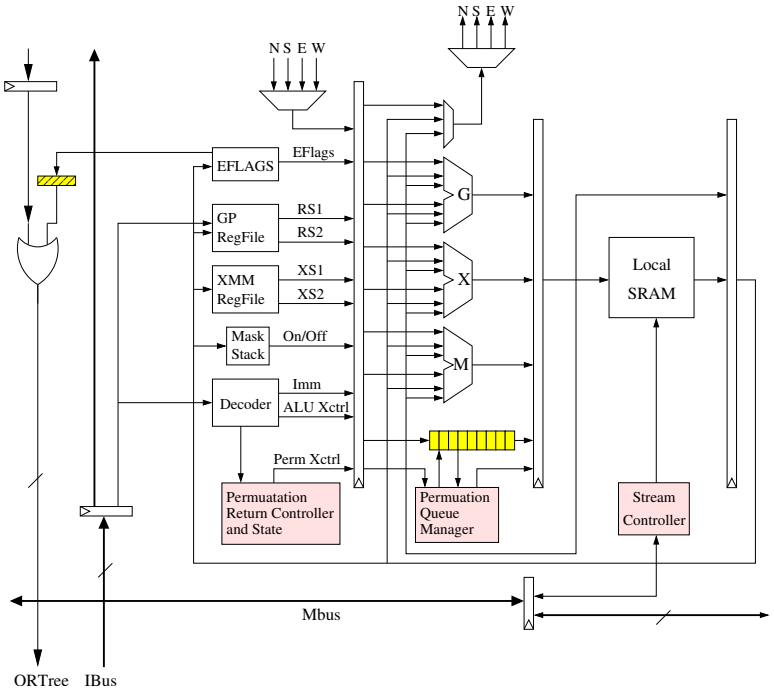


Figure 7: A processing element tile.

has the same fixed size. The processor might also broadcast 64-bit immediate values.

The POD layer is essentially a massively parallel SIMD processing element (PE) array. The target SIMD PE array is a sea of $n \times n$ tiles, where $n = 8$ in Figure 6. The PE array executes instructions broadcast through the IBus and generates a flag-tree output that is tied together logically via an OR gate (ORTree) and is routed back to the host pipeline. The computed results from the PEs can be retrieved through the data return buffer (DRB) or memory hierarchy.

4.2 Heterogeneous ISA

Each PE tile contains a high-performance arithmetic unit with its own private register file and local SRAM memory (Figure 7). To provide a baseline performance level and support a subset of the host instruction set to facilitate PE virtualization, we use an existing 128-bit streaming SIMD extension (SSE) engine from a contemporary Intel 64 processor in POD.

Although each PE will ideally be capable of decoding conventional CISC instructions for execution, this approach will require a large instruction decoder for each PE. This is less desirable when considering performance per square millimeter or performance per joule. Instead, to make each PE compact and efficient, we chose a very long instruction word (VLIW) execution model for the PE. The PE instructions, each 12 bytes wide, are broadcast from the host processor. Each 12-byte word forms a partially predecoded VLIW packet of three instructions that eliminate the CISC decoding overheads in POD. Each VLIW packet has a fixed format of one G (generic), one X (SSE), and one M (memory) pipeline instruction. Because the host processor orchestrates the execution of PE instructions, there is no need for implementing an instruction cache within the PE. Furthermore, because each PE is executing the same instruction and there is no instruction equivalent to a branch, no branch predictor or associated flush/control logic is required, which keeps the PE small and simple.

Because the host processor will orchestrate and broadcast POD instructions, we enhanced the Intel 64 instruction set architecture (ISA) to handle such heterogeneous instructions. To enable this, we added a new instruction prefix byte called *SendBits* to the existing Intel 64 ISA. When this prefix byte is encountered, it indicates that the following 12-byte word is an encapsulation of three POD operations. The host processor will then dispatch the 12-byte VLIW to the POD array. Also, we rely on the compiler or assembler to schedule and pack the three-way VLIW instructions.

Unlike conventional massive SIMD machines, the POD integrates a massive SIMD PE array with a modern out-of-order host processor. To ensure execution correctness, we should address two major challenges in the host processor: recovery from misspeculation and out-of-order dispatch of POD instructions. To support speculative execution, we need some recovery mechanism to roll the machine back to the correct architectural state. For example, Tarantula, which had relatively narrow SIMD engines attached to a superscalar processor, implemented a recovery mechanism in

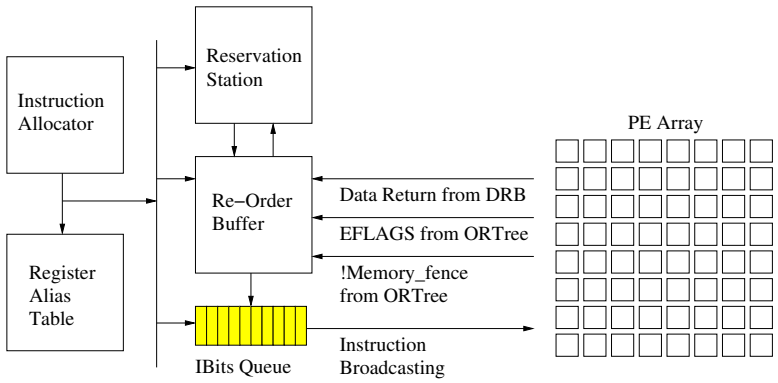


Figure 8: IBits queue in the superscalar pipeline.

each PE [33]. Unfortunately, this results in substantial overhead to both the area and power for a massively parallel SIMD engine. To simplify the PE design (and build as many PEs as possible), we design the host processor of POD to broadcast POD instructions in a non-speculative manner. In other words, the POD instructions will not be dispatched from the host processor until its preceding branches are resolved. From a performance standpoint, as long as the code that runs on the host processor does not depend on the results from the POD, this approach will not degrade performance [1].

Another issue is that the host processor might reorder the POD instructions, which might lead to incorrectness because the PE is ignorant of program order. To prevent such reordering, we strongly order the POD instructions by implementing an IBits queue along with a conventional out-of-order pipeline (Figure 8), similar to the store queue found in an out-of-order processor. When a SendBits instruction is issued, its 12-byte immediate field (encoding a VLIW POD instruction) is entered into the IBits queue. Upon the retirement of the SendBits instruction from the reorder buffer (ROB), the corresponding 12-byte immediate value is latched onto the IBus and is

[1] This is the case for all of benchmark programs that we evaluated in this study, except k-means. The host processor does not issue any data-dependent instruction that reads data updated by the POD immediately for these benchmark programs. This event is extremely rare even in k-means simulation.

broadcast to PEs.

4.3 Wire-Delay-Aware Design

To enable SIMD-style instruction execution where every PE executes each instruction at the same global clock cycle, we have two design candidates: executing an instruction immediately upon arrival to a POD row, leading to a north-south time-zone effect, or buffering each arriving instruction for sufficient time such that every PE will execute the same instruction at the same instant.

The time-zone effect can be challenging for programmers and architects to work around, as any given row will be executing instruction j while the preceding row is executing $j+1$ and the successor row is executing $j-1$. To avoid undesired complexity for programmers, architects, and compilers, we use a buffering model to eliminate the time-zone effect and enable lock-step execution.

Figure 9 shows such a model for a single column in the POD. Instructions are broadcast using the IBus and queued before being executed by the PE. For n rows, it takes $n-1$ cycles before every PE executes the instruction. The queue size shrinks monotonically as the location of a PE gets farther away from the host processor[2]. For an $n \times n$ POD, where $n = 8$, there are seven entries for the bottom-most PE while no queue is needed for the topmost PE. The delay units (D block) are inserted to delay each instruction broadcast to synchronize the SIMD execution. Similarly, when gathering results (such as EFLAGS) from PEs, the results from the PEs closer to the host processor must be delayed and wait in their queue until the farther results arrive for combining. Figure 9 depicts such queuing in the propagation paths with correct delay queues on the left side. Compared to the previous immediate execution model, there is no overhead to the PEs with this implementation except for a buffer to hold the instructions broadcast. The round-trip latency for the host processor to evaluate

[2]Although each PE is identical, programmable fuses are burned after die manufacture to set the used numbers of slots in the queue.

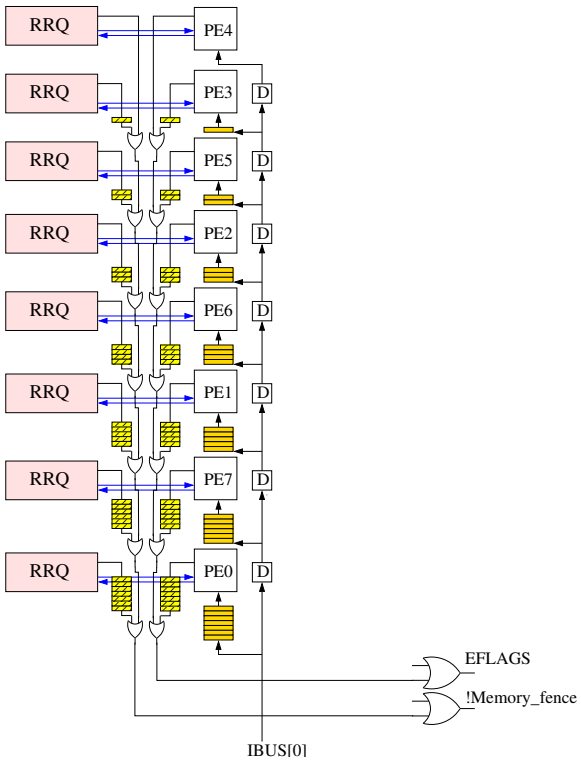


Figure 9: One detailed POD column.

the conditional loop also remains the same, which, for n rows, is $2 \times (n + f_0) + l$ cycles where $2n$ cycles are consumed for instruction and EFLAGS propagation, $2f_0$ is fan-in and fanout latency between the host processor and PEs in the first row, and l is the actual instruction latency.

Regardless of how the PEs are connected to each other, the instruction and data-value broadcast from the host processor are arranged as a fan-out tree. The bottom-most row of the PE array will receive the same instruction at the same cycle. The instruction is then passed up to the next row in sequence. This fan-out tree is implemented on a POD layer so that the number of die-to-die vias between a general-purpose processor layer and a POD layer remains unchanged (Figure 6), regardless of the number of PEs. This constant interface lets us extend a POD layer without redesigning a general-purpose layer every time we introduce a new product.

4.4 Energy-Efficient Interconnection Network

Figure 6 also shows that POD adopts a folded torus network [27]. To minimize latency and maximize packing, we designed each PE to be small enough so that signal propagation time over one PE is less than one clock cycle. Ideally, each side of a PE will be no longer in any direction than 95 percent of the wire distance in one clock cycle with all surrounding line drivers, buffers, and so forth. The ordering of the number labels inside the PEs of the leftmost column in Figure 9 indicates the north-south nearest neighbor connection pattern. In the same way, the communication links for each row are laid out in east-west direction. In addition to providing shorter links, such a layout also leads to deterministic communication latency. The significance of such deterministic latency is that we can disable communication-related logic and wires safely when they are not used. Moreover, the lock-step execution model will make the entire computation predictable and fully debuggable. There are no tricky issues such as race condition, live lock, deadlock, and so on found in normal SMP-style many-core processors.

At any given moment, only one direction (input and output) must be enabled. Because each nearest-neighbor communication pattern has a known latency, and because the communication is directed by communication instructions, the links are disabled during the course of pure computation or when links in the other direction are not being used. Furthermore, no power-hungry routers are required for this point-to-point network. All we need are a single 4:1 multiplexer and a 1:4 demultiplexer for input and output, respectively. This reduced power profile allows growth of the POD array to be limited by the average power consumption of each PE and the manufacturing die reticle. This approach contrasts with other tiled designs such as the Raw processor from MIT [117] or the TRIPS processor from the University of Texas [108] where any of the interconnection network links could be active at the same time due to dynamic routing.

Each PE can communicate with its nearest neighbor by either directly moving a register value of up to 128 bits or by transferring memory in 64-bit chunks. Because the nearest-neighbor latency for a folded torus is targeted to be two cycles or less, this allows for high-throughput computation even when the algorithm requires neighboring registers and memory values. A fully synchronized computation model allows register transfer operations to utilize full communication bandwidth without any network overhead, such as additional latency due to contention and header encoding overhead. In the case of memory transfer operations, only a half communication bandwidth can be utilized because the upper 64 bits are used to encode other information such as memory addresses.

When one PE needs to communicate to another PE in a non-nearest-neighbor fashion, we use the k-permutation routing in our interconnect design [41]. Rather than providing dynamic-wormhole-routing hardware support for a relatively infrequent operation, we use a dedicated algorithm to drive the collective POD multiplexers into a series of sweeps to migrate all data to the intended targets. These algorithms require each PE to support n hardware buffer slots (permutation queue) of the bit size matching the point-to-point link width in an $n \times n$ SIMD array.

The basic algorithm proceeds by having all PEs send messages to the east, with each message stopping when it reaches its target column. This takes $n - 1$ hops, and at the end at most n messages will be buffered in any one PE. At the end of this sweep, every message in every POD row will be in its target column. Then, the same algorithm needs to be applied to the north. This process requires as many as n^2 steps until all the buffered messages reach their target PE. As messages reach their target PE, they are processed (stored into the appropriate memory location). This two-phase sweeping algorithm ensures that for any permutation of routing, even all-to-one, all messages are delivered after a fixed latency. This fixed routing would not be an optimal solution, but each PE needs to enable only one link at the same

time, which is more energy efficient. Although the fixed latency might be high for such generic routing support, we have made the trade-off to keep nearest-neighbor communications fast, which is a much more frequent event than generic routing. To reduce the high latency of a full any-to-any communication, we also support more optimized row-only and column-only sweeps of just $n - 1$ steps for more structured communication.

4.5 Virtual Address Support

Aside from a 128-Kbyte private local SRAM dedicated to each PE, applications must also be able to communicate with the system memory through normal loads and stores. To manage this interaction, we further enhance each PE with two unidirectional memory buses (MBuses) to the main memory via an interface called the row response queue (RRQ). One bus streams data back from main memory to the PEs in the row while the other bus streams data from the PEs in the row to the main memory. Because the system memory operations of all PEs are synchronized by barrier operations, PEs can safely disable their MBus and its related logic to minimize energy consumption when they are not communicating with the system memory. The RRQ is the queuing point for transactions in both directions and, in turn, is connected to a memory ring with the last level cache (LLC) of the host processor and all memory controllers (MCs). The ring is good not only for easy arbitration and high throughput but also for a POD layer and a general-purpose layer to be merged into one system as Figure 6 shows.

System memory accesses use virtual addresses acquired from the host processor. All n^2 PEs share one pipelined translation look-aside buffer (TLB), which is external to the host processor but managed by it. The external TLB (xTLB) in Figure 6 need not be organized along traditional lines; the TLB lookup is not as critical as it is in the host processor. This allows for a superpipelined, high capacity xTLB to

be implemented, much like the texture sampler TLB in a general-purpose graphics processing unit. In the event of a fault or miss event in the TLB, the host processor is notified, and the request in the RRQ control ring is flagged as a TLB failure. When the host processor updates any TLB entry, a dedicated control signal in the RRQ control ring is set to indicate that any prior TLB failure may now retry.

4.6 Minimal ISA Modification in the Host Processor

As we discussed earlier, the host processor manages the SIMD execution inside PEs completely. To enable this, we extend the host processor ISA with five new instructions and three modified instructions. The new instructions are

- *SendBits*, to broadcast instructions to PEs;
- *GetFlags*, to obtain the return status;
- *DrainFlags*, which assures that the initial setup of a known state in the flag tree is complete;
- *SendRegister*, to broadcast a host register value to PEs;
- *GetResult*, to obtain a return buffer value from PEs without using system memory as a go-between.

The three modified host instructions are the various fence operations (load, store, and combined) that are extended for monitoring the return status of memory interface system of POD.

4.7 POD Virtualization

There are two main reasons for virtualizing the POD accelerator. The first is to provide execution compatibility for various POD sizes. Without such resilience in the design, software vendors would need to recompile their code to fully utilize all PEs for each particular platform. To virtualize the number of PEs, we hardwired six variables in POD: the number of PEs in each row, the number of PEs in each

column, the number of PEs, x- and y-coordinates of each PE, and the PE ID. The host processor can retrieve the first three values by using a CPU identification (CPUID) instruction, and each PE can retrieve all six values from the protected memory space of its local memory, which is preset at boot time. By forcing each PE to read these six values from its local memory, the same POD binary code can continue to improve performance as the number of PEs grows.

The second reason for POD virtualization is to circumvent the compatibility issue of running POD code on a platform without an integrated POD acceleration layer. There are several potential solutions; one can rely on a software or hardware binary translator. Because the POD ISA inherently originates from Intel 64 ISA, the POD code can be dynamically translated into Intel 64 ISA-compatible instructions. Three-operand POD operations can be translated into two-operand Intel 64 ISA instructions at the cost of less efficiency[3]. The fact that the number of PE registers is greater than that of the host processor can result in reduced efficiency. Communication instructions can be simply ignored as if there is only one PE. Masking operations can be converted into branch instructions. Local memory of a PE can be emulated by copying the original data into a virtual memory space of the same size as the local memory of a PE.

4.8 Physical Design Evaluation

POD aims for a 3 GHz clock speed assuming a 45 nm or better process. For this target frequency, the memory ring is capable of a bandwidth up to 192 gigabytes per second (Gbytes/s), servicing up to eight 24 Gbytes/s MCs before any modification is required. For an 8×8 POD array, with each PE containing 128 Kbytes of SRAM, connected in a torus, the peak performance of single-precision and double-precision

[3]An fma, or floating-point multiply and add, instruction needs to be translated into two Intel 64 ISA instructions, and it introduces a rounding error between the original code and the translated code.

IEEE FP operations is 1.5 Tflops and 768 Gflops, respectively.

Based on published data from Intel [46] and the die photo of its 45-nm Intel Xeon processor E5472 (formerly code-named Penryn), a single PE is estimated to occupy approximately 1.90 mm^2. In other words, the entire 8×8 PE array will amount to 122 mm^2. We assumed each RRQ, given the complexities of the various bus wirings and the ring interfaces, will be allotted an area on par with that of each PE. Therefore, the POD layer will amount to 137 mm^2. Compared to one E5472 core (22.26 mm^2), one PE consumes 9 percent of die area because it does not have a CISC decoder, an instruction cache, branch predictors, TLBs, out-of-order execution related circuits, and so on.

Given that a PE consumes roughly 9 percent of die area of a single core, we used the E5472 product specification [58] and a simple heuristic that power consumption of a certain block is proportional to the number of transistors in the block, to calculate that a PE will consume 1.37 W. Consequently, we expect 64 PEs and eight RRQs to consume 103.6 W. We additionally modeled The global interconnection power consumption using the Berkeley Predictive Technology Model [17]. We use 1.25 V, 3 GHz, and 0.5 for supply voltage, clock frequency, and switching factor, respectively. Based on these models, we expect the 96-bit IBus and two 80-bit MBuses to consume 1.90 and 3.16 W overall. This interconnect power is low mainly because all global communication links are highly pipelined, and each PE is extremely small.

In sum, we expect a POD layer to peak at 108.7 W. Assuming that a dual-core processor is bonded with a POD layer, and these two cores are fully utilized, we expect both layers to consume 148.7 W. In common scenarios, this maximum power is unlikely to be reached because the host processor will not be fully active while the POD layer is in operation. During this period, the host processor is only active for decoding encapsulated POD instructions, resolving branches for POD control, and so forth.

Table 1: POD benchmark.

Name	Description	Dataset	Type
DenseMMM	Dense Matrix-Matrix Multiplication based on Cannon's algorithm [72]	512×512 matrix	SP FP[a]
FFT	One-dimensional complex number Fast Fourier Transform	1,024 points	SP FP
IDCT	IEEE 1180 8×8 Inverse Discrete Cosine Transform used in MPEG2 decoder of MediaBench [71]	8×8 points	DP FP[b]
OptionPricing	A financial application that computes the risk of a portfolio by projecting future option prices	256 data set	SP FP
DownSampling	2:1 image down-sampling application using 1×7 low-pass filter	2112×2112 image	SP FP
Histogram	A MapReduce-style image histogram application from Phoenix application set [104]	6816×5112 image	INT[c]
LinearRegression	A MapReduce-style linear regression application from Phoenix application set [104]	4M data points in two dimensional space	SP FP
K-means	A MapReduce-style mean-based data clustering application of Minebench [91]	17,695 data points in 18 dimensional space	SP FP
CollisionDetection	A collision detection algorithm heavily used in physics simulation	128K pairs of bounding spheres	SP FP
RayCasting	A 3D rendering algorithm onto a 2D screen	A 1024×768 scene of two balls in a room	SP FP
MotionEstimation	An motion estimation (optical flow) algorithm based on the Lucas-Kanade method [76]	A 64×64 image with 3×3 windows	Mostly INT
Viterbi	The Viterbi algorithm (or Hidden Markov Model)	1024 states and 512 outputs	SP FP
Turbo	An extrinsic log-likelyhood ratios (LLR) based on BCJR algorithm [9]	Turbo code for the CDMA 2000 standard	INT

[a] A single-precision floating point application

[b] A double-precision floating point application

[c] An integer application

4.9 Performance Evaluation

We developed a cycle-level POD simulator to carry out our performance study. This simulator models every single feature of the PEs and memory subsystem, including RRQ, xTLB, and memory controller. It also accurately models on- and off-chip communication bandwidth. Off-chip DRAM bandwidth is modeled as 4×32 Gbytes/s (four on-chip memory controllers where each can provide 32 Gbytes/s bandwidth) and DRAM latency as 50 ns, unless otherwise stated. To evaluate the performance

of POD architecture, several data-parallel benchmark programs (Table 1) are ported using inline assembly because we do not have a fully-fledged compiler yet. To factor out performance improvement due to larger on-chip memory as the number of PEs increases, we assume that the aggregate size of the on-chip memory to remain the same regardless of the number of PEs for a fair comparison. For example, a PE of 1×1 POD has 8 Mbytes of local SRAM, while each PE of 8×8 POD has a 128 Kbyte SRAM only. We also conservatively assume that the access latency of 8 Mbyte SRAM is equivalent to that of a 128 Kbyte SRAM, which is three cycles for a load or one cycle for a store. In reality, the access time of an 8 Mbyte SRAM of the baseline, a 1×1 POD, will be much longer, so the actual speedup will be even greater than our results indicate.

Figure 10 shows relative performance improvement, normalized to the performance result of a 1×1 POD. As the number of PEs increases, so does the achieved gigaflops[4]. The figure shows that the trend of the speedup is approaching linear. When it runs a compute-intensive application, such as DenseMMM or OptionPricing, it achieves more than 800 Gflops. In the case of DenseMMM, very low latency (two cycles) of neighbor-to-neighbor communication makes it possible to completely hide communication overhead with computation. The reason the performance does not show an ideal linear speedup is that the efficiency of each PE decreases, although not severely, because the working set of each PE becomes smaller when we increases the number of PEs to 64.

We found that the highly communication-intensive benchmark FFT also shows fairly good speedup. To demonstrate how effectively PEs exchange data, we chose a small input size (1,024 points) so that the computation latency could not hide the communication latency. Clearly, as the number of PEs increases, communication

[4]We count each add, sub, mul, div, max, min, and cmp as one floating-point operation, and fma as two.

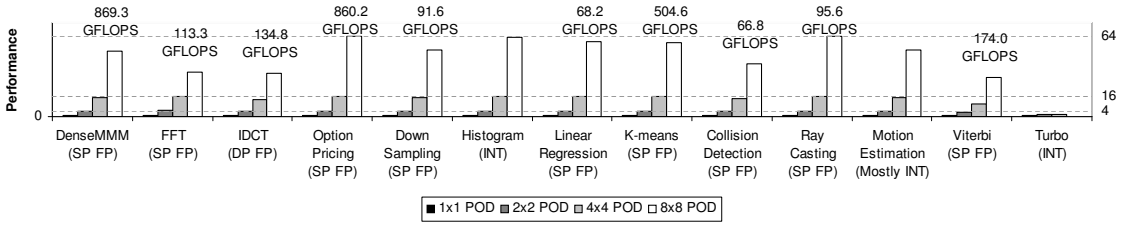


Figure 10: Relative performance (normalized to 1×1 POD).

overhead becomes dominant, but POD can still achieve good performance improvement because of high-efficiency communication architecture of POD. Similarly, we also found that another highly communication-intensive benchmark, Viterbi, shows fairly good speedup.

Unfortunately, out of 13 benchmark programs that we ported, we found that we cannot improve the performance of Turbo well. As shown in the figure, a speedup on four processing elements (PEs) is found to be only 1.65x, and that on 16 PEs is found to be even worse, 1.34x. (We failed to parallelize this algorithm over 64 PEs, which will be detailed below.) To explain such a low speedup, here we will use C code for the extrinsic LLR algorithm [94] as shown in Figure 11. As shown in the figure, this algorithm has a reasonable amount of parallelism. For example, the first 16 lines can be parallelized and even SIMDified over 16 PEs. (Note that although different lines of the C code uses different offset values, we SIMDified this code by precomputing a table that contains those different offset values and by making each PE to use its PE ID to read a different entry of this table for its local computation.) This is why we were not able to parallelize this algorithm over more than 16 PEs. Such limited parallelism is originated from the fact that the number of states that the Turbo coding algorithm of the CDMA 2000 standard maintains is only eight.

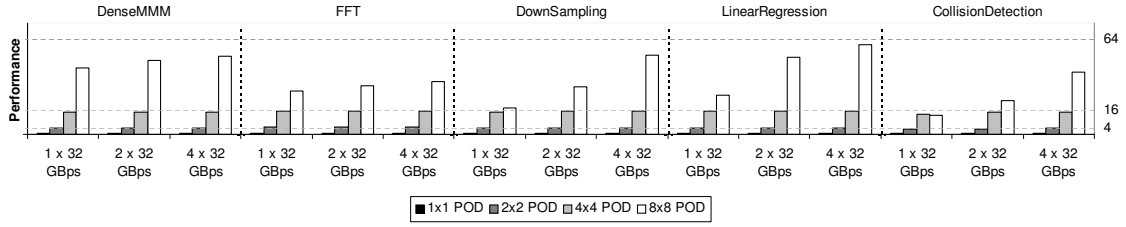
As well explained in the code, the amount of local (per core) computation is very small (a single line of C code). As a result, the overhead of frequent communication becomes relatively high, which resulted in a very low speedup shown in Figure 10.

```
1:  m00_1 = alpha[(k-1)*8+0] + beta[k*8+0];          ▷ Line 1 to 16: parallel execution on 16 cores
2:  m00_2 = alpha[(k-1)*8+1] + beta[k*8+4];
3:  m00_3 = alpha[(k-1)*8+6] + beta[k*8+7];
4:  m00_4 = alpha[(k-1)*8+7] + beta[k*8+3];
5:  m01_1 = alpha[(k-1)*8+2] + beta[k*8+5];
6:  m01_2 = alpha[(k-1)*8+3] + beta[k*8+1];
7:  m01_3 = alpha[(k-1)*8+4] + beta[k*8+2];
8:  m01_4 = alpha[(k-1)*8+5] + beta[k*8+6];
9:  m10_1 = alpha[(k-1)*8+2] + beta[k*8+1];
10: m10_2 = alpha[(k-1)*8+3] + beta[k*8+5];
11: m10_3 = alpha[(k-1)*8+4] + beta[k*8+6];
12: m10_4 = alpha[(k-1)*8+5] + beta[k*8+2];
13: m11_1 = alpha[(k-1)*8+0] + beta[k*8+4];
14: m11_2 = alpha[(k-1)*8+1] + beta[k*8+0];
15: m11_3 = alpha[(k-1)*8+6] + beta[k*8+3];
16: m11_4 = alpha[(k-1)*8+7] + beta[k*8+7];
17: if(m00_2 > m00_1) m00_1 = m00_2; ▷ Line 17 to 28: local (per row) result collection and local maximum
    calculation
18: if(m00_3 > m00_1) m00_1 = m00_3;
19: if(m00_4 > m00_1) m00_1 = m00_4;
20: if(m01_2 > m01_1) m01_1 = m01_2;
21: if(m01_3 > m01_1) m01_1 = m01_3;
22: if(m01_4 > m01_1) m01_1 = m01_4;
23: if(m10_2 > m10_1) m10_1 = m10_2;
24: if(m10_3 > m10_1) m10_1 = m10_3;
25: if(m10_4 > m10_1) m10_1 = m10_4;
26: if(m11_2 > m11_1) m11_1 = m11_2;
27: if(m11_3 > m11_1) m11_1 = m11_3;
28: if(m11_4 > m11_1) m11_1 = m11_4;
29: m11 = m_11[k-1];          ▷ Line 29 to 37: global (among rows) result collection and final computation
30: m10 = m_10[k-1];
31: m01_1 = m01_1 - m10;
32: m00_1 = m00_1 - m11;
33: m10_1 = m10_1 + m10;
34: m11_1 = m11_1 + m11;
35: if(m00_1 > m01_1) m01_1 = m00_1;
36: if(m11_1 > m10_1) m10_1 = m11_1;
37: ext[k-1] = m10_1 - m01_1 - Lu[k-1];
```

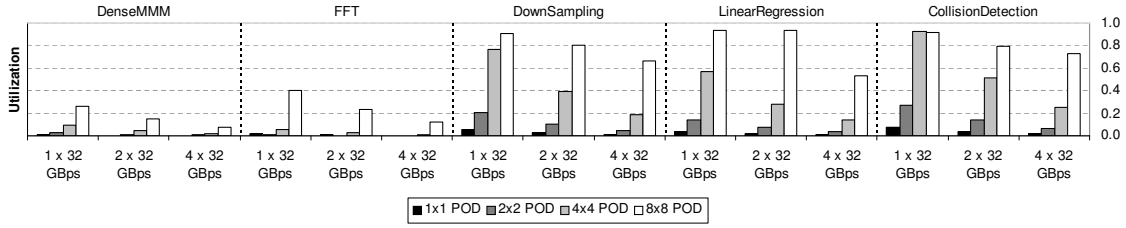
Figure 11: Extrinsic LLR algorithm [94].

Note that the torus network of the POD architecture provides 2-cycle neighbor-to-neighbor communication instructions. Thus, a local collection operation per row (Line 17 to 28) takes 6 cycles on 4x4 PEs. For the global collection among rows (Line 29 to 37), it also takes 6 cycles to transfer one local maximum value to other rows on 4x4 PEs. However, in spite of such low-latency communication operations, we need to transfer many variables (four variables for the local collection and four variables for the global collection) due to the aforementioned aggressive parallelization, which makes communication overhead relatively high.

On the other hand, another important observation is that, although a target application can be easily ported to POD for acceleration, and its performance can be improved well, off-chip memory bandwidth can still be the greatest performance



(a) Relative performance with different memory bandwidth



(b) Corresponding memory bandwidth utilization

Figure 12: Effect of off-chip memory bandwidth.

bottleneck in the future many-core era. Figure 12 shows the simulation results of five memory bandwidth-sensitive applications. The figure shows the performance improvement of different POD configurations with different off-chip memory bandwidth. Here, $x \times 32$ Gbytes/s means that the system has x on-chip memory controllers and channels, and each memory controller can support up to 32 Gbytes/s. As Figure 12 shows, memory bandwidth can be a serious bottleneck with on-chip many-core architectures when the number of cores is large. In these simulations, larger memory bandwidth improves the performance of an 8×8 POD rather substantially in several benchmark programs. Once memory bandwidth is saturated, overall performance does not scale or can even degrade, as we can observe from the simulation result of the CollisionDetection with 1×32 Gbytes/s off-chip memory bandwidth. When the performance does not scale well due to saturated off-chip memory bandwidth, an intelligent mechanism to detect it and to disable some of the PEs to balance computation power and memory bandwidth would be an improvement.

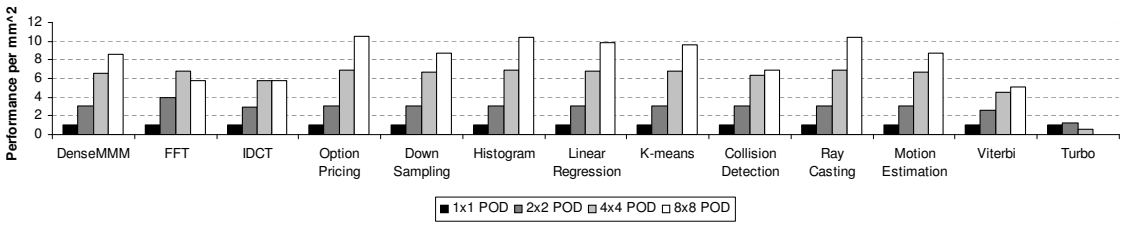


Figure 13: Relative performance per mm^2 (normalized to 1×1 POD).

To demonstrate the efficiency of POD as a snap-on accelerator, we used the following metrics for evaluation: performance per square millimeters and performance per joule. Figure 13 shows the relative area efficiency of POD, represented by performance per square millimeters. Because one PE consumes approximately 9 percent of the host processor's area, performance per square millimeters will keep improving as the number of PEs increases, given the overall performance scales. In an SMP-style many-core processor, this metric is at most one because n cores consume n times space, and it can achieve n times speedup at the best scenario of linear speedup.

The performance per joule metric represents achievable speedup given a fixed energy budget such as battery lifetime, and it is equivalent to a reciprocal of energy-delay product [40]. Figure 14 shows that we can easily improve performance per joule with parallelization. All cases show that the larger the PE array, the more improvement this metric can achieve. Furthermore, performance per joule scales super-linearly. In the case of an SMP-style many-core system, execution time can be reduced by a factor of n times at most while it consumes n times more power. Thus, the performance per joule of an SMP-style many-core system scales linearly in the best scenario of linear speedup. The additional performance per joule benefit of POD comes from the efficient design of PEs compared to the host processor.

In addition to computation efficiency, POD has an efficient communication architecture. A major problem with a conventional tile-based many-core architecture is that we expect the interconnection architecture to consume a high percentage of space

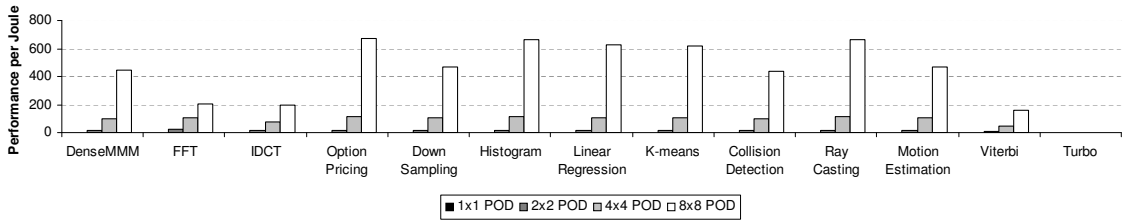


Figure 14: Relative performance per joule (normalized to 1×1 POD).

and unsustainable energy. For instance, the Raw processor consumes 40 percent of die area in its crossbar and buffers [117]. On the other hand, according to estimates reported for Raw and TRIPS, the interconnection-related circuits and wires can consume approximately 36 and 25 percent of the overall chip power [66, 98]. Out of these consumptions, input buffers and crossbar consumed 61 and 68 percent [124]. Without any software hint, it is difficult to apply clock-gating to these systems because the communication patterns are completely nondeterministic [11].

In contrast, each PE in POD requires only one multiplexer and one demultiplexer for its 2D torus network. A PE requires neither crossbars nor input/output buffers of a conventional packet-switched 2D torus network. The only buffer required is the permutation queue, which is activated only during non-nearest-neighbor communication. In addition to this efficiency, a software-directed communication mechanism along with its deterministic latency makes the 2D torus traffic predictable. Thus, we can use clock-gating to further suppress the energy consumption of wires. Figure 15 shows the active time of inter-PE point-to-point links with respect to the overall execution time for PODs with difference sizes. Although Viterbi is a well-known communication-intensive application, the synchronized computation and communication model of POD makes it possible to disable its point-to-point links for more than 85 percent of the total execution time, which minimizes the energy consumption of communication links. (The high utilization of Turbo is caused by the aforementioned aggressive parallelization, which did not lead to a high speedup due to the

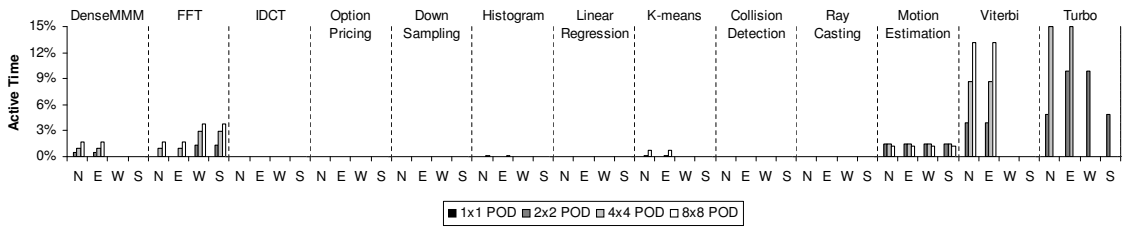


Figure 15: 2D torus network active time.

nature of the algorithm.)

4.10 Summary

In this chapter, we propose a 3D-integrated broad-purpose accelerator architecture called POD to address looming challenges and design constraints including power efficiency, on-chip wire delay, efficient interaction with a superscalar host processor, binary compatibility, and extensibility. POD integrates a SIMD array accelerator into a superscalar host processor with minimally new instruction support. The POD design also has the advantage that it is a substantially good fit for implementing highly parallel versions of CISC instruction sets without having to pay the CISC penalty in power and complexity on every processing element. In other words, as one scales a POD array to larger sizes, the inefficiencies of the base architecture will be largely hidden, which makes the designs compatible with existing ISAs and power/performance competitive and flexible with more special-purpose acceleration engines. Furthermore, by leveraging emerging 3D die-stacking technology, POD provides extensibility to a conventional processor while virtualizing the existence of an acceleration layer.

CHAPTER V

CHAMELEON ARCHITECTURE

As shown in Chapter III and Chapter IV, heterogeneous computing has become more attractive to address the growing concerns of energy efficiency and silicon area effectiveness. Small-scale heterogeneous multi-processor system-on-chips (MPSoCs) have been used in embedded systems for years [31, 8]. Meanwhile, general-purpose processor designers are also advocating such heterogeneous architectures for future multicore or many-core processors to optimize a system's energy efficiency (measured in *performance per joule*) or area effectiveness (measured in *performance per* mm^2). For example, the first generation of the IBM Cell Broadband Engine (Cell/BE) [100] integrated eight synergistic processing elements along with a superscalar PowerPC processor on the same die. CUDA [15], CTM [50], RapidMind [83], OpenCL [88], PeakStream [96], and Ct [39] serve as programming abstractions to enable heterogeneous computing based on a cooperative computing model between the central processing units (CPUs) and many-core graphics processing units (GPUs) and shield programmers from managing the complexity of these heterogeneous components. Although some heterogeneous computing platforms have their resources distributed across multiple chips, the trend of future technology is toward integrating them all onto the same die [85].

Unfortunately, an heterogeneous many-core processor has a drawback; While the host processor executes the sequential code of a parallelized workload or unparallelized legacy applications, the acceleration cores of a heterogeneous multicore become idle contributing nothing to single-thread performance while consuming area and additional power if not completely turned off. From the standpoints of area and energy

efficiency, the unused idle resources could dwarf the interests of adopting such a heterogeneous platform for general-purpose computing.

To address this under-utilization problem during sequential computation, we envision that we could better utilize these idle PE resources to accelerate the sequential execution on the host processor. In this chapter, we introduce *Chameleon*, a flexible architecture with low-cost enabling techniques to provide several dynamic operation modes for better resource allocation. Using Chameleon techniques, idle cores can be virtualized into (1) a unified last-level cache, (2) a data prefetcher, or (3) a hybrid caching/prefetching component. In addition, we propose an adaptive operation mode that adapts Chameleon among different modes to find the best possible performance. To justify the performance benefits of our Chameleon architecture, we perform a case study using POD[1] and present its hardware and power overheads as well as energy implications in our evaluation.

5.1 C-Mode: Virtualizing Idle Cores for Caching

As shown in Chapter IV, the original purpose of integrating a heterogeneous PE array onto general-purpose processor cores is to exploit data-level parallelism (DLP) for maximizing energy- and area-efficiency. We call this operation mode *A-mode* (or Acceleration mode) to differentiate it from the new modes we will introduce. Our first goal is to virtualize this idled heterogeneous PE array into additional caching space when the A-mode is not in use. This virtualization must be simple and should not affect the efficiency of the A-mode. The idea is to configure the unused local scratch-pad memories collectively into a last-level cache by using PEs' basic operations for caching control. We call this operation mode the *C-mode* (or Caching mode). Similar to the A-mode, the PEs will be responsible for decoding instructions received from the

[1]We use the exactly same architecture except that PEs are connected through a mesh network. A folded torus network mentioned in Chapter IV can be reconfigured to the mesh network using simple switches [110]

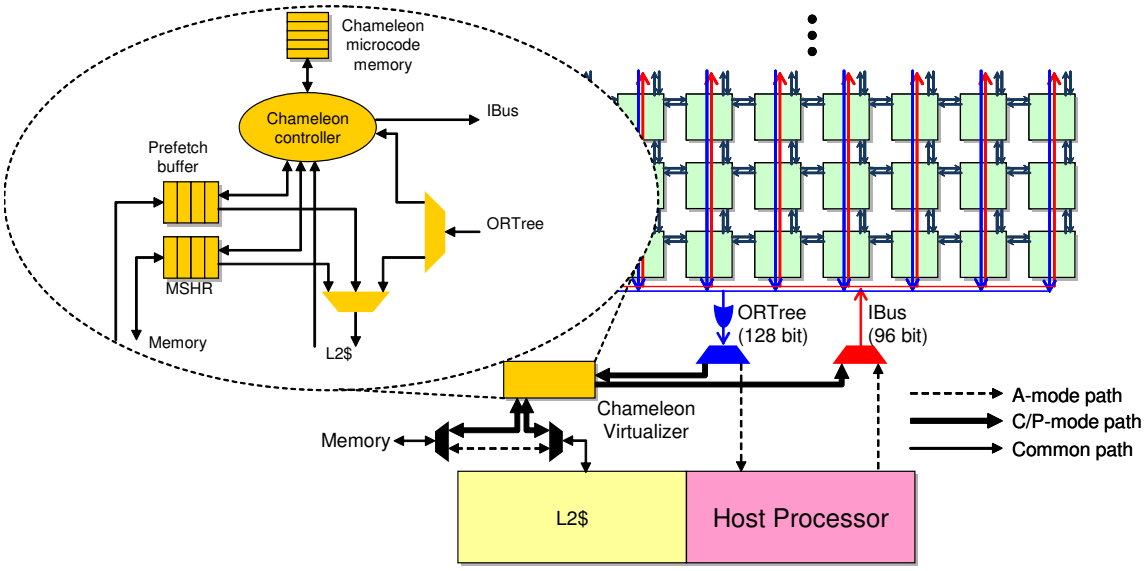


Figure 16: Chameleon Virtualizer (not scaled).

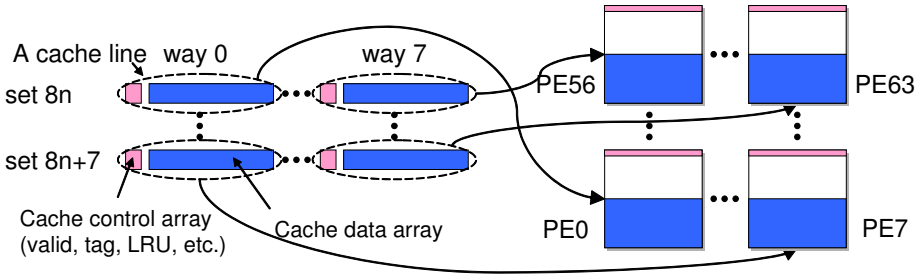
IBus, performing corresponding local computation, and routing computation results back. For example, to calculate the cache index bits, the PE is programmed to perform an SHR (logical shift-right) to eliminate the cache line offset and an AND (logical and) to mask out tag bits. Using this calculated index bits, the PE can read data from its local 128KB scratch-pad memory with a load instruction.

To control the PEs array and to have it function like a soft cache, we add a new interface between the L2 cache of the host processor and the baseline PE array. As shown in Figure 16, this new interface, called *Chameleon Virtualizer*, is in charge of orchestrating memory management operations for implementing the virtualized last-level cache using microcode stored inside the Chameleon microcode memory. The microcode is written in the original PE ISA, and it consists of tens of PE instructions. Upon a cache read miss in the L2 cache, for example, the miss address is forwarded to the Chameleon controller. Once receiving the address, the Chameleon controller forwards the miss address to the PEs via IBus and starts to broadcast a cache read microcode to the PEs. To perform a cache read, PEs perform the following tasks: (1) calculating cache index bits, (2) matching valid and tag bits, and (3) sending a

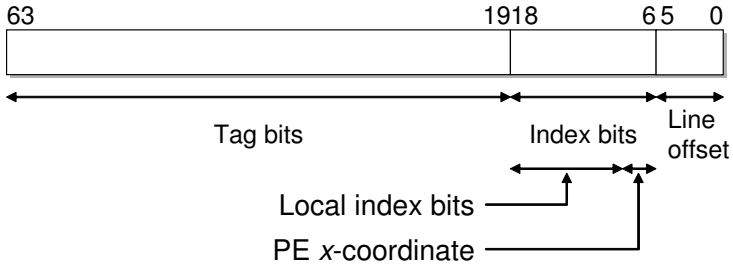
hit/miss signal to the Chameleon Virtualizer. Upon a cache hit, the hit PE has to perform the following additional tasks: (4) loading the cache line from the scratch-pad memory, (5) routing the line back to the Chameleon Virtualizer, and (6) updating the corresponding L2 LRU bits. Once the cache line reaches the Chameleon Virtualizer, it is forwarded to the L2. On the other hand, upon a cache miss, the Chameleon controller initiates an off-chip memory request through the MSHR of the Chameleon Virtualizer. The Chameleon Virtualizer also contains a prefetch buffer to support the virtualized prefetcher to be detailed in Section 5.2. Note that the overhead of the Chameleon Virtualizer is only incurred in the C-mode because the controller will be bypassed when operating in the conventional A-mode.

To facilitate the mechanisms for a cache line hitting in the PEs, we reuse the existing 128-bit wide ORTree bus, which was originally designed for obtaining the flag status of the PE array, but which is idle when operating in the C-mode. Hence, we hijack this bus to send hit/miss signals and transfer requested cache lines. However, to use the ORTree bus for such purposes, we need to add a new instruction called xferortree into the PE's ISA. This special move instruction drives a register value onto the ORTree. This new instruction requires adding a mux in each PE for selecting either the flag status (in the A-mode) or the output operand of an xferortree instruction (in the C-mode) for ORTree. On the other hand, the Chameleon Virtualizer is connected to the other end of the ORTree. Note that, in our implementation, only one PE in the same column can transfer data to the Chameleon Virtualizer at any given time, and the ORTree output value of all other PEs in the column is zero. Thus, ORTree can safely deliver the data to the Chameleon Virtualizer without being corrupted by OR operations.

In the following sections, we will address the challenges with respect to the styles of cache line layout across the PE array. We also detail these design alternatives and evaluate and quantify their trade-offs in our experiments. Furthermore, we investigate



(a) Layout



(b) Indexing

Figure 17: Way-level parallelism (8-way 4MB cache).

how we can optimize their access latency by adopting non-uniform cache architecture (NUCA) and discuss the required architectural support.

5.1.1 Design for Way-Level Parallelism

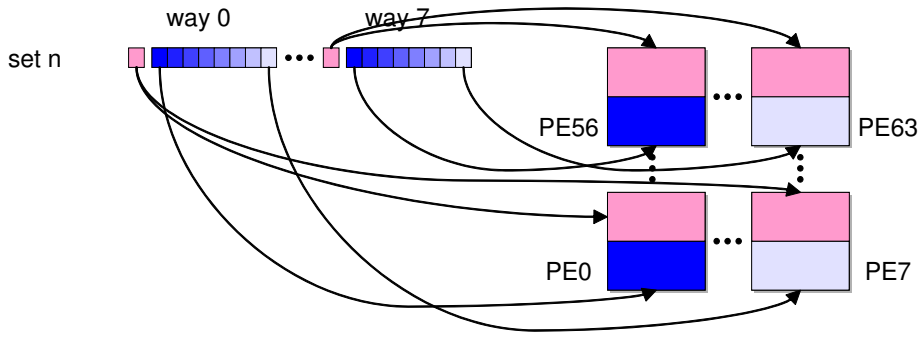
Our first design is to distribute multi-way cache lines of the same set across PEs in the same column. Figure 17 shows an example of an eight-way set-associative 4MB cache. In this example, eight cache lines (each 64 bytes) mapped to the same cache set are distributed across eight PEs in the same column (e.g., PE0, PE8, to PE56). Also shown in the figure is how to index this cache. Out of the global index bits, three least significant bits (LSBs) are used to find the x-coordinate of the target PE column. The rest of the index bits (10 bits) are used as the local index for finding the cache line from the eight local scratch-pad memories on the indexed column. (In this

design, up to 1,024 cache lines can be stored inside each PE. This will be clarified later in this section.) Mask instructions are used to disable the other 56 PEs after the x-coordinate is calculated. In this particular design, all eight active PEs on the same column will perform a tag comparison in parallel. Hence, we say this design exploits Way-Level Parallelism (WLP).

One challenge for having a functional WLP cache is how to perform LRU updates across PEs in the same column. To solve this issue, we chose to implement the *counter LRU algorithm* [61] and program Chameleon microcode to perform replacement operations. This software-based LRU replacement mechanism will read the LRU state of the hit line and broadcast the outcome back to all PEs in the same column. The PEs will then update their own LRU bits accordingly. Note that these updates simply use *subtract* and *compare* instructions already provided by the PE ISA. Although a software-based LRU may take longer than a hardware-based LRU update, we found that properly scheduled microcode can hide much of this latency.

On the other hand, this WLP design has a space overhead for cache control bits. To implement an eight-way 4MB cache with 64B line for a 64-bit host processor, we need one valid bit, one dirty bit, 45 tag bits, three LRU state bits, and a few coherence protocol bits for each cache line. These control bits amount to around 10% overhead. Thus, for N cache lines, the total storage needed will be $1.1 \times 64 \times N$ bytes, and it should fit into a 128KB scratch-pad space. Furthermore, the number of sets stored in each PE should be a power-of-two for cache indexing. This explains why each PE accommodates 1,024 cache lines in our WLP design.

In this design, once the set is determined, only one corresponding column is enabled to complete one cache operation. In other words, if the Chameleon Virtualizer can provide eight instruction streams to decode eight returning messages, we can build a virtualized eight-bank cache. To implement it, eight different IBuses and



(a) Layout

(b) Indexing

Figure 18: Way- and subblock-level parallelism (8-way 4MB cache).

eight different ORTree buses should be directly connected to the Chameleon Virtualizer instead of using fan-out tree (IBus) and fan-in OR tree (ORTree) as in the baseline processor.

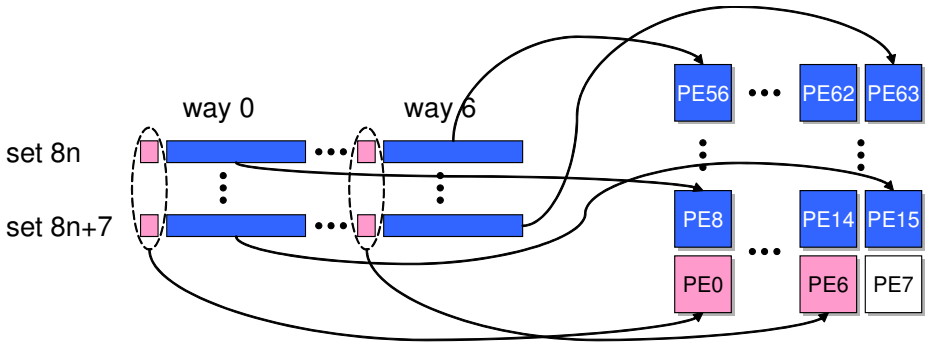
5.1.2 Design for Way- and Subblock-level Parallelism

Since the 128-bit ORTree bus and the 96-bit IBus are used for reading and writing cache lines in the WLP-style cache, it will take four and eight cycles to transfer an entire 64B cache line on the buses.[2] On the other hand, to prepare data transfer, four SIMD load instructions (or eight regular store instructions) are used to load each 16B chunk into the XMM registers (or store 8B chunk to general purpose registers),

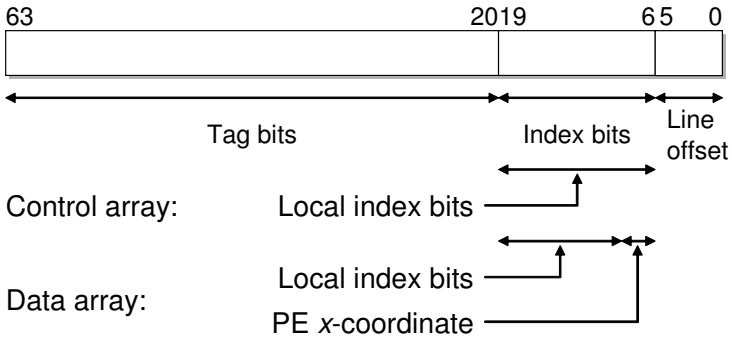
[2]The 96-bit IBus can only broadcast 64-bit data at each cycle due to instruction encoding overhead.

which adds extra overheads in accessing cache lines. This is an artifact caused by mapping one entire cache line onto a single PE as shown in Figure 17. To alleviate this issue, we investigate another design option in which a 64B cache line is split across eight PEs on the same row as shown in Figure 18. To read a cache line in this design, each PE in the same row will load an 8B subblock of the requested cache line. All eight subblocks will be routed back to the Chameleon Virtualizer simultaneously without modifying the PE microarchitecture. We call this design exploiting way- and subblock-Level parallelism (WBLP). Due to subblocking, we only need one load and one xferortree instruction for reading an entire cache line, and one 64-bit immediate broadcast and one store for writing it. In this design, the Chameleon Virtualizer is made to broadcast an immediate move operation with eight different immediate values and to retrieve eight different data return values. In this design, as in the eight-bank WLP-style cache, eight different IBuses and eight different ORTree buses should have direct connection to the Chameleon Virtualizer instead of using fan-out IBus and fan-in ORTree as in the baseline processor.

The primary challenge of such a WBLP design is the area overhead in keeping the cache control bits. As a cache line is split into eight subblocks, all eight PEs that keep a subblock of the same cache line need to have redundant valid, tag, LRU and coherence bits. Otherwise, more delay will incur for communicating this information. We found that each PE can accommodate this redundant information without sacrificing the overall cache capacity. As explained in Section 5.1.2, at most 64b of overhead is required per 64B cache line. In the WLP design, out of the 128KB scratch-pad memory per PE, 64KB is consumed by its data array, and less than 8KB is consumed by these cache control bits (i.e., each 128KB scratch-pad memory is quite under-utilized.) In the WBLP design, at most 64b overhead is required per 8B subblock. Thus, 64KB is used by its data array, and at most 64KB is consumed by the cache control array with no further implication to utilizing the maximally available cache



(a) Layout



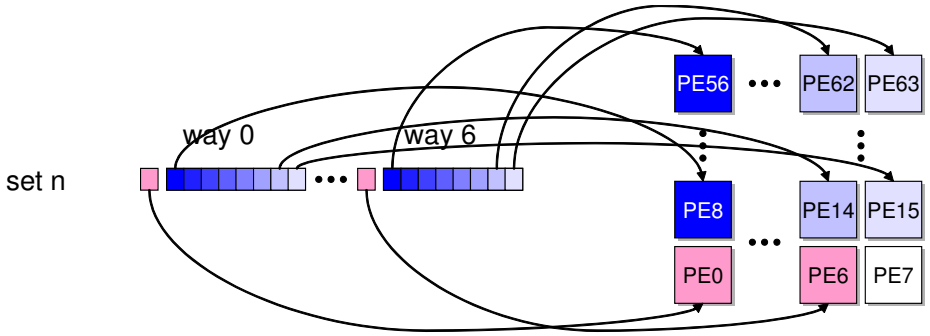
(b) Indexing

Figure 19: Decoupled WLP cache (7-way 7MB cache).

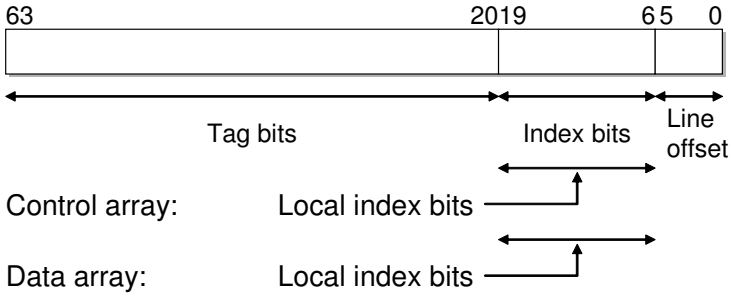
capacity.

5.1.3 Decoupled Design

The two designs discussed previously place the cache control array and data array in one PE so that each PE can locally detect whether the request is a hit or a miss and route the hit line back to the Chameleon Virtualizer. Such a local decision mechanism allows these two transfer operations to be pipelined so that the overall lookup latency can be reduced. However, as explained previously, these designs cannot utilize the memory space efficiently because the number of sets in each PE must be a power of two.



(a) Layout



(b) Indexing

Figure 20: Decoupled WBLP cache (7-way 7MB cache).

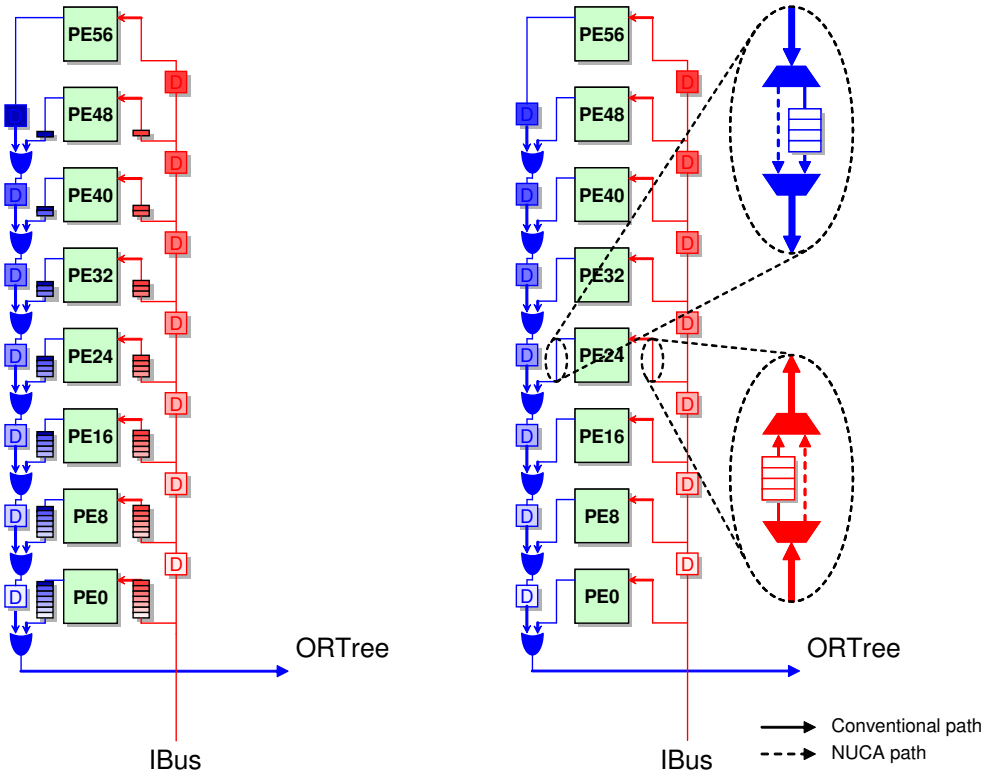
Instead, we study an alternative design style in which the cache control array and data array are spread across different PEs. In this design, the Chameleon Virtualizer needs to read the hit/miss signal first from PEs that store the cache control array, and then it needs to request the target PE that stores the hit line to route the line back to the Chameleon Virtualizer. Figure 19 and Figure 20 show such decoupled designs. As shown in the figures, the cache control array is stored in seven PEs in row 0. The Chameleon Virtualizer needs to look up these PEs' local scratch-pad memory space to see whether the requested block is a cache hit or miss. Upon a hit, it also needs to request one (decoupled WLP) or eight (decoupled WBLP) PEs out of 56 PEs (row 1 to row 7) to route the hit data array back to the Chameleon Virtualizer. In our decoupled design, PEn ($0 \leq n \leq 6$) keeps a cache control array for the data array of

PEs in row $n+1$. For example, in case of a decoupled WBLP cache (Figure 20), PE6 stores the cache control array of way 6 while the cache data array of way 6 is stored in PEs of row 7 (PE56 to PE63). Although the lookup latency of this style cache is longer than that of previous two designs, 7-way set-associative 7MB cache (total 16k sets) can be stored in 64 PEs — Each of seven PEs in row 0 stores the cache control array of 16k cache lines of each way; Each PE in other rows stores the cache data array of 2k lines (the decoupled WLP cache (Figure 19)) or the 8B subblock of 16k lines (the decoupled WBLP cache (Figure 20)). Note that these 63 PEs fully utilize their 128KB local scratch-pad memory. The only unused space is the local memory space of PE7 as shown in the figures.

5.1.4 NUCA Cache Design

In the conventional A-mode, for synchronizing the computation for each PE, a PE located at row i (where i ranges from 0 to 7) in an 8×8 PE array contains an instruction queue with $7-i$ entries as shown in Figure 21(a). Instructions are broadcast through IBus and queued prior to the execution by its designated PE. The delay units (shown as D blocks) are inserted to synchronize each instruction broadcast in a SIMD-style execution. With the instruction queue and pipelined IBus, PEs in different rows will execute the same instruction at the same cycle, fully synchronized. Similarly, a pipelined ORTree and flag queue are used to synchronize flag status globally. Such a strictly synchronized execution model keeps the architecture and its programming models simple. For example, neither the processor architects nor the programmers have to deal with complicated synchronization issues such as live-locks or deadlocks.

However, if the PEs are collectively used as a virtualized last-level cache, it will be beneficial to have non-uniform access latencies, i.e., accessing each PE row out-of-sync. As shown in previous studies [65, 56, 23], a non-uniform cache architecture (NUCA) helps reduce the average cache access latency, thereby improving the overall



(a) Conventional, synchronized model (b) Pipelined model to support NUCA

Figure 21: Execution model (column 0 only).

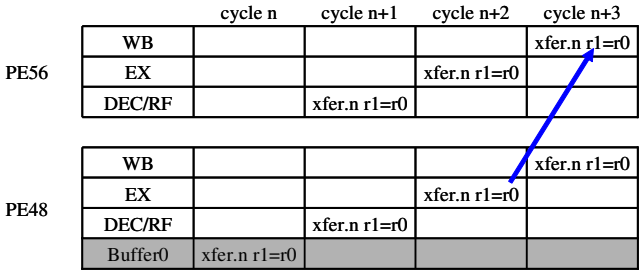
performance. As such, it will be more desirable to keep data with good temporal locality in a nearby memory bank of a large NUCA structure. Although our baseline PE array already has a partitioned array of 64 PEs that uses mesh topology,[3] it requires certain changes in the architecture to enable non-uniform latencies across PE rows. To eliminate the strictly synchronized execution nature of the baseline, the instruction and flag queues, originally designed for synchronizing their broadcasting, are bypassed when the NUCA model is enabled. As shown in Figure 21(b), the NUCA path directly bypasses and does not buffer any incoming cache access

[3]In this section, we assume a mesh network for simplicity. The mesh network can be easily reconfigured to a folded torus network using simple switches [27, 110].

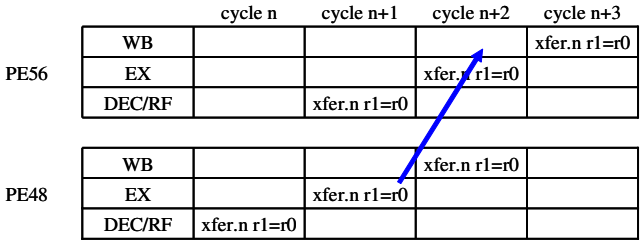
microcode instruction and outgoing requested cache lines. Consequently, in this execution model, different PEs in different rows execute different instructions at the same cycle. However, the pipelined execution model could complicate the synchronization of the ORTree values and that of northbound and southbound transfer instructions. This is what we call *time-zone effect*. Fortunately, the ORTree time-zone effect is not an issue in the C-mode because C-mode microcode uses the ORTree to obtain a requested cache line. Furthermore, the Chameleon Virtualizer is allowed to issue one memory lookup microcode at a time, so there is no concern of data corruption between distinct memory accesses.

The next problem is synchronizing the northbound transfer instruction. The *xfer.n* instruction is a special type of *move* instruction that copies a register value of a PE into a register of its northern neighbor PE. In synchronized execution (Figure 22(a)), the r0 value of PE48 reaches PE56 at cycle $n + 3$ when the xfer.n instruction being executed by PE56 is in the write-back (WB) stage. In this example, PE56 expects to have its r1 value from PE48 as the same xfer.n instruction is decoded and being executed by PE56 itself. Therefore, PE56 will set up control signals prior to the reception of the value. However, in the pipelined execution model (Figure 22(b)), the r0 reaches PE56 at cycle $n + 2$ when the same xfer.n instruction is in the EX stage. In other words, PE56 has not set up control signals to read the transferred value from PE48 and to update its r1. Without any support, the r1 of PE56 will not be correctly updated. To address this issue, we propose virtually synchronizing this xfer.n instruction by adding one more pipeline register in the northbound output mesh driver of PE48, so that the r0 value can reach PE56 at cycle $n+3$ (Figure 22(c)).

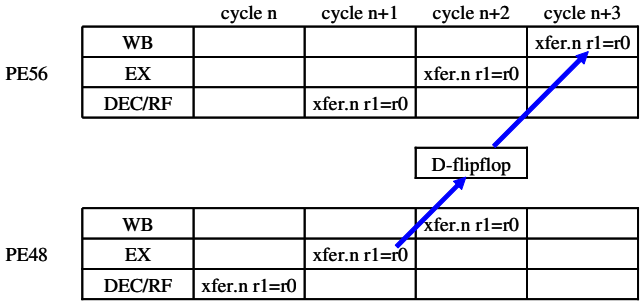
A similar problem is present in synchronizing the southbound transfer xfer.s instruction. Figure 23(a) shows the synchronized execution model. However, in the pipelined execution model (Figure 23(b)), the r0 of PE56 reaches PE48 at cycle $n + 3$. At this moment, the xfer.s is no longer in the pipeline of PE48. As such,



(a) Conventional, strictly synchronized execution model

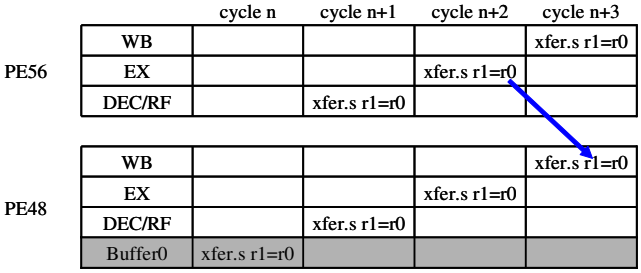


(b) Pipelined execution model: time-zone effect of xfer.n

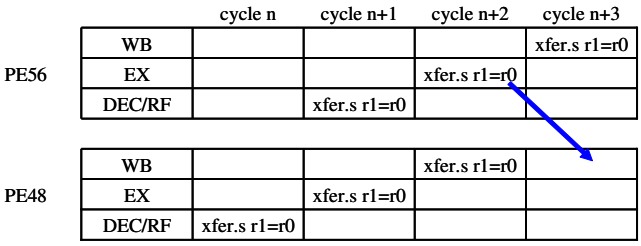


(c) Pipelined execution model: a solution for time-zone effect of xfer.n

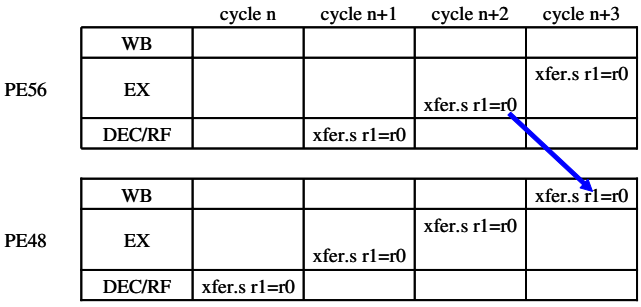
Figure 22: Conventional and NUCA execution model of a xfer.n instruction.



(a) Conventional, strictly synchronized execution model



(b) Pipelined execution model: time-zone effect of xfer.s



(c) Pipelined execution model: a solution for time-zone effect of xfer.s

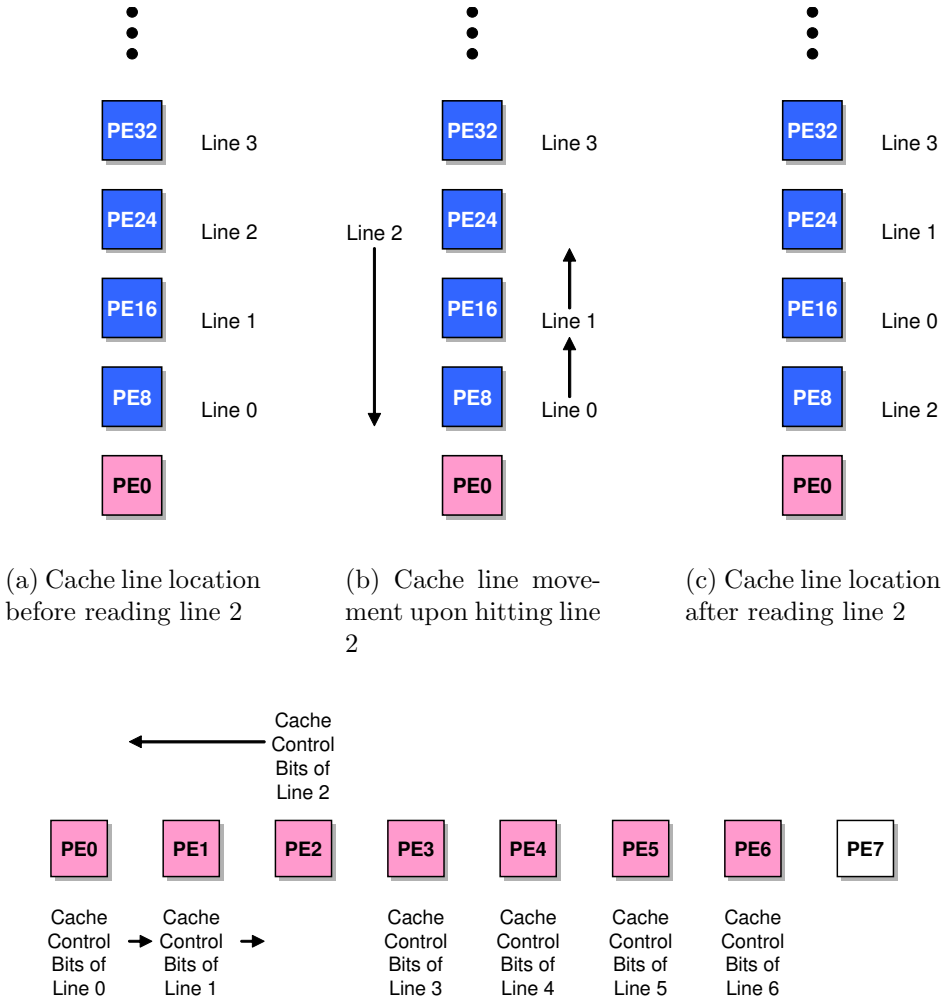
Figure 23: Conventional and NUCA execution model of a xfer.s instruction.

the r1 of PE48 will not be correctly updated. Fortunately, as shown in Figure 23(c), this problem can easily be solved by architecting the latency of this instruction as two cycles. Any instruction that is dependent on the destination register of the xfer.s instruction should be scheduled one cycle later, and this is the responsibility of a programmer or a compiler.

Another NUCA design issue is with respect to how to implement the LRU policy efficiently. Figure 24 illustrates an instance for our decoupled WLP cache. In this example, the host processor issues to read *line 2*, and seven PEs have seven different cache lines mapped to the same set as shown in Figure 24(a). Upon detecting the requested *line 2* in PE24, PE24 transfers it to the Chameleon Virtualizer, and those PEs whose row numbers are smaller than PE24 will transfer their cache lines to the north (Figure 24(b)). This movement allows PEs to maintain more recently used cache lines closer to the Chameleon Virtualizer as shown in Figure 24(c).

The final design consideration of our NUCA C-mode is the placement of the cache control array. In a decoupled design, the cache control array is located in row 0, nearest to the Chameleon Virtualizer, reducing the tag lookup latency significantly. Our NUCA design adopts a decoupled design as its base so that the Chameleon Virtualizer can detect the location of the target data line early in its lookup stage. By moving the cache control bits across PEs in row 0 as shown in Figure 24(d), we can force the y-coordinate of a PE that keeps the target data line to be always bigger by one than the x-coordinate of a PE that keeps the target control bits. For example, if the x-coordinate of the PE that has the control bits of the requested cache line is three, the Chameleon Virtualizer can find corresponding data line in row 4.

Furthermore, we added another special move instruction to optimize the access latency of NUCA designs. This instruction is executed by PEs in row 1 but ignored by PEs in other rows. Upon decoding this instruction, the PEs in row 1 snoop the ORTree bus and store the values of the ORTree into a destination register. With



(a) Cache line location before reading line 2

(b) Cache line movement upon hitting line 2

(c) Cache line location after reading line 2

(d) Cache control array movement on hitting line 2

Figure 24: LRU management of a NUCA C-mode (decoupled WLP cache).

this instruction, the PEs in row 1 can obtain the hit line directly when the line is transferred to the Chameleon Virtualizer. Without this instruction, the Chameleon Virtualizer has to read back this cache line and write it to the PEs using IBus, which takes at least eight cycles in our baseline architecture.

5.2 P-Mode: Virtualizing Idle Cores as a Prefetcher

In addition to the C-mode, which supplies a virtualized last-level cache, we also investigate enabling mechanisms to reconfiguring idle PEs to work as a data prefetcher. The rationale behind this is from the following observation— the off-chip bandwidth of a heterogeneous multicore processor is typically very large for fulfilling the heavy input demand of the acceleration cores. This bandwidth, when running single thread applications, may be left unused. Reusing this bandwidth resource to perform data prefetching can potentially improve performance. Even in the scenarios when the prefetches issued are less accurate, they would unlikely affect the overall memory performance if the amount of off-chip bandwidth can satisfy both demand fetches and prefetches. We call this prefetching operation mode of the PE array *P-mode* (or prefetching mode).

In this work, we evaluate two data prefetchers for Chameleon P-mode: a Markov prefetcher [60] and a program counter (PC)-indexed delta correlation prefetcher [63, 92]. We chose these prefetchers because they use reasonably large prefetch tables to track miss addresses. These are non-trivial overheads to the hardware if implemented exclusively for prefetching purposes. Hence, even state-of-the-art processors do not adopt such implementations, rather, they implemented a simpler next-line prefetcher [48] or a stride prefetcher [118]. We will demonstrate that Chameleon can realize such area-consuming schemes by virtualizing the resources in P-mode.

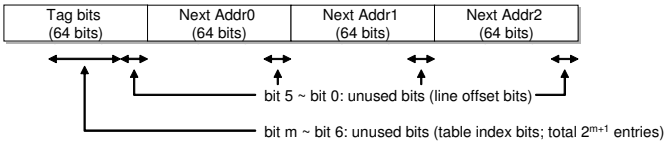
In the P-mode, the prefetch table is virtually laid out across PEs. Upon an L2 cache miss, the Chameleon Virtualizer checks its prefetch buffer first (Figure 16). If

the requested line is not found, then the Chameleon Virtualizer broadcasts microcode to look up the virtualized prefetch table. This microcode drives each PE to perform index hashing, to match tag bits, and to route a target prefetch table entry back to the Chameleon Virtualizer. Then, the Chameleon Virtualizer decodes the table entry and generates prefetch requests. To support P-mode, we added a small data prefetch buffer (a 32-entry buffer in this study) in the Chameleon Virtualizer as shown in Figure 16. A prefetched cache line is temporarily stored in this buffer, which is checked upon every L2 cache miss.

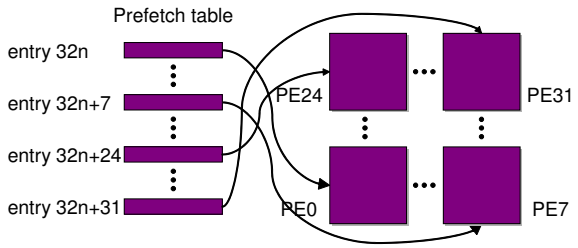
5.2.1 Virtualized Markov Prefetcher

Figure 25(a) shows the prefetch table design for a virtualized Markov prefetcher. Although the original Markov prefetcher paper [60] showed that a prefetch table with four next-miss addresses provides a reasonable balance between coverage and accuracy, implementing a virtualized Markov prefetcher with four next-miss addresses per entry is challenging because the number of entries per PE should be a power-of-two for simpler PE indexing and because an entry with four next-miss addresses requires at least 40B (larger than 32B but significantly smaller than 64B). In other words, an entry with four next-miss addresses requires 64B with 24B of unused bits, which results in area inefficiency. Thus, we evaluate a Markov prefetcher with three (instead of four) next-miss addresses. As shown in the figure, the size of the prefetch table entry is 32B, and one entry consists of the 8B current-miss address in the tag and three 8B next-miss addresses. Clearly, this design contains unused bits, i.e., table index bits and line offset bits (Figure 25(a)), which results in area inefficiency. However, we cannot compact the table entry because a 8B load or a 8B store instruction of a PE is aligned at an 8B boundary. In our proposed design, 4,096 entries can be stored in each PE's local scratch-pad memory space.

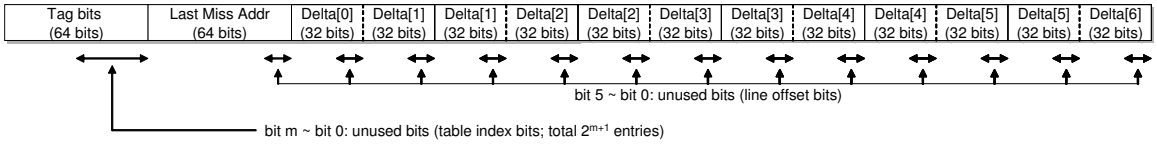
The P-mode Markov prefetcher is indexed by taking a group of bits (e.g., 17 bits



(a) A Table Entry of a Markov Prefetcher



(b) Table Layout (when only 32 PEs are used.)



(c) A Table Entry of a Delta Correlation Prefetcher

Figure 25: P-mode prefetcher.

on 32 PEs) from a miss address. These index bits consist of PE ID bits (e.g., 5 LSBs on 32 PEs) and local index bits (e.g., 12 MSBs). The PE ID bits are used to select only one PE that has the target prefetch table entry while the local index bits are used to generate the memory address of the selected PE's local scratch-pad memory. Mapping between the logical table entries and PEs is shown in Figure 25(b). In this example, each prefetch table entry is stored in one of the 32 PEs using the five LSBs of the table index as shown in the figure. One design issue is the trade-off between the size and latency of the P-mode prefetcher. If there is no need for a large prefetch table, it would be better off to enable only eight PEs in the first row to reduce the lookup latency. In this study, we vary the size of the prefetch table (one, two, four,

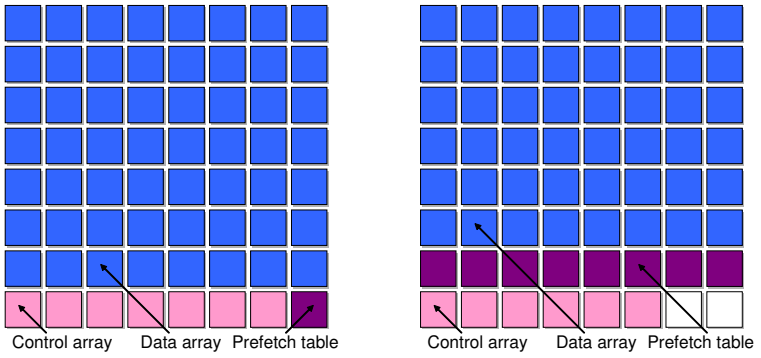
and eight rows) and perform a sensitivity study in our result section.

The overall procedure is as follows: Upon an L2 cache miss, the Chameleon Virtualizer broadcasts the current data miss address followed by microcode to perform table lookup. This microcode retrieves the hit-prefetch table entry along with its three next-miss addresses. Then, the Chameleon Virtualizer decodes this return message and generates three prefetch requests.

5.2.1.1 Virtualized Delta Correlation Prefetcher

In addition to a Markov prefetcher, we also evaluate a delta correlation prefetcher [63, 92] that keeps the seven latest address delta values. In this delta correlation prefetcher, we compare a pair of two consecutive delta values, (δ_i, δ_{i+1}) $(2 \leq i \leq 5)$, with the pair of two latest consecutive delta values, (δ_0, δ_1) where δ_n is the n^{th} latest delta value. Figure 25(c) shows a prefetch table entry of a P-mode delta correlation prefetcher. The size of each entry is 64B: 8B for tag bits, 8B for the last miss address, and six pairs of 4B delta values. As shown in the figure, instead of keeping seven distinct delta values, our implementation keeps six pairs of two consecutive delta values. In other words, we keep redundant delta values between neighboring pairs. For example, the first pair consists of δ_0 and δ_1 while the second pair consists of δ_1 and δ_2 (Figure 25(c)). Clearly, such data layout is inefficient in terms of area. However, we found that, with this layout, we can accelerate the correlation matching process by performing an 8B comparison operation instead of performing two 4B comparison operations or concatenating two delta values. Furthermore, this layout is not perfect in terms of the number of bits due to those unused bits, i.e., table index bits and line offset bits (Figure 25(c)). However, due to the same reason as the P-mode Markov prefetcher, the overall table size will be no larger even if we compact the table entry.

As mentioned earlier, the P-mode delta correlation prefetcher is indexed by taking several LSBs from the PC. As in the P-mode Markov prefetcher, these index bits are



(a) 7-way 7MB NUCA WBLP cache + 128KB delta correlation prefetcher

(b) 6-Way 6MB NUCA WBLP cache + 1MB delta correlation prefetcher

Figure 26: Hybrid design (cache + prefetcher).

used to locate a target PE and to locate a target table entry within the selected PE's local scratch-pad memory. Upon an L2 miss, the Chameleon Virtualizer broadcasts the instruction's PC followed by the microcode to perform the table lookup. This microcode retrieves a corresponding prefetch table entry from one of the PEs, and the Chameleon Virtualizer calculates the next miss address(es) based on the miss address and the delta values stored in this table entry.

5.3 HybridCP-Mode: Virtualizing Idle Cores for Caching and Prefetching

Instead of dedicating all idle cores as either a last-level cache or a prefetcher, in this section, we propose a hybrid design that virtualize idle cores as a last-level cache backed by a prefetcher. We call this operation mode the *HybridCP-mode*.

Figure 26(a) shows a design of the HybridCP-mode based on a 7MB NUCA WBLP cache and a 128KB prefetch table. As explained earlier, in a NUCA design, 7 PEs in row 0 are filled with cache control bits, while PEs in row 1 to row 7 are filled with cache data. The example shown in the figure places its prefetch table in PE7,

which is not utilized in the NUCA WBLP cache. In this example, upon an L2 cache miss, the Chameleon Virtualizer broadcasts cache lookup microcode to all PEs, and it handles returning messages. Once it detects a miss in a virtualized last-level cache, it looks up its prefetch buffer first and broadcasts prefetch table lookup microcode if the target line has not been prefetched. However, because the C-mode microcode and the P-mode microcode share IBus bandwidth, their operations cannot be overlapped. Figure 26(b) shows another example of the HybridCP-mode in which a 6-way 6MB cache is co-located with a 1MB delta correlation prefetcher table. In this example, the virtualized last-level cache is shrunk, but the prefetch table is enlarged. Note that certain partitioning such as a 4MB cache and a 3MB prefetch table may be infeasible due to PE indexing.

5.4 AdaptiveCP-Mode: Mode Adaptation in Chameleon

Although we have a wide spectrum of Chameleon design space, it is unlikely that any single design choice will prevail in performance for all applications due to the unpredictability of the characteristics in the algorithms and their workloads. A hybrid design will perform better when the host processor runs an application with good locality and a reasonable size of working set. However, when the host processor runs a streaming application (high L3 miss rate), due to its additional cache lookup latency, a hybrid design may perform worse than one with data prefetching capability only. Furthermore, some applications are purely computation-intensive, thus Chameleon will not help to improve the overall performance but consume more power. To obtain the best breed of all, we propose an adaptive mode that dynamically selects one of these Chameleon modes. As Chameleon itself is built on microcode-controlled PEs, an adaptive mode can be implemented at mild hardware cost: changing the microcode PC to be executed, adding a couple of performance counters, and adding additional control logic in the Chameleon Virtualizer.

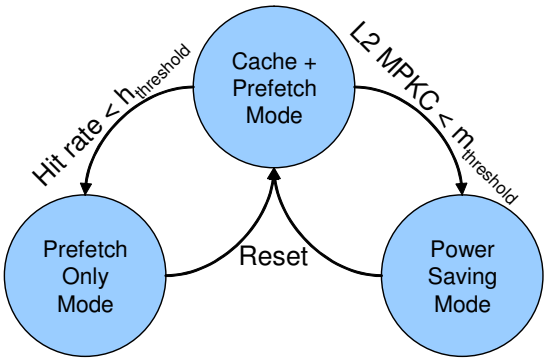


Figure 27: AdaptiveCP-mode (MPKC: misses per kilo cycles).

Figure 27 shows an example mode transition of our adaptive mechanism. Once an application is launched, Chameleon is operated in the HybridCP-mode. If the application does not show good locality or has a large working set resulting in a low hit rate, then Chameleon will disable its cache functionality completely and use only its data prefetching functionality. If the application does not have many L2 cache misses (i.e., measured by MPKC, misses per kilo cycles), then Chameleon disables both caching and prefetching to save power. We do not include cache-only mode as the P-mode prefetcher does not harm the overall performance as will be shown in Section 5.5 because it does not pollute the regular cache hierarchy. To implement an adaptive mechanism, two performance counters are added: an L3 cache hit counter and an L3 cache access counter (equivalent to the L2 cache miss counter). After launching a new process and warming up the C-mode cache, the Chameleon Virtualizer can monitor these two performance counters to make a decision of what mode is more appropriate for the running application.

5.5 Experimental Results

5.5.1 Simulation Environment

Two simulators were used in our analysis. The first one is a cycle-level simulator we developed for the baseline SIMD engine. In addition to an accurate model of PE

Table 2: Host processor configuration.

Clock frequency	3.0 GHz
Processor model	out-of-order
Machine width	3 (fetch) / 3 (issue) / 3 (retire)
The number of pipeline stages	1 (fetch) / 4 (decode) / 2 (rename) / 4 (wakeup) / 1 (schedule)
ROB size	128
Physical register file size	96 (INT) / 96 (FP)
Branch predictor	Hybrid branch predictor (16k global / local / meta tables), 2k BTB, 32-entry RAS
ITLB	dual-port, 4-way set-associative, 64-entry
DTLB	dual-port, 4-way set-associative, 64-entry
L1 instruction cache	dual-port, 2-way set-associative, 32KB cache with 64B line; 1 cycle hit latency; 1 cycle throughput
L1 data cache	dual-port, 2-way set-associative, 32KB cache with 64B line; 1 cycle hit latency; 1 cycle throughput
L2 cache	single-port, 8-way set-associative, 512KB (1MB, 2MB) cache with 64B line; 15 cycle hit latency; 3 cycle throughput; a stride prefetcher with a 256 entry prefetch table
Memory	Four 64-bit channels, 800MHz double data rate, 350 cycle latency

microarchitecture pipeline, it models latency and bandwidth of the interconnection network among PEs including the IBus, ORTree, MBus, and mesh network. Additionally, we integrated the Chameleon functionality into this simulator. The second simulator is SESC [105], a cycle-level architectural simulator. SESC is used to model the host processor, its conventional cache hierarchy, and the off-chip DRAM memory. SESC retrieves latency and throughput[4] information measured by the baseline SIMD simulator and uses them to simulate the entire heterogeneous architecture. Table 2 lists the configuration of the simulated host processor. Unless otherwise stated, the capacity of our baseline L2 cache is 512KB. We also show simulation results with 1MB and 2MB baseline models later. Throughout this section, the baseline performance is measured with this host processor model without any Chameleon capability unless stated otherwise.

To evaluate the effectiveness of the Chameleon architecture for improving sequential performance, we used the SPEC2006 benchmark suite. The entire SPEC2006 benchmark suite was used except 434.zeusmp, 465.tonto, and 470.lbm, which incurred

[4]In this article, throughput is defined as the number of cycles a cache port is occupied by a cache operation. For example, the throughput of a fully-pipelined cache is one, while the throughput of an non-pipelined cache is generally equal to its access latency.

Table 3: Latency and throughput of different C-mode designs.

Legend	Description	LRU State of the Hit Line	Read				Write				Replace
			Latency		Throughput		Latency		Throughput		Throughput
			Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss	
wlp	WLP-style 8-way 4MB	MRU	43	40	44	40	40	40	44	40	37
		non-MRU			46				49		
wblp	WBLP-style 8-way 4MB	MRU	37	36	37	36	36	36	37	36	18
		non-MRU			42				42		
wlp_nuca	Decoupled WLP-style 7-way 7MB	row1 (MRU)	39	21	29	21	20	20	43	20	45
		row2	41		44				57		
		row3	43		46						
		row4	45		48						
		row5	47		50						
		row6	49		52						
		row7 (LRU)	51		54						
wblp_nuca	Decoupled WBLP-style 7-way 7MB	row1 (MRU)	35	20	24	20	20	20	25	20	23
		row2	37		37				36		
		row3	39		39						
		row4	41		41						
		row5	43		43						
		row6	45		45						
		row7 (LRU)	47		47						
wlp_8banks	8-bank *wlp*	Latency and throughput of each bank is same as *wlp*									

issues such as cross-compiling failure and unsupported system calls in our simulators. For all simulations, we fast-forwarded the first 10 billion instructions and simulated next two billion instructions.

5.5.2 Evaluation of C-Mode

First of all, we measured the latency and throughput of each C-mode design. Unlike a conventional cache where its latency and throughput are solely determined by the characteristics of transistors and wires, the latency and throughput of a C-mode cache are determined by the number of instructions that control PEs and the order of these instructions. For example, for a cache read operation, the read latency can be reduced if instructions routing a read-hit line back to the Chameleon Virtualizer are scheduled earlier than instructions that updates LRU bits. On the other hand, the number of instructions to perform a single cache operation will determine the throughput of a C-mode cache (in a single-bank design) because they consume the IBus bandwidth for the same number of clock cycles. In this work, we wrote microcode using PE assembly code to implement different designs and scheduled them carefully to minimize the latency. The throughput is measured by counting the number of PE

instructions to perform a cache operation, and the latency is measured by monitoring the time when a hit/miss signal or a requested cache line is returned to the Chameleon Virtualizer. We assume the PE array operated in the A-mode and C-mode runs at the same frequency of the host processor, 3GHz.

Table 3 summarizes each cache design and their latency/throughput studied in this section. As shown in Table 3, the latency and throughput of NUCA models vary depending on which row an access hits[5]. Furthermore, even in non-NUCA designs, the throughput can vary depending on whether a hit line is located at an MRU position or not. When hitting an MRU line, we do not need to update the LRU bits, so the Chameleon Virtualizer does not need to broadcast instructions to update the LRU state.

Not surprisingly, the latency of a WBLP-style cache is lower than that of its WLP-style counterpart. In the case of a WLP-style cache, if the row number of a hit PE is greater than three, the latency of the NUCA design will be worse. In a WBLP-style cache, this threshold will be two. The sophisticated LRU management of NUCA designs is found to be the main reason for this effect.

The table also shows the read throughput of each design. As shown, there exists a trade-off in throughput between a NUCA design and its counterpart. In the WLP- and WBLP-style caches, not updating the LRU status upon hitting an MRU line helps reduce their throughput by two and five cycles, respectively. A similar trend is observed for the latency and throughput of a write and replacement operation as well.

Now we evaluate and quantify the performance potential for single-thread applications by using the C-mode on a heterogeneous multicore processor. Figure 28 shows the performance impact of different C-mode designs. To show the theoretical limit,

[5]In this study, we use an expression, a hit PE, to address a PE which has a requested cache line in its local scratch-pad memory space. Similarly, a hit row is defined as the number of a row to which the hit PE belongs.

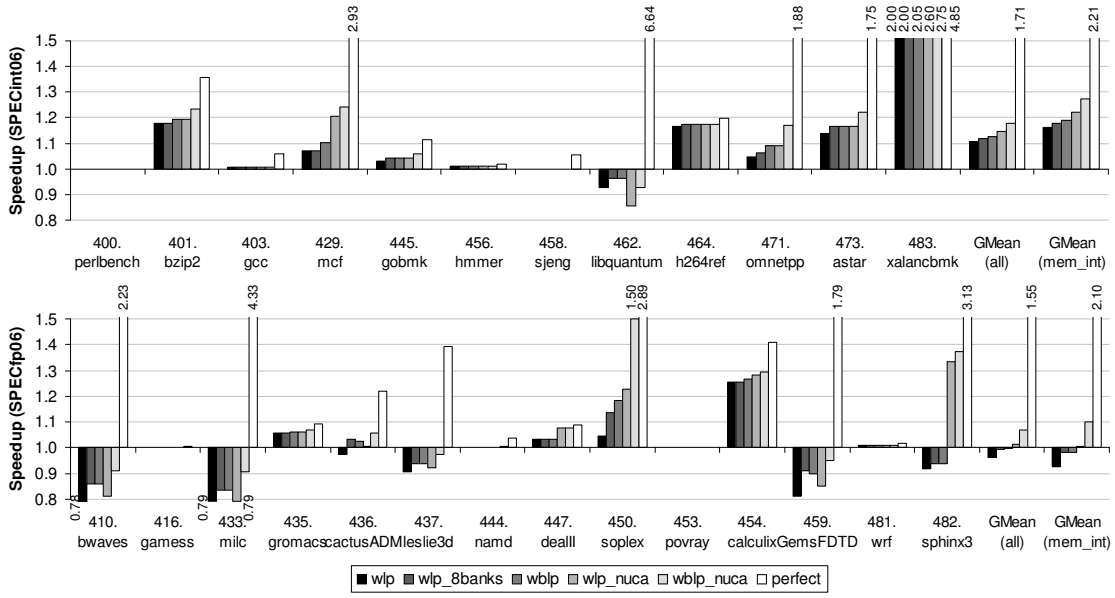


Figure 28: Relative performance of different C-mode designs.

we also simulated a *perfect* memory model in which the L2 cache is assumed perfect. This model also reveals those benchmark programs that are memory-intensive. In this article, we define memory-intensive applications as applications whose performance can be improved more than 10% with a perfect L2.

Not surprisingly, the C-mode improves the performance of memory-intensive applications, e.g., 401.bzip2, 429.mcf, 464.h264ref, 471.omnetpp, 473.astar, 483.xalancbmk, 450.soplex, 454.calculix, and 482.sphinx3. For example, the NUCA WBLP-style (*wblp_nuca*) C-mode improves the performance of 483.xalancbmk by 175%. Overall, it is found that the NUCA WBLP-style C-mode is the most effective design. On average (geometric mean), it improves the performance of SPECint06 applications and SPECfp06 by 18% and 7%, respectively. For the memory-intensive application category, the average performance improvements for them are 27% and 10%, respectively.

However, the performance of some memory-intensive applications was degraded including 462.libquantum, 410.bwaves, 433.milc, 437.leslie3d, and 459.GemsFDTD. We found that the hit rates of the C-mode cache were very low when the host processor

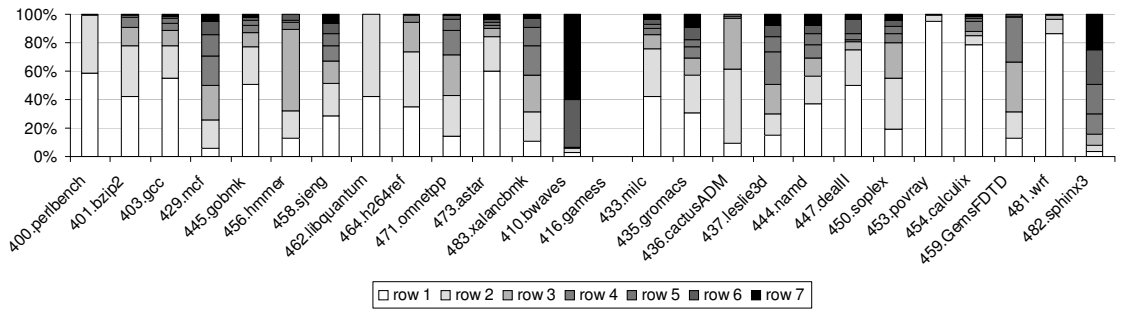


Figure 29: Distribution of hit rows.

runs these applications, so an additional cache level will only introduce extra latency in bringing data back.

Apparently, the NUCA models are effective despite their longer latency when a hit PE is located far from the Chameleon Virtualizer. Figure 29 shows the distribution of hit rows. Note that 416.gamess is extremely computation-intensive and generates a small number of cold misses, which results in a 100% L3 miss rate. Therefore, no bar is shown in Figure 29 for it. As shown in the figure, most of the cache hits are found in the PEs close to the Chameleon Virtualizer, which justifies the hardware/software effort for addressing the time-zone effect.

Another interesting result is that a multi-banked WLP-style cache (*wlp_8banks*) is not as effective as its counterpart: a single-bank WBLP-style cache (*wblp*). As shown in Figure 28, the performance improvement by a single-bank WBLP-style cache is always higher than or close to that of its multi-banked WLP-style counterpart. This implies that a C-mode cache is accessed infrequently so that designing a faster C-mode cache is more favorable than designing a slower but multi-banked C-mode cache.

We also performed simulations with a baseline with a larger L2 cache. On average (geometric mean), when a 1MB L2 cache is used, a NUCA WBLP-style cache improves the performance of SPECint06 and SPECfp06 by 14% and 3%, respectively (21% and 5% for memory-intensive applications). When a 2MB L2 cache is used, the

Table 4: Latency and throughput of different P-mode designs.

Legend	Description	Latency	Throughput
Markov1	1MB Markov prefetcher table on 8 PEs in row 0	21	22
Markov2	2MB Markov prefetcher table on 16 PEs in row 0 and 1	23	
Markov4	4MB Markov prefetcher table on 32 PEs in row 0 to 3	27	
Markov8	8MB Markov prefetcher table on 64 PEs in row 0 to 7	35	
delta1	1MB delta correlation prefetcher table on 8 PEs in row 0	37	29
delta2	2MB delta correlation prefetcher table on 16 PEs in row 0 and 1	39	
delta4	4MB delta correlation prefetcher table on 32 PEs in row 0 to 3	43	
delta8	8MB delta correlation prefetcher table on 64 PEs in row 0 to 7	51	

NUCA WBLP-style cache improves the performance of SPECint06 and SPECfp06 by 11% and 2%, respectively (17% and 3% for memory-intensive applications).

5.5.3 Evaluation of P-Mode

Table 4 describes each prefetcher design used in this section and shows its table lookup latency and throughput. As expected, we found a trade-off between the table lookup latency and the table size. For example, the lookup latency is 21 cycles for a 1MB Markov prefetcher table while it is 35 cycles for an 8MB table. This trade-off is represented in the overall performance graphs shown in Figure 30. For brevity, we show only the performance result of some prefetcher designs that reveal the trade-off well. For the P-mode Markov prefetcher, a large table is more useful as shown in the simulation results of 483.xalancbmk. This is intuitive because a Markov prefetcher is indexed by a miss address, so a larger table will be able to cover more miss addresses. However, we found that a 1MB P-mode delta correlation prefetcher is sufficiently large because it is indexed by a PC.

In most cases, a P-mode delta correlation prefetcher performs better than a P-mode Markov prefetcher. Seven exceptions are 445.gobmk, 471.omnetpp, 473.astar, 483.xalancbmk, 435.gromacs, 447.dealII, and 454.calculix. However, we also found that the performance improvements achieved by a P-mode Markov prefetcher on these applications are actually lower than those by a C-mode cache. In brief, the P-mode Markov prefetcher is less appealing compared to other C-mode caches or the P-mode delta correlation prefetcher. On average, the 1MB P-mode delta correlation prefetcher

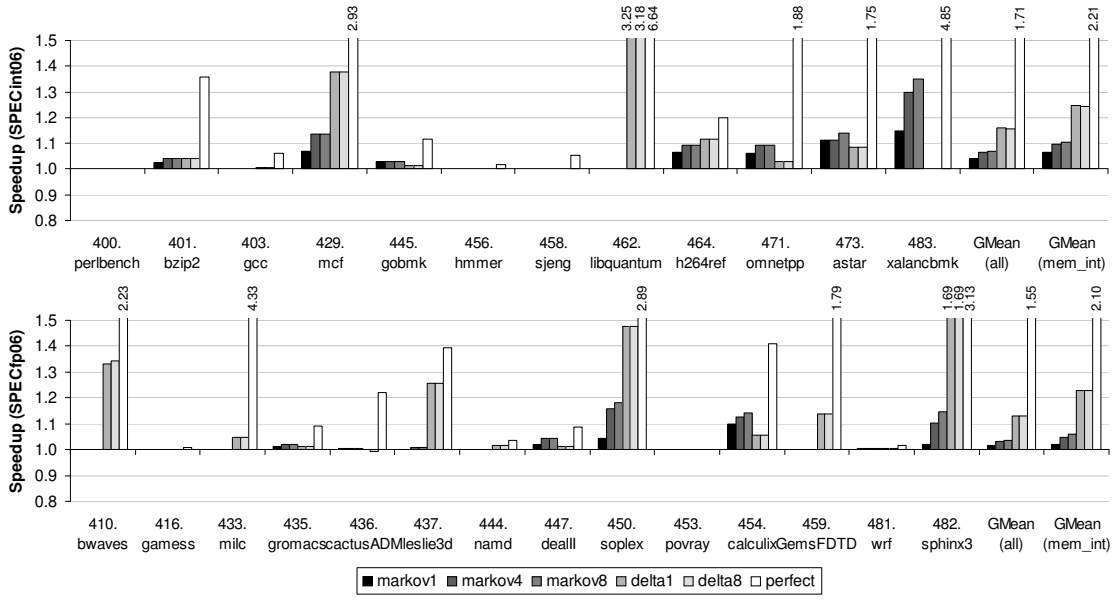


Figure 30: Relative performance of different P-mode designs.

improved the performance of SPECint06 and SPECfp06 applications by 16% and 13%, respectively. Their average improvements for memory-intensive applications are 25% and 23%.

5.5.4 Evaluation of HybridCP-Mode and AdaptiveCP-Mode

As shown previously, certain applications benefit more from a C-mode cache while some show more improvement when a P-mode prefetcher is used. For example, the NUCA WBLP-style C-mode cache improves the performance of 483.xalancbmk by 175%, but no improvement is obtained with a 1MB P-mode delta correlation prefetcher. In contrast, the 1MB P-mode delta correlation prefetcher improves the performance of 462.libquantum by 225%, but using the NUCA WBLP-style C-mode cache degrades it by 7%. More interestingly, Figure 31 shows that the HybridCP-mode and the AdaptiveCP-mode can provide reasonable performance improvement across applications with different characteristics. For easier comparisons, the figure also show their best performing C-mode (*wblp_nuca*) and P-mode (*delta1*).

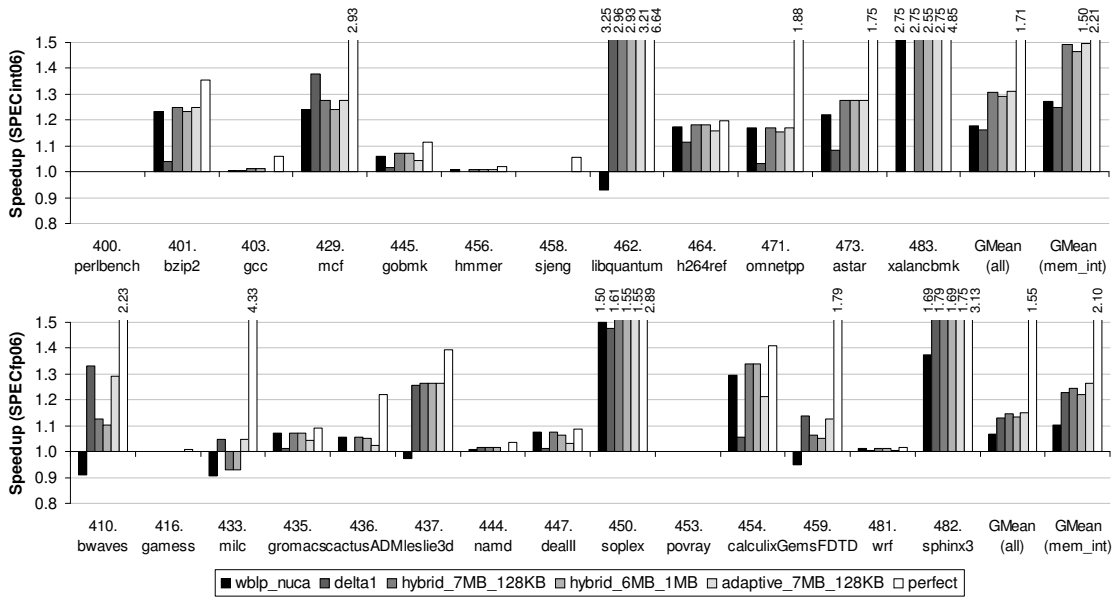


Figure 31: Relative performance of HybridCP-mode and AdaptiveCP-mode.

Two HybridCP-mode designs were evaluated: a hybrid design with a 7MB NUCA WBLP cache and a 128KB delta correlation prefetcher (*hybrid_7MB_128KB* of Figure 26(a)) and a hybrid design with a 6MB NUCA WBLP cache and a 1MB delta correlation prefetcher (*hybrid_6MB_1MB* of Figure 26(b)). In most cases, the performance difference between these two hybrid designs is small except two applications: 483.xalancbmk and 482.sphinx. As shown in Figure 28, their performance is improved a lot with a bigger cache, and that is why their performance is improved more with the hybrid design with a 7MB NUCA WBLP cache and a 128KB delta correlation prefetcher. On average, this hybrid design improves the performance of SPECint06 and SPECfp06 by 31% and 15%. The average performance improvements for memory-intensive ones are 49% and 24%. (For the remaining of the article, the HybridCP-mode refers to the hybrid design with a 7MB cache and 128KB prefetcher unless explicitly stated otherwise.)

On the other hand, we found an AdaptiveCP-mode can perform as well as the HybridCP-mode. This adaptive one is based on the previous HybridCP-mode with

a 7MB NUCA WBLP cache and a 128KB delta correlation prefetcher. However, we disable its cache functionality to behave as a 128KB delta correlation prefetcher based on the algorithm shown in Figure 27. In this evaluation, we modeled the AdaptiveCP-mode to make decisions after warming up the cache during the first 30 million cycles (0.1 ms) and then monitoring the number of cache accesses and hits for the next 30 million cycles. In this set of simulations, we perform this sampling every 600 million cycles (2 ms) to find a better Chameleon mode just in case a program phase changes. The model in Figure 31 uses 30% for the threshold of the cache hit rate and 0.5 for the number of L2 misses (= L3 accesses) per kilo cycles (MPKC), which are found to provide the highest average improvement (although not significantly) according to our sensitivity study. As shown in the figure, the AdaptiveCP-mode performs at least as well as the HybridCP-mode. In particular, it prevails when the host processor runs applications favoring a prefetcher, e.g., 462.libquantum and 433.milc. On average, the AdaptiveCP-mode improves the performance of SPECint06 and SPECfp06 by 31% and 15%, repectively. For memory-intensive applications, it improves by 50% and 26% on average.

Furthermore, this adaptive design does not degrade the performance of any application. (Note that the only application whose performance is degraded by the HybridCP-mode is 433.milc with a 7% degradation.)

We also performed a sensitivity study with different L2 sizes and summarize their average performance improvements as follows. Using a 1MB L2 cache baseline, the AdaptiveCP-mode improved the performance of SPECint06 and SPECfp06 by 27% and 12%, respectively (43% and 23% for memory-intensive applications). When increasing the capacity to 2MB, it improved the performance of them by 25% and 11%, respectively (40% and 21% for memory-intensive applications).

5.5.5 Hardware and Power Overhead

The hardware overhead to support Chameleon is insignificant. First of all, the Chameleon Virtualizer requires a prefetch buffer, an MSHR, a Chameleon microcode memory, and corresponding control logic changes. In our simulations, we modeled a 32-entry prefetch buffer and an 8-entry MSHR. In case of the HybridCP-mode, for example, less than 128 PE instructions are required. Thus, a 128-entry Chameleon microcode memory is sufficient to implement both the cache and the prefetcher. Conservatively assuming that supplementary control logic requires the same amount of space of these memory components, the area overhead compared to a baseline SIMD engine is estimated to be 0.01%. In this estimation, we used Intel's data [46] and Penryn die to estimate the sizes of the Chameleon Virtualizer and the baseline SIMD engine. Second, to support Chameleon, we added two new special "move" instructions explained in Section 5.1 and Section 5.1.4. Third, to provide a NUCA model, we added two sets of mux and demux as shown in Figure 21(b) as well as additional pipeline registers to solve the time-zone effect of the northbound transfer instruction. Fourth, to widen the datapath in a WBLP-style cache, we directly connected IBus and ORTree to the Chameleon Virtualizer without using conventional fan-out and fan-in trees. Note that this new wiring does not require any wiring change in each PE. Lastly, to support the AdaptiveCP-mode, we need to add two performance counters that count the number of accesses and the number of hits in the Chameleon cache.

We also evaluated the extra dynamic power dissipation for the AdaptiveCP-mode using Wattch [14] model. We additionally modeled the global interconnect (IBus, ORTree, Mesh) power consumption using the Berkeley Predictive Technology Model [17]. We conservatively modeled the power by assuming the worst-case power consumption in the cache and prefetch operations. For example, if a cache read hits in row 0 of the NUCA model, no data and cache control array migration is required. However,

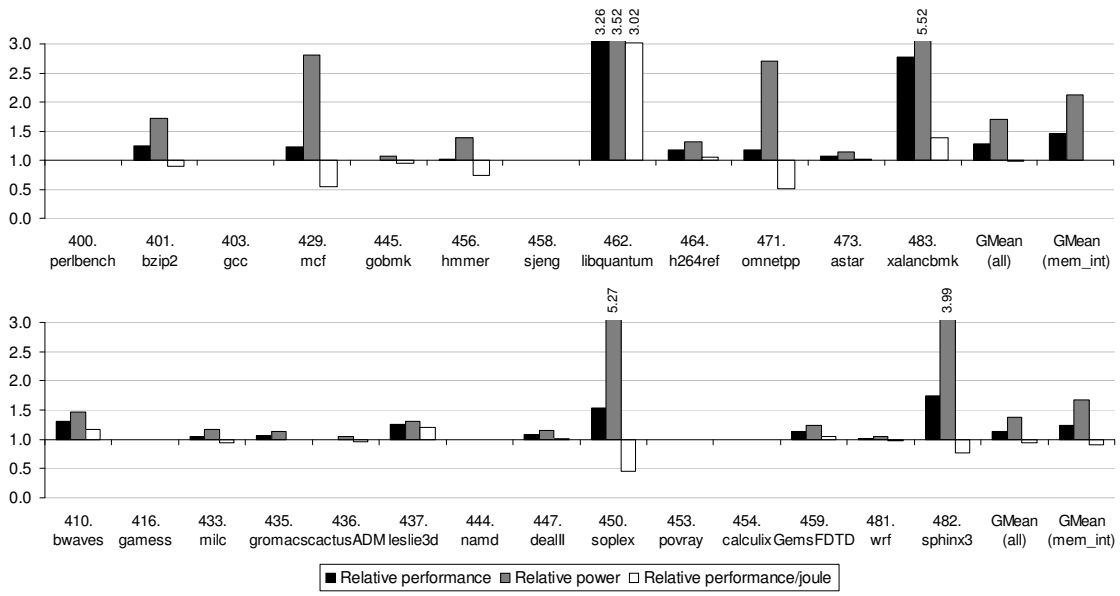


Figure 32: Performance, power, and performance per joule of the AdaptiveCP-mode.

for convenience, we conservatively modeled that all 56 PEs are active regardless of the LRU state of a hit line.

Figure 32 shows the relative power consumption and *performance per joule* of the Adaptive-CP mode. Note that the performance improvement is also shown for easier reference. Although Chameleon consumes a considerable amount of energy by running microcode on 64 PEs, because Chameleon is accessed very infrequently (on an L2 miss), Chameleon will consume only 69% (SPECint06) and 37% (SPECfp06) more power (Figure 32) than the baseline host processor with a conventional L1 and L2 caches. Note that the baseline SIMD engine is already designed to accommodate the power consumption of 64 active PEs. Thus, the power consumption of the AdaptiveCP-mode is still below the total chip power budget. This indicates that Chameleon is more power-efficient than other thread-level speculation techniques for improving sequential performance [2, 75]. Although power overhead analysis was not reported in these prior works, their power overhead is likely to exceed the power overhead of Chameleon due to their full utilization of all high performance cores while

Chameleon is only accessed upon an L2 cache miss.

Figure 32 also shows the energy efficiency represented in *performance per joule*, which represents achievable speedup under the same energy budget or energy efficiency. Overall, the AdaptiveCP-mode degrades the *performance per joule* of SPECint06 and SPECfp06 by 2% and 5% (0.98x and 0.95x in the geomeans). Interestingly, there are some applications whose *performance per joule* is improved a lot, such as 462.libquantum, 483.xalancbmk, and 437.leslie3d. In other words, as their performance is improved a lot, their energy efficiency can be improved in spite of Chameleon's power overhead. Another interesting observation is that if the application does not get any benefits using Chameleon, their energy efficiency is not affected as well for Chameleon is rarely accessed. However, energy efficiency of some applications, such as 429.mcf, 456.hmmer, 471.omnetpp, 450.soplex, and 482.sphinx3, is degraded as shown in the figure. We found that they prefer the HybridCP-mode, so they consume much energy upon an L2 cache miss. If one is particularly interested in energy efficiency rather than the performance itself, she or he can tune the threshold values of the adaptive Chameleon so that Chameleon is not turned on when the host processor runs other applications. However, optimizing for energy efficiency is out of scope of this study, and it remains as our future work.

5.6 Summary

In this study, we propose Chameleon, a flexible heterogeneous multicore processor that virtualizes idle acceleration cores for improving the memory performance of sequential code. To address the under-utilization issue when these cores are not used for DLP processing, Chameleon can virtualize these idle cores collectively into a last-level cache (C-mode) or a table-based data prefetcher (P-mode). for single-thread applications running on the host processor. We studied the trade-off between performance and architectural complexity of several caching designs. For data prefetching,

we demonstrated the mechanisms to reconfiguring these acceleration cores into a Markov prefetcher and a delta correlation prefetcher. Moreover, we introduce a hybrid mode to enable caching and data prefetching simultaneously using the collective acceleration cores. To achieve the highest efficiency for performance versus energy, we devise an adaptive mode to migrate the functionality of Chameleon between the hybrid mode and prefetch-only mode by monitoring the cache behavior.

We used the POD architecture for our case study. Using the SPEC2006 benchmark suite, we found that, on average, the Chameleon C-mode can improve the performance of SPECint06 and SPECfp06 by 18% and 7% while the Chameleon P-mode can improve them by 16% and 13%. Furthermore, our hybrid mode shows a 31% and 15% improvement, respectively. In the adaptive mode, 31% and 15% are observed for SPECint06 and SPECfp06. Finally, when accounting for memory-intensive applications only from the suite, the average speedups of the adaptive mode are increased to 50% and 26% for SPECint06 and SPECfp06, respectively.

CHAPTER VI

COMPASS: COMPUTE SHADER ASSISTED PREFETCHING

6.1 Introduction

In the previous chapter, we have demonstrated that idle accelerator cores can be used to improve the memory performance of the host processor. However, techniques proposed in the previous chapter were tightly coupled with our baseline architecture proposed in Chapter IV. In this chapter, we design and evaluate such an idea in a more generic heterogeneous computing platform, an integrated CPU and GPU platform. To meet the modern needs of game developers, a traditional fixed-function graphics accelerator has evolved into a programmable graphics processing unit (GPU), which allows game developers to write their own shaders for specific special effects. For its vast computational capability, a modern GPU is also designed to run non-graphics, compute-intensive applications, referred to as general-purpose GPU (GPGPU) [77]. Recently, Intel and AMD announced their integrated solutions to encompass the GPU, the memory controller, and the CPU onto a single die for netbook, laptop, and desktop products [85, 112]. Although the integrated chip is not likely to be as powerful as a standalone CPU or GPU due to several reasons such as power budget, it lowers the overall system cost and reduces the form factor with reasonable performance for its particularly aimed applications and market. Furthermore, the performance can be compensated to some extent due to the substantially reduced latency between the host CPU and the integrated GPU.

Unfortunately, while the host CPU executes the sequential part of a parallelized application or an unparallelized legacy application, the integrated GPU will sit idle

contributing nothing to the single-thread performance. Unlike symmetric multicore processors in which many sequential processes can concurrently run on multiple cores, an idle GPU cannot run a conventional CPU process due mainly to the heterogeneity between the ISAs. Moreover, an idle GPU cannot take advantage of other types of techniques, such as speculative multi-threading or helper threads [113, 45, 30, 25, 78, 6, 73, 89], to boost single-thread performance unless the GPU is completely re-designed to support it, which could unnecessarily complicate the entire design and lead to performance degradation when running conventional graphics applications.

One way to improve the performance of a CPU while an on-chip GPU is idle is to exploit the remaining power budget. Because an idle GPU only consumes a small amount of idle power compared to an active GPU, the CPU can then be given the unused power by increasing its supply voltage and clock frequency, similar to the Turbo mode employed in Intel's Core i7 (Nehalem) processor [59]. Nonetheless, this method will not improve the performance of memory-intensive, single-thread applications, which are typically unscalable and insensitive to clock frequency.

Instead of letting the GPU sit idle, we envision that the OS can utilize the idle GPU to run compute shaders to enhance the memory performance for single-thread applications. In this study, we propose COMPASS, a **comp**ute shader-**ass**isted prefetching scheme, to achieve our goal. With very lightweight architectural support, we demonstrate that COMPASS can enhance the single-thread performance of an integrated CPU by emulating the function of a hardware prefetcher using the programmable shader.

6.2 Baseline GPU Architecture

Figure 33 illustrates the baseline GPU architecture used in this study. Because the details of modern GPU architectures are not completely open to the public, we employed a baseline architecture that resembles an abstract GPU from several publicly

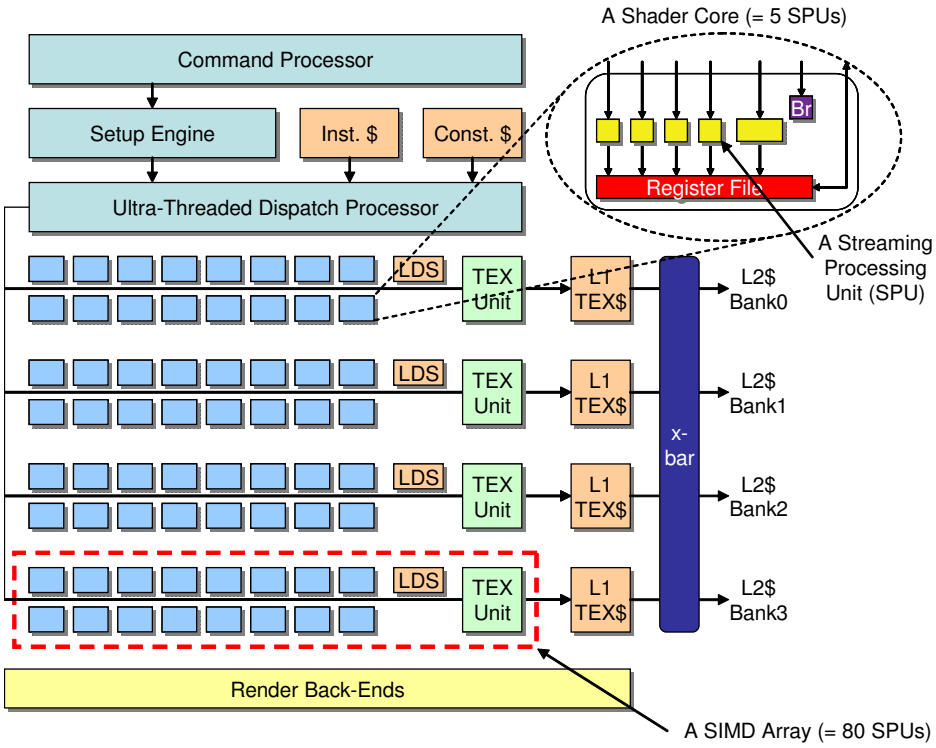


Figure 33: Baseline GPU architecture.

available documents [1, 55, 80, 81, 107]. As shown in the figure, at the front-end of a GPU pipeline, a programmable command processor interprets command stream from a graphics driver. The command processor executes a RISC-based microcode with its computation logic and memory. Then, a setup engine prepares data for a different shader (*e.g.*, a vertex, a fragment, a geometry, and a compute shader) and submits threads of each shader code to an ultra-threaded dispatch processor (or a thread scheduler). The ultra-threaded dispatch processor maintains separate command queues for different shader codes. It also has two arbiters for 16 cores of each SIMD array as well as an arbiter for each vertex/texture fetch unit. Furthermore, the ultra-threaded dispatch processor has a *shader instruction cache* and a *shader constant cache* to supply instructions and constant values.

The next pipeline stage of the baseline GPU executes a given shader code. As shown, the GPU consists of several SIMD arrays (four SIMD arrays in the figure),

each of which consists of 16 shader cores forming a 16-way SIMD array. Each shader core is a five-way VLIW machine, each execution unit of which is referred to as a streaming processing unit (SPU). An additional branch execution unit of each shader core handles flow control and conditional operations.

In the right-most column of the SIMD array, a specialized vertex/texture unit (labeled as a TEX Unit in the figure) is connected to a vertex and a texture cache, each of which supplies requested memory values to the SIMD array. (In the figure, only the L1 texture cache is shown for brevity.) Furthermore, 16KB local data share (LDS) is placed between the 16 shader cores and the vertex/texture unit. LDS enables efficient data sharing between threads mapped to the same SIMD array. In addition to LDS, another 16KB global data share (not shown in the figure) is present to allow data shared among different SIMD arrays, but we will not use it in this study. Additionally, the baseline GPU has other hardware units such as a render back-end unit for color blending, alpha blending, depth testing, and stencil testing, but we do not elaborate them here as they are not essential to the main idea of this study.

With these hardware resources, the baseline GPU is able to tolerate long cache miss latency (often in hundreds of cycles) by executing many threads alternately. Upon a cache miss of a thread, the ultra-threaded dispatch processor suspends the execution of the thread and schedules another thread to sustain the overall throughput. To achieve this, the ultra-threaded dispatch processor forms a group of 64 threads and uses this group as a *thread scheduling unit*. It essentially dispatches a group of 64 threads to the SIMD array simultaneously and later dispatches another group of 64 threads upon a cache miss of the previous group. This group is referred to as a *wavefront* (or a *warp* in NVIDIA terminology [36]). A wavefront (64 threads) executes one VLIW bundle on a 16-way SIMD array over four cycles as shown in Figure 34. In other words, one VLIW bundle of the first 16 threads is dispatched to the SIMD array at cycle $4n$, the same bundle of the next 16 threads is dispatched to the SIMD

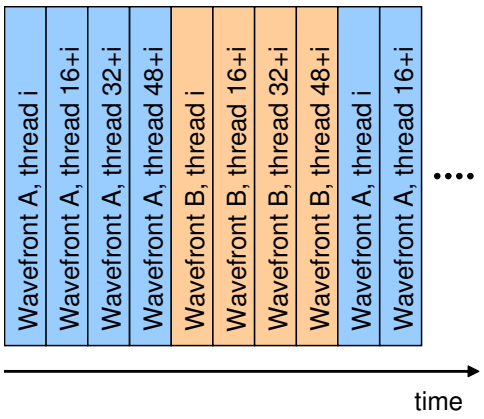


Figure 34: Thread scheduling at SIMD core i of a 16-way SIMD engine.

array at cycle $4n + 1$, and so on. As the ultra-threaded dispatch processor has two arbiters per SIMD array, two wavefronts compete to dispatch their instructions to the SIMD array. In this study, we assume that the same wavefront can be dispatched only after another wavefront is dispatched as shown in Figure 34.

To support such a large number of threads on four 16-way SIMD arrays, the GPU requires large register files. Following a speech from AMD [107], we assume that the capacity of the register file of each SIMD array is split into 256 sets, each consisting of 64 128-bit registers. It amounts to a total of 256KB ($256 \times 64 \times 16$) register space evenly split across 16 shader cores of each SIMD array. The total capacity of register files can vary according to target market or over different GPU generations. For different market segments, the GPU industry used to design its products with a different number of shader cores. For example, ATI Radeon HD 4890 (for the most enthusiastic gamers' market) consists of ten 16-way SIMD arrays while ATI Radeon HD 4600 (for the mainstream market) contains only four 16-way SIMD arrays. In this study, we employ a baseline GPU of four 16-way SIMD arrays for the integrated chip. Considering the number of SIMD arrays will continue to soar in the future GPU, our results based on this assumption of four SIMD arrays can be considered conservative.

Such a large pool of registers is shared by threads executed on the same SIMD

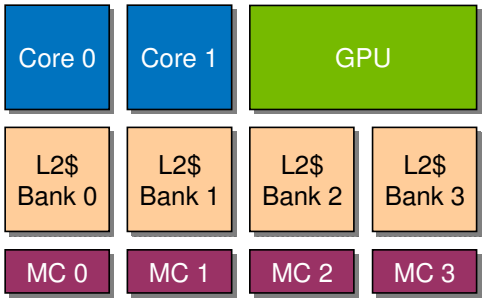


Figure 35: Integrated platform.

array. It also implies that more registers each thread uses, fewer threads can be simultaneously active. Such register partitioning is managed by GPU itself using a relative indexing scheme [1]. Because the details on register partitioning is not disclosed, we assume a simple indexing scheme— a global register index is the sum of a base register of a thread and a relative register index within the thread. For example, if we have a pool of 32 registers and if each thread uses four registers, only eight threads can be active simultaneously. In this example, the global index of a physical register is calculated by adding the register identifier to the base register index of the thread, which can be calculated by simply shifting the thread ID to the left by two.

Lastly, in this study, we assume that the integrated CPU cores and the GPU share a four-bank L2 cache as shown in Figure 35. Such architecture proposals have been considered in an early, scratched integrated solution, e.g., Intel's Timna processor, and the upcoming Intel's Sandy Bridge microarchitecture (shared L3). Such integrated chips, to our firm belief, will emerge and gain more interests for the following reasons: (1) saving the overall cost, (2) providing efficient and coherent communication between the host and the accelerator, and (3) giving more flexibility in cache space consumption, similar to the rationale of the Advanced Smart Cache in Intel's multicore processors. Note that, although our baseline architecture assumes a shared L2, COMPASS does not necessarily need a shared L2. In the case of a private L2

cache, our proposed mechanism can still be used if a very small design hook is implemented to forward an L1 miss request of the GPU to the L2 of the CPU when COMPASS is enabled. Using a private L2 (as employed by the recent IBM Power7) should not affect the overall results at all since COMPASS shaders do not generate any memory traffic except prefetch requests (will be detailed later). In addition, we also assume that memory controllers are integrated on-die as they are already in a current discrete GPU product.

6.3 COMPASS

Although data prefetching techniques have been widely researched, state-of-the-art commercial processors only either employ simple schemes such as a next-line prefetcher [48] or a stride prefetcher [118], or push it to the software side by providing prefetch instructions and let the compilers or programmers insert them at appropriate locations inside the code. Most of the advanced complex hardware schemes remain in literature due to their prohibitively expensive hardware cost [54, 60, 92, 114]. Furthermore, different applications may favor different types of prefetchers. Thus, a one-size-fits-all hardware-based prefetcher may not be in the best of interests of all applications. More importantly, the GPU consumes enormous memory bandwidth when active, which competes the same bandwidth shared by a dedicated hardware prefetcher for the CPU. Such scenarios, in effect, will deteriorate the performance of both the CPU and the GPU, diminishing the purpose of GPU acceleration.

To ameliorate the shortcomings of prior art, in this study, we propose COMPASS, a compute shader-assisted prefetching scheme, which uses the idle GPU to achieve the functionality of hardware prefetchers for improving single-thread performance on an integrated single-chip system. The rationale behind our design is as follows: (1) An on-chip GPU has large register files and rich computational logic, so it can emulate the behavior of hardware prefetchers, (2) While a GPU is idle, much of memory

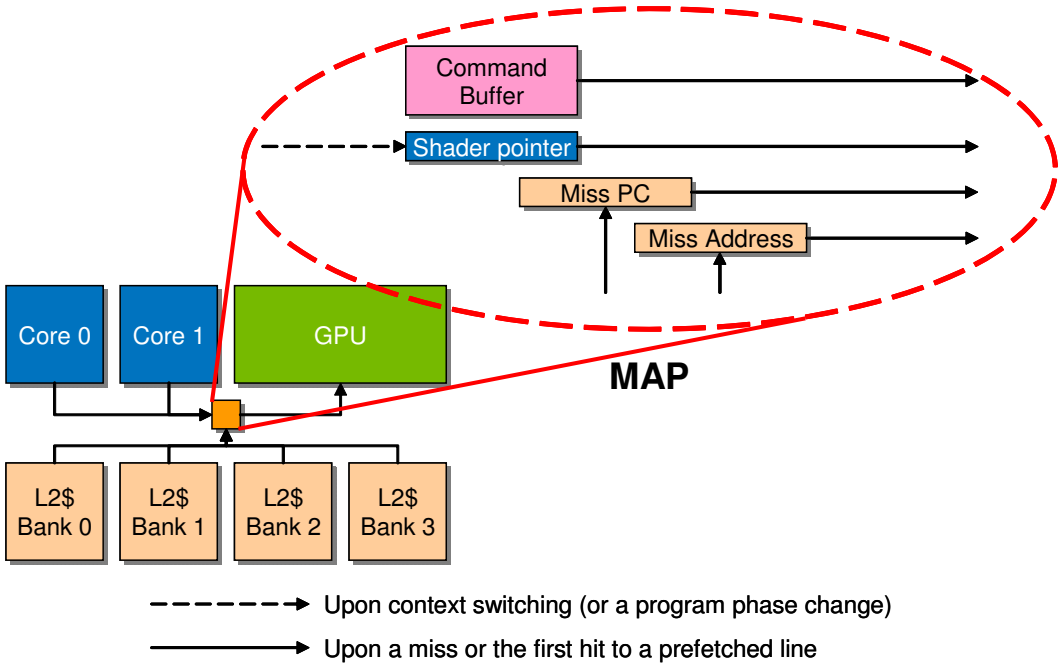


Figure 36: Miss address provider.

bandwidth originally designed to meet the requirement of a GPU, will be left unused and can be re-harnessed to assist the CPU cores, in particular, for data prefetching, and (3) The tight coupling of the CPU and the GPU on a single die facilitates efficient, prompt communication leading to synergistic outcome.

Note that one of COMPASS design goals is to reuse existing hardware as much as possible to minimize the overall hardware cost. Also, the overheads incurred by COMPASS should not affect and compromise the massive parallel computation capability in a GPU originally designed for 3D rendering and high-performance computing. Toward these objectives, we describe how to emulate different types of data prefetchers in subsequent sections.

6.3.1 Miss Address Provider

Because COMPASS is based on a compute shader, which is programmable, it provides much more flexibility than conventional hardware prefetchers. As such, instead of implementing COMPASS with a fully automatic hardware mechanism, we opted for an

OS or application vendors to offer and enable COMPASS prefetch capabilities. In this respect, we propose to add the *Miss Address Provider* (MAP), a hardware/software interface bridging the L2 cache, the GPU, and the OS. MAP is located between the L2 cache and the GPU as shown in Figure 36.

The operation of MAP is described step-by-step in the following. (1) Once the OS has no pending job to run on the GPU, the OS provides MAP a pointer to a compute shader for prefetching, referred to as a COMPASS shader. (2) Upon an L2 miss or the first hit to a prefetched line,[1] the program counter (PC) generating this memory request and its physical address are forwarded to MAP for prefetching. (3) Upon receiving these two values, MAP sends a GPU command to trigger the execution of a COMPASS shader. These two values, a PC and a physical address, are stored in the constant cache and read by the COMPASS shader. Furthermore, the command also indicates which value, the PC or the physical address, should be used to index a thread. (More details will be explained in Section 6.3.2.) (4) The role of the COMPASS shader is to read miss history from GPU's register files to predict the subsequent miss addresses, to update history information, and to execute prefetch instructions that bring data back to the L2. (5) Prior to a new job to be scheduled onto the GPU by the OS, MAP will be disabled. Note that the OS intervenes in COMPASS only when it needs to enable or disable a COMPASS shader upon context switching. The OS can change a COMPASS shader more often, *e.g.*, upon a program phase change, but such fine-grained execution is outside the scope of this study.

6.3.2 Threads and Register Files

A COMPASS shader triggered by MAP emulates a hardware prefetcher in the following manner. Basically, we emulate each entry of a prefetch table with one GPU thread (or with a set of GPU threads when using a multi-threaded COMPASS shader.) For

[1]Here, as in many previous prefetch studies, we assume that an L2 cache has a tag bit per cache line for prefetch. It is set when a line is prefetched in but not consumed.

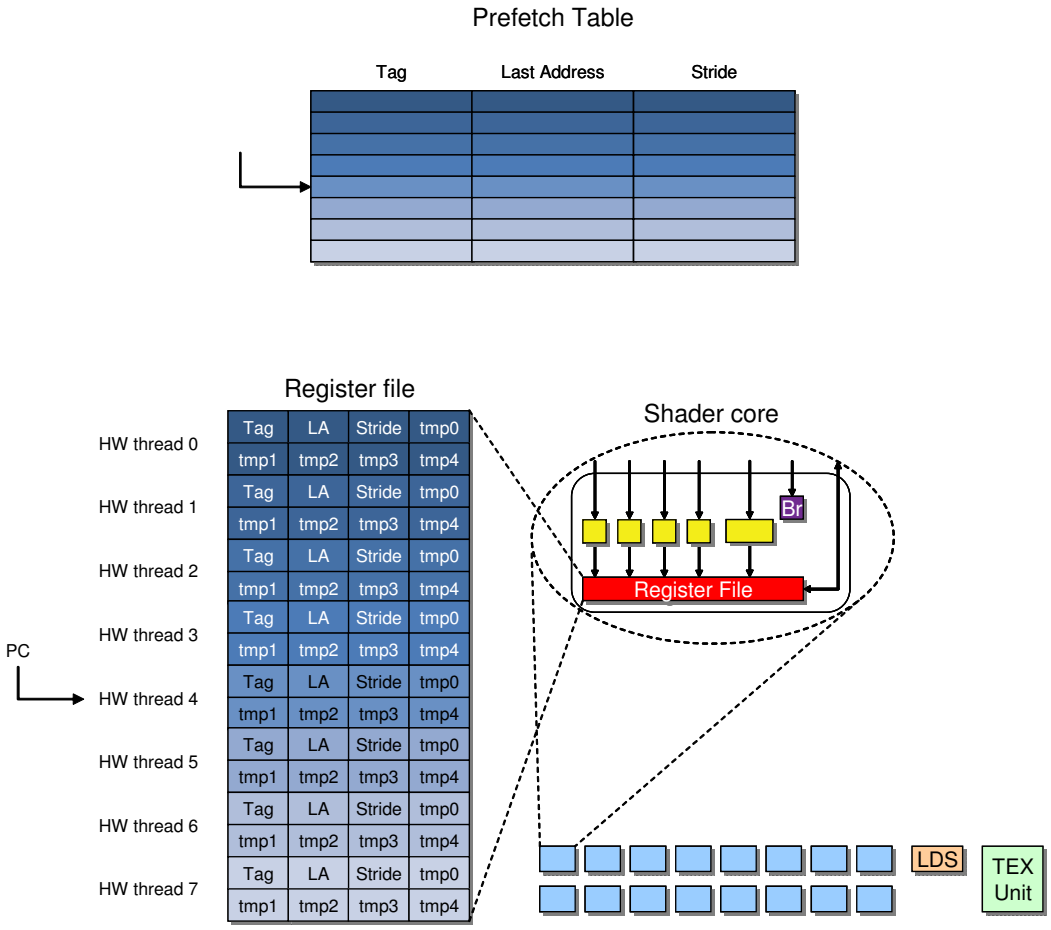


Figure 37: Hardware prefetcher vs. COMPASS.

example, as shown in Figure 37, GPU thread4 emulates the behavior of a prefetch operation that accesses the 5th entry of the prefetch table. In other words, the registers of a GPU thread (or a set of GPU threads) are used to record miss history information across the application's lifetime to emulate the same bookkeeping in a hardware prefetch table.

To allow such mapping, the setup engine assigns a hardware thread ID using the same index function as the index function of a hardware prefetcher. Such thread mapping circuit is not currently implemented in the setup engine of the GPU architecture. Clearly, this is an additional hardware overhead but is rather insignificant for the indexing requires only simple masking.

Because a typical hardware prefetcher table has a power-of-two entries to facilitate PC indexing, we also force the number of concurrently active hardware threads to be in power-of-two. Also, we wrote our COMPASS shader to always use power-of-two registers. With this simple trick, we do not need to modify the register partitioning hardware of the baseline GPU, but the requirement of using power-of-two registers is clearly a storage constraint.

Those registers allocated to a thread (or a set of threads in a multi-threaded COMPASS shader) are used for two different purposes. One group of registers stores miss history information of each table entry, and the others are used as temporary registers for calculating the next prefetch addresses and for updating miss history. For example, to store miss history of a stride prefetcher, each thread uses three registers to keep a tag value, a last address value, and a stride value, corresponding to an entry of a conventional stride prefetcher (Figure 37). To calculate future prefetch addresses and update the emulated table, the thread needs several temporary registers.

6.3.3 An Example of a COMPASS Shader

Figure 38 shows an example shader code for stride prefetching. This shader code uses three general-purpose registers to store a tag, a last miss address, and a stride value while using one more register as a temporary register. Two other values, a current PC and a current miss address, are provided by MAP in the form of constant values stored in the constant cache.

```
 1: if tag == currentPC then
 2:     tmp ← lastAddr + stride
 3:     if currentAddr == tmp then
 4:         tmp ← currentAddr + stride
 5:         prefetch   tmp
 6:     end if
 7: end if
 8: tag ← currentPC
 9: stride ← currentAddr − lastAddr
10: lastAddr ← currentAddr
```

Figure 38: Stride prefetching.

As shown in the code, one run of a COMPASS shader emulates one lookup of a

conventional hardware prefetcher. Such design necessitates the registers keeping the miss history information to remain unmodified until the thread that emulates the same prefetch table entry is dispatched. Such integrity of register files is enforced by the setup engine, which uses the same index function of a hardware prefetcher to allocate a hardware thread ID to each COMPASS lookup as we explained previously.

6.3.4 A Prefetch Instruction

To perform a prefetching operation, we can use a conventional load instruction of a GPU. However, the load instruction requires a destination register, which consumes one more register per prefetch request. As explained previously, more registers each thread uses, a smaller number of threads can be executed simultaneously. In other words, using a load instruction reduces the capacity of a prefetch table being emulated by a COMPASS shader. Hence, to reduce the number of required registers in each thread, we propose to add a prefetch instruction in the GPU. (Note that a GPU is unlikely to support a prefetch instruction given cache misses can always be hidden by the execution of a plethora of independent threads.) Implementing a prefetch instruction does not incur much hardware overhead because the prefetch instruction can be supported by disabling the write-back path of a load instruction.

6.3.5 Usage Model

We design a COMPASS shader to be selected by an OS. Through profiling, the OS can select the best matched COMPASS shader and enables it by setting a pointer to the selected COMPASS shader. On the other hand, the OS can also provide an API call to allow an individual application to select an appropriate COMPASS shader from a COMPASS library provided by the OS. Furthermore, an application vendor can provide an application-specific COMPASS shader through another API call. This API call provides the OS a pointer to their custom COMPASS shader. When the OS schedules the target application, the OS can read this pointer value and store the

pointer in the shader pointer register of the MAP (Figure 36). This execution model implies that even if the application has nothing to do with graphics rendering, the application developers can still use the GPU to enhance its performance by writing a prefetcher shader code and having it loaded by the runtime system.

6.4 Different COMPASS Shader Designs

In this section, we describe different COMPASS shader designs and analyze their design trade-offs. First of all, we evaluate three table-based prefetchers: a PC-indexed stride prefetcher [19], a Markov prefetcher [60], and a PC-indexed delta correlation prefetcher [63, 92]. On the other hand, we also evaluate a region prefetching technique [74]. In addition to these generic prefetchers, to demonstrate the unique feature of COMPASS, we also design and evaluate a custom-designed prefetcher for 429.mcf, which is known to be memory-bound. As mentioned earlier, an application vendor can write its own custom COMPASS prefetcher and pass it to the OS via an API call.

One issue to be addressed is the long latency of instructions in a GPU shader code. Notice that the design principle of GPUs is to optimize for high throughput. They exploit a large amount of thread-level parallelism to hide the instruction latency. As illustrated in Figure 34, if read-after-write dependency exists between two successive instructions of a COMPASS shader, the dependent instruction will be dispatched eight GPU cycles after its producer instruction is dispatched. This is certainly not a performance bottleneck for throughput-oriented graphics rendering algorithms, it is, however, clearly a performance issue for a data prefetcher to bring in missing cache lines in advance. On the other hand, a GPU clock can be slower than a CPU clock ($\frac{1}{2}$ CPU frequency assumed in this study), albeit an integrated design could bring the GPU up to the CPU's speed. In our more pessimistic assumption, we could waste 16 CPU cycles between two successive VLIW bundles of a COMPASS shader.

On the other hand, one limitation of COMPASS is its throughput. COMPASS is based on a compute-shader, which may need tens of instructions to emulate a hardware prefetcher. Before a shader code completes its execution, another run of the shader code mapped to the same hardware thread ID cannot be dispatched. Furthermore, a GPU may not have enough command queue to have many different shaders. (Note that this is not a queue for threads spawned from a single shader execution command, but a queue for different shader execution commands.) The maximal number of shader codes that can be queued in our baseline GPU is 16. Once this queue is full, the shader execution command from MAP is ignored until the queue releases a slot for a new shader execution. These two issues, if left unaddressed, will make the idea of COMPASS less useful. We will focus on them in describing our COMPASS shaders.

6.4.1 Stride COMPASS

In the implementation of PC-indexed stride COMPASS, we use three registers to keep a PC tag, its associated last miss address, and stride. Upon a cache miss, the setup engine uses a PC given by MAP to generate an index, which is used as a hardware thread ID for our COMPASS shader. Once a thread is selected, it compares whether a requested PC matches to the tag value and whether a current miss address is equivalent to a previously predicted miss address. If both conditions are met, the shader code generates next d prefetch addresses where d is the prefetch depth.[2] To avoid branch instructions, we use predication for condition checking and unroll the loop to generate d prefetch addresses.

We have two different types of stride COMPASS shaders to trigger d prefetches. One is a single-threaded stride COMPASS, which activates only one thread upon a

[2]In this study, we follow the prefetching terminology used in [92]. For example, a prefetcher with a prefetch depth of four prefetches four cache lines that will likely be requested by four consecutive misses in the future. On the other hand, a prefetcher with a prefetch width of four prefetches four potential cache line candidates that will likely be requested by the next miss.

miss. In this case, each entry of the prefetch table is modeled with one thread. This thread consumes three registers to maintain miss history and generates d prefetches itself. The utilization of the GPU that runs this shader code is very low since we only enable one out of 64 threads in a wavefront. The second design is a multi-threaded stride COMPASS, which activates d threads within a wavefront. Although these d threads emulate just one prefetch table entry, the same miss history (*e.g.*, tag, last miss address and stride stored in three registers) needs to be duplicated for each thread. Since each thread only generates one prefetch request, it requires only one temporary register for the prefetch address of the thread compared to d temporary registers of a single-threaded stride COMPASS shader. Thus, the number of temporary registers required per thread is smaller, and the multi-threaded stride COMPASS shader can be completed sooner, thereby reducing the latency and increasing the throughput. When d is not in power-of-two, we made a group of D threads to emulate a prefetch table entry where D is the smallest number in power-of-two greater than d. For example, to emulate a stride prefetcher with a prefetch depth of five, we group eight threads into one group so that this group can emulate a prefetch table entry. In this case, three remaining threads are never activated. Such inactive threads are found to be another source of storage inefficiency.

6.4.2 Markov COMPASS

In contrast to a PC-indexed stride prefetcher, a Markov prefetcher [60] uses a miss address to index a prefetch table. Thus, the setup engine uses a current miss address to index a thread (for prefetching) and the last miss address to index another thread (for updating the table) (Figure 39). Here, the command processor stores the miss address of the last execution and provide it as the last miss address to the setup engine for updating the table upon receiving a new miss address from the MAP. Each thread of our Markov COMPASS shader maintains w next addresses, where w

denotes the prefetch width. Our Markov COMPASS maintains these w next addresses in a FIFO manner. One advantage of Markov COMPASS over a hardware Markov prefetcher is its programmability for using different prefetch width. For example, an application that heavily uses a binary tree favors a Markov prefetcher with a prefetch width of two (or three if a pointer to a parent node is required). On the other hand, another application that heavily uses a singly-linked list favors a Markov prefetcher with a prefetch width of one because reducing the size of each entry allows more entries to be emulated. Therefore, an application vendor can configure (dynamically) the appropriate prefetch width of their own COMPASS shader according to program behavior.

```
1: if tag == currentAddr then                          ▷ Prefetch shader
2:     prefetch  nextAddr0
3:     prefetch  nextAddr1
4:     prefetch  nextAddr2
5: end if
6: tag ← currentAddr
```

```
1: nextAddr2 ← nextAddr1                               ▷ State update shader
2: nextAddr1 ← nextAddr0
3: nextAddr0 ← currentAddr
```

Figure 39: Markov prefetching (prefetch width: 3).

6.4.3 Delta COMPASS

The third prefetcher we evaluated is a PC-indexed delta prefetcher [63, 92]. The pseudo code is depicted in Figure 40. As shown, implementing delta COMPASS is found to be more challenging because the process of delta correlation matching is complicated, and we have to accumulate delta values to calculate prefetch addresses. Considering that a minimum interval for dispatching two successive VLIW bundles of a thread is eight GPU clock cycles, the latency of delta COMPASS implemented with a single thread will be very high.

To improve its efficiency, we implemented multi-threaded delta COMPASS. For example, to emulate a delta prefetcher with eight delta buffers, we use eight threads; each has a part of the delta buffer and performs delta correlation matching in parallel.

```
1:  for i ← depth − 1 to 1 do                                  ▷ State update
2:      δ_i ← δ_{i−1}
3:  end for
4:  δ_0 ← currentAddr − lastAddr
5:  lastAddr ← currentAddr
6:
7:  for i ← depth − 1 to 2 do                                  ▷ Delta correlation matching
8:      if (δ_0 == δ_i)&&(δ_1 == δ_{i+1}) then
9:          break
10:     end if
11: end for
12:
13: prefAddr ← currentAddr                                     ▷ Prefetch address calculation
14: for j ← i − 1 to 0 do
15:     prefAddr ← prefAddr + δ_j
16:     prefetch   prefAddr
17: end for
```

Figure 40: Delta prefetching (δ_i: i^{th} entry of a delta buffer).

In particular, thread i ($0 \leq i \leq 7$) keeps δ_0, δ_1, δ_i, and δ_{i+1} where δ_n is the n^{th} entry of the delta buffer. To update the state, thread i passes δ_{i+1} to thread $(i+1)$ through LDS (Figure 33), so that the delta buffer can be synchronized globally. To perform correlation matching, thread i compares a pair of δ_0 and δ_1 against a pair of δ_i and δ_{i+1}. Once a thread finds a match, it broadcasts its thread ID[3].

However, calculating a prefetch address in each thread is still challenging because thread j ($j < i$) requires the shader code to perform a reduction of $\sum_{n=j}^{i-1} \delta_n$, which should be accumulated after finding a match. This accumulation requires sequential scanning among threads, leading to elongated latency and lowered throughput. To avoid this loop, we contrive the delta COMPASS shader so that thread j also maintains another value, $\sum_{n=0}^{j-1} \delta_n$, which we call delta sum. Basically, delta sum is a sum of the last j delta values. Thread j can easily maintain this value by accumulating $(\delta_0 - \delta_j)$ upon a miss or upon the first hit to the prefetched line. Once thread i finds a correlation match, this thread broadcasts its own delta sum, $\sum_{n=0}^{i-1} \delta_n$, along with its thread ID. Once thread j receives this value, it can calculate $\sum_{n=j}^{i-1} \delta_n$ by subtracting its own delta sum, $\sum_{n=0}^{j-1} \delta_n$, from broadcast thread i's delta sum, $\sum_{n=0}^{i-1} \delta_n$. With this modified algorithm, we can eliminate the iterative accumulation process from each

[3]Multiple threads may find a match when the length of a correlation sequence is short, but here we use a special broadcast instruction that broadcasts a value from the first valid thread in a wavefront to all threads of the wavefront [1].

Table 5: A modified delta prefetch address calculation algorithm (miss address = 1024).

	thread 0	thread 1	thread 2	thread 3	thread 4	thread 5	thread 6	thread 7
Delta (δ_j)	1	2	1	2	1	2	3	4
Delta Sum ($\sum_{n=0}^{j-1} \delta_n$)	0	1	3	4	6	7	9	12
Match?	O	X	O	X	O ($i = 4$)	X	X	X
$\sum_{n=j}^{i-1} \delta_n$	6	5	3	2	disabled	disabled	disabled	disabled
Prefetch Address	1030	1029	1027	1026	disabled	disabled	disabled	disabled

shader execution.

A detailed example is shown in Table 5. If we use the original algorithm (Figure 40), thread4 finds a match and broadcasts its thread ID to all other threads. After receiving the thread ID of four, thread1, for example, needs to accumulate the delta value of thread3 (= 2), that of thread2 (= 1), and that of thread1 (= 2). The sum of these values, 5 (= 2 + 1 + 2), are added to a current miss address, 1024, to calculate a prefetch address, 1029. On the other hand, in our modified algorithm, each thread keeps updating the delta sum when it updates its local delta values. When thread4 finds a match, it broadcasts its thread ID and its own delta sum (= 6) to all other threads. Upon receiving this value, thread1 subtracts its own delta sum (= 1) from the receive delta sum (= 6) and adds this difference (= 5) to the miss address, 1024, to calculate the prefetch address, 1029.

6.4.4 A Simplified Region Prefetcher

Additionally, we also evaluated a simplified version of a region prefetcher [74]. Although the original region prefetching technique monitors the utilization of a memory channel and prefetches an entire page while the channel is idle, we cannot perform such fine-grained monitoring and prefetching because we do not want to heavily modify the GPU design. Thus, in this study, we simplified the design by fetching a page upon the first touch of a certain page. Basically, the setup engine uses a miss address to index a group of 64 threads or a wavefront. In particular, as shown in Figure 41,

thread i of a wavefront prefetches $P + 64 \times i$ where P denotes the base address of a page to which a requested miss belongs. In other words, 64 threads of the wavefront prefetches 64 cache lines that belongs to the same page. (In this study, the L2 cache line size is 64B, and a page size is 4KB.) Such a brute-force, non-controlled prefetching technique cannot be used in a conventional prefetcher, which will affect the performance of all applications. However, due to its flexibility, a COMPASS shader enables such an aggressive technique whenever an algorithm has a demand for.

```
1: pageAddr ← currentAddr & 0xfffff000
2: for i ← 0 to 63 do
3:     prefAddr ← pageAddr + 64 × i
4:     prefetch   prefAddr
5: end for
```

Figure 41: Region prefetching.

6.4.5 Custom COMPASS Design for 429.mcf

To demonstrate and evaluate a custom COMPASS shader, we selected 429.mcf from SPEC2006 for our case study. It is known that 429.mcf demonstrated a peculiar memory access pattern [28, 109]. The address strides of several PCs missing the L2 are increased exponentially as shown in Table 6 making them hard to be recognized by most hardware prefetchers. To capture the exponential stride pattern, we designed a COMPASS shader that performs exponential stride prefetching. The pseudo code is given in Figure 42. Our exponential stride prefetcher maintains the last miss address and the last stride value. If a current stride value meets the condition shown in line 1 of the figure, we prefetches multiple cache lines using exponential strides. We multi-threaded this COMPASS shader so that thread i can fetch three lines that are fetched by iteration i of a loop shown in Figure 42. To reduce the latency of this computation, we avoid the accumulation process in this loop by designing thread i to compute the accumulated sum of exponential strides by itself. Basically, instead of calculating $\Sigma_{n=1}^{i} 2^n \delta_0$, each thread calculates $2 \times (2^i - 1)\delta_0$ which is equivalent to the previous equation, where δ_0 is a current stride value.

Table 6: A sample stride pattern of 429.mcf (cache line size: 64B).

Miss Address	0x4b19ba40	0x4b19e080	0x4b1a2d40	0x4b1ac680	0x4b1bf900	0x4b1e5e40	0x4b2328c0
Stride (δ_j)		9792	19648	39232	78464	156992	313984
$\delta_j - 2 \times \delta_{j-1}$			64	-64	0	64	0

Note that, prior research [109] had attempted to capture such access patterns in a hardware prefetcher, yet often resulting in prohibitively large hardware structure, thus is impractical. Again, the programmable COMPASS shader will enable such prefetching in an economical manner.

```
 1: δ1 ← δ0
 2: δ0 ← currentAddr − lastAddr
 3: lastAddr ← currentAddr
 4:
 5: if (δ0 ≥ 2δ1 − 64)&&(δ0 ≤ 2δ1 + 64) then
 6:     stride ← δ0
 7:     prefAddr ← currentAddr
 8:     for i ← 0 to depth − 1 do
 9:         stride ← stride × 2
10:         prefAddr ← prefAddr + stride
11:
12:         prefetch   prefAddr
13:         prefetch   prefAddr + 64
14:         prefetch   prefAddr − 64
15:     end for
16: end if
```

Figure 42: Exponential stride prefetching (δ_0: a current stride, δ_1: the last stride).

6.5 Experimental Results

In this section, we will first describe our simulation framework and evaluate each COMPASS design. For each design, we will employ one or two applications that clearly reveals the difference (or the design trade-off) between different COMPASS shaders. After that, we will show overall results with our benchmark applications. Lastly, we will discuss hardware, software, and power overhead of COMPASS.

6.5.1 Simulation Framework

We evaluate COMPASS by extending the SESC simulator [105]. In particular, we use an existing CPU model of SESC as well as its memory back-end. In addition, we integrated a GPU pipeline to perform cycle-level simulation. Our GPU simulation

Table 7: Platform configurations.

CPUs	Processor model	Two 3.0GHz, 14-stage, out-of-order, 4-wide fetch/issue/retire superscalar processors, 192 ROB entries, 128 (INT) + 128 (FP) physical register file
	Branch predictor	Hybrid branch predictor (16K global / local / meta tables), 2K BTB, 32-entry RAS
	L1 instruction cache	dual-port 2-way set-associative, 64B-line, 32KB cache, LRU policy, 1 cycle latency, 1 cycle throughput, 8-entry MSHR
	L1 data cache	dual-port 4-way set-associative, 64B-line, 32KB write-back cache, LRU policy, 2 cycle latency, 1 cycle throughput, 8-entry MSHR
GPU	Clock frequency	1.5 GHz
	Front-End	Command Processor (4 GPU cycle latency) + Setup Engine (3 GPU cycle latency)
	Ultra-Threaded Dispatch Processor	2 arbiters per SIMD array, 1 arbiter per TEX unit, minimum 1 GPU cycle latency, FIFO scheduling policy
	Array Size	4 SIMD arrays, 16 shader cores per SIMD array, 5 SPUs per shader core
	Register file	4-banked, 16KB per shader core (or 256KB per SIMD array)
	TEX unit	One per SIMD array, 4 address processors per TEX unit
	TEX L1 Cache	1 cycle latency, 4-way set-associative, 64B-line 32KB (tightly coupled with a TEX unit)
Memory Back-End	Shared L2 cache	dual-port, 4-banked, 8-way set-associative, 64B-line, 2MB inclusive write-back cache, LRU policy, 6 cycle latency, 1 cycle throughput, 8-entry MSHR per bank
	Baseline prefetcher	Next-line prefetcher in the L2 cache
	Memory	350-cycle minimum latency, four 8B-wide buses, 800 MHz clock, double-data-rate

model includes the latency model of GPU front-end, the FIFO scheduling policy and the limited queue entries of the ultra-threaded dispatch processor, the latency and throughput model of VLIW execution inside each shader core, and the latency and throughput model of L1 TEX caches that are connected to the shared L2 cache. Figure 35 illustrates the overall system architecture of our CPU-GPU integrated platform, and its details are listed in Table 7.

To quantify the performance advantage of COMPASS, we use memory-intensive applications from SPEC2006. We define a memory-intensive application as an application whose speedup with a perfect L2 cache is greater than 1.1x compared to a baseline CPU with a next-line prefetcher. For computation-intensive applications, as COMPASS is fully programmable, an OS can opt for not using it. As such, COMPASS will not adversely affect the performance for these applications. We also excluded 434.zeusmp, 465.tonto, and 470.lbm due to cross-compilation issues or unsupported syscalls in SESC. For each experiment, we fast-forwarded the first 10 billion instructions and measured the performance for the next billion instructions unless

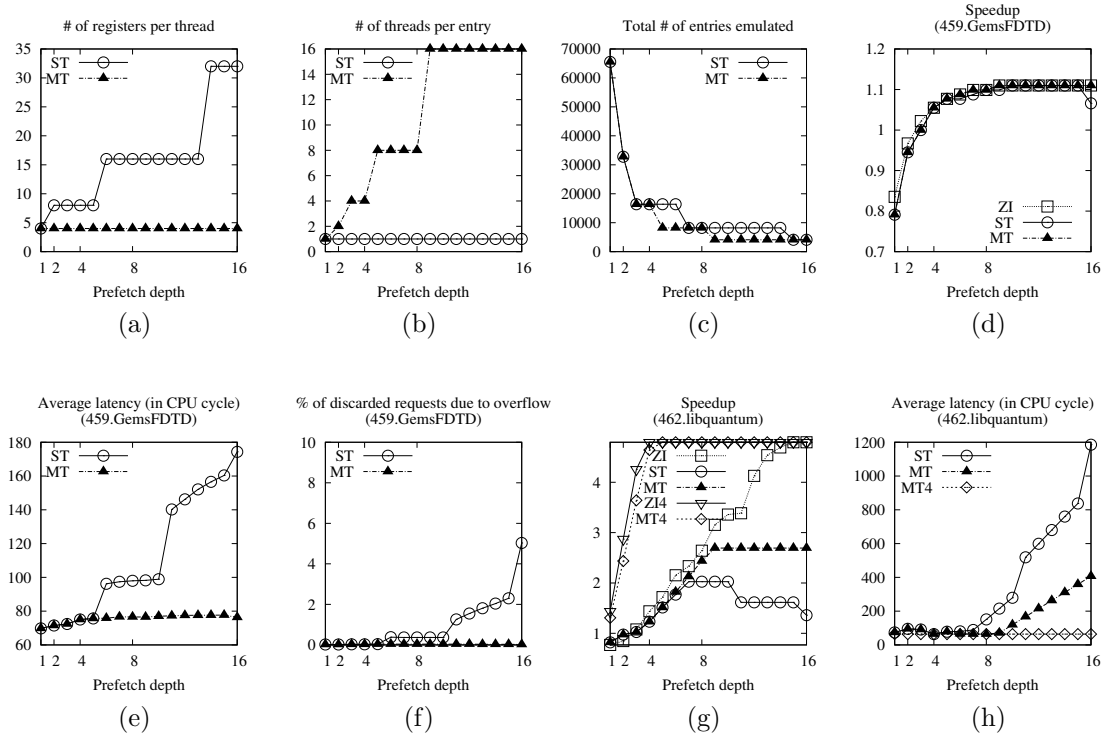


Figure 43: Evaluation of Stride COMPASS (ST: single-threaded, MT: multi-threaded, ZI: zero-latency, infinite-throughput).

otherwise mentioned. Throughout this study, the baseline has a next-line prefetcher associated with its L2 cache.

6.5.2 Evaluation of Stride COMPASS

To evaluate the stride COMPASS, we performed a vast number of simulations with different types of shader codes and different prefetch depth. First of all, Figure 43(a) and Figure 43(b) show the number of required registers per thread and the number of threads required for simulating one prefetch table entry, respectively. As shown, as the prefetching depth increases, a single-threaded (ST) stride COMPASS shader requires more registers per thread. On the other hand, a multi-threaded (MT) stride COMPASS shader requires more threads to emulate one entry while the number of

required registers per thread remains constant. This trade-off results in different numbers of table entries that can be emulated with COMPASS as shown in Figure 43(c). The figure suggests that, in general, an ST stride COMPASS shader can emulate more table entries than an MT stride COMPASS shader. Note that both stride COMPASS shaders can emulate at least 4096 entries. Assuming that an entry of a hardware-based stride prefetcher table is 12B wide (including a tag, a last address, and a stride value), our stride COMPASS with four SIMD arrays can emulate the behavior of a 48KB hardware stride prefetch table.

Although both of stride COMPASS shaders, ST and MT, emulate a larger stride prefetcher than a typical hardware-based counterpart, their performance may not be as high due to the longer latency and lower throughput. To observe this effect, we performed another set of simulations with a zero-latency, infinite-throughput (ZI) model that has the same number of table entries as the MT stride COMPASS. Figure 43(d) shows the speedup of these three models that run 459.GemsFDTD. As can be seen, both ST and MT models underperform slightly but will catch up with the ZI model if their prefetching depth is deep enough. This can be explained by analyzing Figure 43(e), which shows the average latencies of ST and MT models, and Figure 43(f), which shows the number of discarded requests due to queue overflow (explained in Section 6.4). We do not show the latency and the number of discarded requests for the ZI model since both are zero by their definition. As shown in these figures, when a GPU emulates a stride prefetcher for 459.GemsFDTD, the latency of the COMPASS shader is low enough to make both ST and MT models to catch up with the ZI model when they prefetches more deeply. Besides, the queue occupied by pending miss requests hardly overflows, thus lost miss history due to discarded requests does not affect the overall performance of 459.GemsFDTD significantly.

In contrast, when running 462.libquantum, we found that the ST and the MT stride COMPASS shader cannot achieve the performance of the ZI model as illustrated

in Figure 43(g). Moreover, the ST stride COMPASS shader performs worse than the MT because of its longer latency as shown in Figure 43(h) even though there is no hurdle for ST to emulate a large prefetch table. Also shown in Figure 43(g), once the prefetch depth of the MT stride COMPASS exceeds nine, its improvement levels off with a increasing higher latency shown in Figure 43(h) due to a longer queuing delay within the GPU. After analyzing the simulation trace, we found that a *single PC* of 462.libquantum constantly generates a cache miss. Consequently, our PC-indexed stride COMPASS shader is serialized, failing to use four SIMD arrays or other threads available in the same SIMD array. The following analysis quantitatively explains this phenomenon well. First of all, we found that the saturated IPC is 1.05, and the corresponding L2 misses per kilo instructions (MPKI) is 19.77. In other words, it takes 952.38 cycles to execute 1000 instructions that generate 19.77 misses. Thus, the average time interval between two successive misses is 48.17 cycles while it takes 48 cycles to execute three VLIW bundles for our stride COMPASS shader. In short, our COMPASS shader cannot prefetch cache lines in a timely manner once the IPC of 462.libquantum reaches 1.05.

To mitigate this problem due to address aliasing, we also evaluated a different index function that concatenates the PC with two bits from a cache miss address (bit 7 and bit 6 of the miss address). As we use four different entries for a single PC, the stride of each entry will be 256 while the commonly observed stride of a stride prefetcher with the conventional index function was 64. Figure 43(g) also shows the speedup of a zero-latency, infinite-throughput (ZI4) and MT stride COMPASS shader (MT4) that use this new indexing method. As shown, the speedup has been largely improved compared to our previous design. The reason why the speedup of ZI4 with a prefetch depth of four approaching that of ZI with a prefetch depth of 16 is due to their functional equivalence, i.e., ZI4 prefetches a line 4 (depth) $\times$ 256 (stride) byte away from the current miss address while ZI prefetches a line 16 (depth) $\times$ 64 (stride)

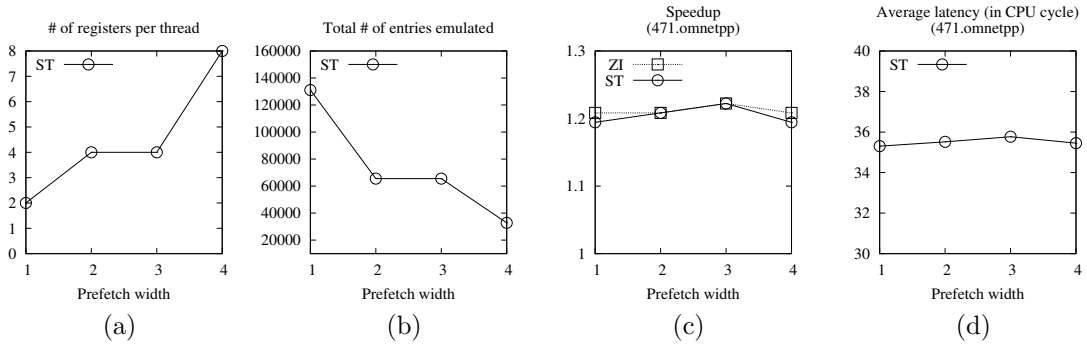


Figure 44: Evaluation of Markov COMPASS (ST: single-threaded, ZI: zero-latency, infinite-throughput).

byte away.

Another noteworthy function is that an L1 TEX cache attached to each SIMD array can combine redundant prefetch requests issued by COMPASS. For the MT stride COMPASS with a prefetch depth of 16 for 462.libquantum, the total number of L1 TEX cache accesses is 316.3 million while the total number of actual prefetches reaching the L2 is 19.8 million. Note that all these memory accesses are prefetch requests because our COMPASS shader code relies only on its own registers for other computation such as recording prefetch history and computing next prefetch addresses.

6.5.3 Evaluation of Markov COMPASS

In contrast to the stride COMPASS for which we varied the prefetch depth, we evaluate the Markov COMPASS shaders by varying their prefetching width. Because a Markov COMPASS shader is very simple, we only evaluate a single-threaded Markov prefetcher. The cost of emulating those different Markov prefetchers are shown in Figure 44(a) and Figure 44(b), in which the number of required registers and the capacity of the emulated Markov prefetch table are plotted with the prefetch width. Due to the fact that the number of registers or the number of entries must be in power-of-two for indexibility, a Markov prefetcher with a prefetch width of two will contain unused

resources while consuming the same resources with the one with a prefetch width of three as shown in Figure 44(b). In our experiments, a Markov prefetcher with a prefetch width of three turns out to be the most effective for two applications: 471.omnetpp and 483.xalacbmk that benefit from a Markov prefetcher. The speedup of these two applications are found to be 1.22 and 1.18, respectively while the performance of the other applications are degraded. For 471.omnetpp shown in Figure 44(c), a Markov COMPASS shader with a prefetch width of one did not outperform the cases with wider prefetch width although it can emulate more table entries. On the other hand, a Markov COMPASS shader with a prefetch width of four reduces the speedup since the number of table entries that can be emulated has significantly dropped. The average latencies of these Markov COMPASS with different prefetch width are found to be similar as shown in Figure 44(d).

When comparing the capability of our Markov COMPASS to a conventional hardware-based prefetcher, our scheme has a huge advantage. For example, a Markov COMPASS with a prefetch width of three emulates 65,536 table entries, each containing 16B (a tag plus three next miss addresses). That amounts to a prohibitively expensive 1MB hardware prefetch table.

6.5.4 Evaluation of Delta COMPASS

As explained in Section 6.4.3, we evaluate only a multi-threaded delta COMPASS shader due to the computation complexity of a delta prefetcher. In our delta COMPASS implementation, each thread uses 16 registers. As shown in Figure 45(a), the capacity of emulated prefetch tables is much larger (at least 1024 entries) than that of a conventional hardware delta prefetcher. Figure 45(b) shows the speedup of 450.soplex using delta COMPASS shaders with varying prefetch depth. Like a stride COMPASS shader, as the prefetch depth increases, the MT delta COMPASS shader can catch up with the ZI delta COMPASS in spite of increasing latency (Figure 45(c)).

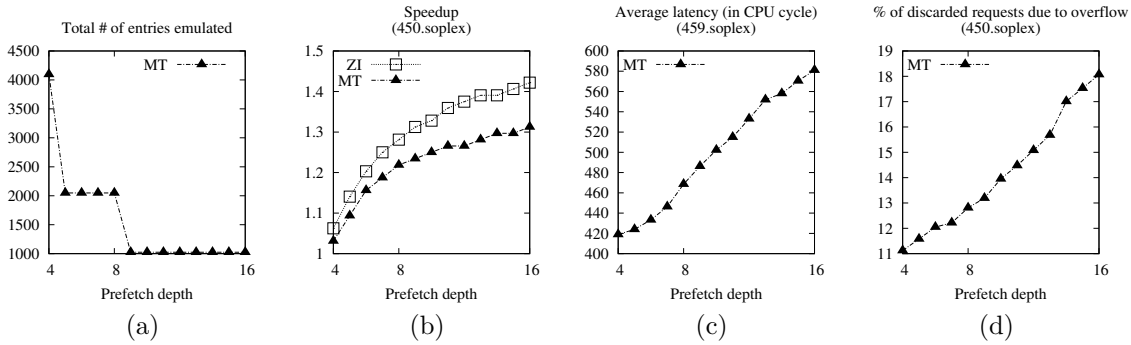


Figure 45: Evaluation of Delta COMPASS (MT: multi-threaded, ZI: zero-latency, infinite-throughput).

Meanwhile, because a delta COMPASS shader executes a rather large shader code, its throughput is low. Thus, due to overflow, more miss requests are discarded as the prefetch depth gets larger as shown in Figure 45(d). Notwithstanding the discarded misses, Figure 45(b) shows that a delta prefetcher trains itself pretty rapidly resulting in high performance.

6.5.5 Evaluation of Region COMPASS

Unlike previous COMPASS designs, a simplified version of a region prefetcher requires a small number of registers with a lightweight COMPASS shader code. To fetch 64 cache lines mapped to the same page, our region COMPASS uses 64 threads of a wavefront, each of which calculates a corresponding cache line using its own thread ID. This computation is extremely simple and requires only four registers per thread. Similar to Markov COMPASS, we use the miss address to index a thread. This decision is to efficiently prefetch cache lines even if a single PC constantly generates cache misses very frequently. Because there is not much trade-off in this design, we do not show further in-depth sensitivity study for it. Its overall benefit will be discussed in Section 6.5.7.

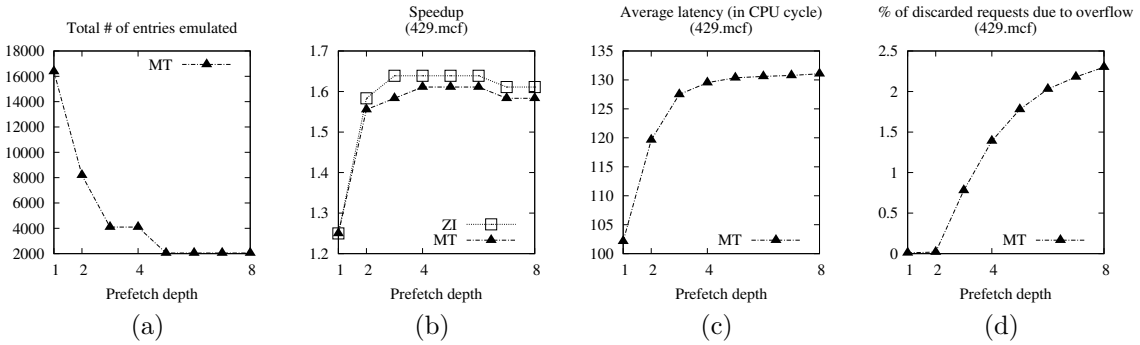


Figure 46: Evaluation of custom COMPASS for 429.mcf (MT: multi-threaded, ZI: zero-latency, infinite-throughput).

6.5.6 Evaluation of Custom COMPASS

We also evaluated a custom COMPASS shader for 429.mcf. This shader code is PC-indexed and uses 16 registers per thread. We varied the prefetch depth for this multi-threaded COMPASS shader. The number of table entries that can be emulated is shown in Figure 46(a). With such a custom COMPASS shader, the performance of 429.mcf is improved by 61% when the prefetching depth of this custom COMPASS shader is four (Figure 46(b)). On the other hands, as shown in Figure 46(c) and Figure 46(d), the latency of this COMPASS shader is around 130 cycles, and the number of discarded requests accounts for around 2% of L2 cache misses.

6.5.7 COMPASS vs. GHB Stride Prefetchers

Table 8 summarizes our simulation results. Instead of showing all results with different prefetching depth or width, we selected the best performing COMPASS shader in each type of COMPASS. Here, we also show the performance of a conventional hardware-based stride global history buffer (GHB) prefetcher (denoted as HW GHB) that has a 256-entry global history buffer, the same size used in the original paper [92]. We use three stride GHB prefetchers with prefetch depth of 4, 8, and 16. Note that a stride GHB prefetcher was found to be the most efficient prefetcher in a previous study [99].

Table 8: Overall results (d: prefetch depth, w: prefetch width).

Speedup		SPECint06					SPECfp06						Geo-mean
		429. mcf	462. lib-quan-tum	471. omnet-pp	473. astar	483. xalan-cbmk	410. bwaves	433. milc	437. leslie-3d	450. soplex	454. cal-culix	459. Gems-FDTD	
HW GHB	Stride (d=4)	0.89	1.44	1.15	1.00	0.96	1.80	1.11	1.27	1.06	1.11	1.05	1.15
	Stride (d=8)	0.89	2.85	1.17	1.04	0.93	1.89	1.11	1.28	1.11	1.11	1.10	1.24
	Stride (d=16)	0.89	4.79	1.15	1.05	0.93	1.80	1.09	1.27	1.14	1.11	1.10	1.29
COMPASS	Stride (d=16)	1.06	2.69	1.15	1.07	0.93	1.84	1.15	1.27	1.34	1.12	1.11	1.28
	Markov (w=3)	0.97	0.74	1.22	0.82	1.18	0.73	0.87	0.69	0.86	0.91	0.60	0.85
	Delta (d=16)	1.33	1.13	1.15	1.05	0.93	1.61	4.22	1.26	1.31	1.10	1.70	1.38
	Region	2.00	3.49	0.25	1.09	0.82	1.46	1.59	1.23	1.47	1.12	0.30	1.07
	Custom (429.mcf) (d=4)	1.61	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Perfect	Perfect L2	2.56	4.82	1.72	1.52	4.07	1.93	4.61	1.30	2.08	1.12	2.08	2.25
												COMPASS Geomean (when properly selected)	1.68

As shown in the table, the stride GHB prefetchers work pretty well with 462.libquantum and 410.bwaves, however, they may also degrade performance, *e.g.*, 429.mcf. This is where the strength comes from by employing our programmable COMPASS. As mentioned earlier, COMPASS allows an application vendor or the OS to select the best performing prefetcher by customizing a GPU prefetch shader code according to the peculiar behavior of an application. Even with the small benchmark program sample shown in Table 8, there exists no one-size-fits-all best prefetcher for them. Flexibility of a programmable COMPASS facilitates the choice of the best prefetching strategy. As shown in the table, for the majority of the benchmark applications, flexibility allows COMPASS to perform better than a GHB prefetcher. Three exceptions are 462.libquantum, 410.bwaves, and 437.leslie3d, all of which are in favor of a stride prefetching technique. As explained in Section 6.5.2, for 462.libquantum, a stride COMPASS fails to catch up with a stride GHB prefetcher with prefetching depth 16 due to its low throughput. (Note that this stride COMPASS design is based on the PC-only indexing scheme. As mentioned in Section 6.5.2, the stride COMPASS shader with a different index function (MT4) can improve the performance as much as the GHB prefetcher does.) For 410.bwaves and 437.leslie3d, the performance difference is negligible. On the other hand, our custom COMPASS shader based on

Table 9: Dual-core simulation results.

Speedup		Benchmark pair					Geomean
		410 462	410 450	433 459	471 483	410 433	
HW GHB	Stride (d=4)	1.69	1.54	0.94	1.00	1.48	1.30
	Stride (d=8)	2.00	1.63	0.98	1.01	1.57	1.38
	Stride (d=16)	2.30	1.59	0.94	1.00	1.53	1.39
COMPASS		1.86 (stride) (d=16)	1.64 (stride) (d=16)	2.01 (delta) (d=16)	1.05 (markov) (w=3)	2.09 (delta) (d=16)	1.68
Perfect L2		2.81	2.22	2.82	2.44	2.88	2.62

exponential increased stride for 429.mcf outperforms all other table-based prefetchers, although a brute-force, region COMPASS is found to perform even better than the custom COMPASS shader.

On average (geometric mean), the stride GHB prefetchers with the prefetching depth of 4, 8, and 16 improve performance by 15%, 24%, and 29%, respectively. In contrast, if an application vendor or an OS chooses the application-specific COMPASS shader, we can improve performance by 68% on average.

6.5.8 Multicore Effects

The majority of data prefetching techniques have been focused on improving the performance of single programs. It is not well understood whether we need to partition a prefetch table into several sub-tables or we should allow different processes to share one table but potentially interfere the miss history of each other. Clearly, this study is a separate piece of work. Thus, in this section, we only briefly discuss how the OS can use COMPASS in a platform with two CPU cores and a GPU. For this platform, obviously, the best way to handle this problem is to co-schedule a computation-intensive application and a memory-intensive application [121]. This is not only good for reducing pollution in a prefetch table but also good for reducing contention in the L2. Several shared cache-aware scheduling methods have been investigated [35, 34, 103, 68]. Another way to address this problem is to prioritize one application. In this case, the highest priority application will run COMPASS alone while other

concurrent applications will not benefit.

On the other hand, we can schedule two memory-intensive applications that benefit from the same type of COMPASS. For example, we can schedule 433.milc and 459.GemsFDTD, the performance of which is significantly improved by delta COMPASS. Even though these two applications share COMPASS, their performance can still be improved because the capacity of a prefetch table that delta COMPASS emulates is a lot larger than that of a conventional delta prefetcher. Table 9 shows results using this policy. Here, the baseline is again a system without COMPASS support (but with a next-line prefetcher as shown in Table 7). For these simulations, we fast-forwarded the first 20 billion instructions and measured the performance of the next two billion instructions. Here again, we also show simulation results with the hardware-based stride GHB prefetchers for comparison. The first set of applications, 410.bwaves and 462.libquantum, in the table represents two applications that benefit a lot from a stride COMPASS shader. As shown in the table, a stride COMPASS shader can improve performance significantly. As explained previously, because a PC of 462.libquantum constantly generates cache misses very frequently, the speedup of COMPASS is not as high as that of the GHB prefetchers. However, when we simulate another two applications, 410.bwaves and 450.soplex, the performance of a stride COMPASS is as good as that of GHB prefetchers. In contrast, when we schedule two applications, 433.milc and 459.GemsFDTD, that benefit from a delta COMPASS, our COMPASS clearly outperforms GHB prefetchers by simply programming the GPU.

On the other hand, when 471.omnetpp and 483.xalancbmk, which benefit from the Markov COMPASS, are scheduled together, the Markov COMPASS outperforms the GHB prefetchers. However, the speedup of these two applications are actually lower than the speedup of each application measured in a single-core simulation. This outcome is not surprising because a Markov prefetcher usually requires a large

table and because these applications share the same capacity as the previous single-core simulations. Lastly, when 410.bwaves and 433.milc, which benefit most from the stride COMPASS and delta COMPASS, respectively, are scheduled together, the delta COMPASS can still improve performance significantly. This is because a typical delta prefetcher can do the same job as a stride prefetcher by simply consuming larger table space.

In summary, COMPASS has potentials to improve the performance of multiple cores if the OS schedules applications wisely. As mentioned, an OS-level scheduling policy is out of scope of this study, and it remains as our future work. Note that as the number of CPU cores scales in the future, the number of SIMD arrays will scale as well, thus a GPU will be able to provide enough throughput for prefetching and to emulate a larger prefetch table.

6.5.9 Hardware, Software, and Power Overhead

The hardware overhead of COMPASS includes the following. (1) The MAP requires three registers to temporarily store a shader pointer, a miss PC, and a miss address. (2) It needs a command buffer, which stores the GPU execution command. Note that the command buffer stores the command itself, not the code for COMPASS shader. Thus, this buffer is extremely small. (3) The command processor of a GPU needs to be modified to understand the new GPU execution command. This overhead is basically a microcode patch because the command processor is a programmable processor. (4) The setup engine should be able to index a thread based on a value retrieved from the command. The overhead of this index function is minimal because it is basically a simple masking operation. (5) Each TEX unit should support a prefetching operation. This is also a negligible change because we just need to disable the write-back path of a conventional GPU load operation. (6) The address translation process through a GPU's TLB should be disabled because MAP is forwarding a physical address from

the L2 cache and we can simply use it without address translation.

On the other hand, to provide a COMPASS shader to MAP, an OS needs to have one or multiple COMPASS shaders built-in so that it can use one of these COMPASS shaders along with CPU processes. Optionally, the OS can provide an API function that allows an individual application vendor to provide its custom-designed COMPASS shaders. Second, the task scheduler of the OS should be able to enable the MAP.

Furthermore, to evaluate the energy implication of COMPASS, we first modeled the utilization of GPU as follows:

$$
\begin{aligned}
(\mathit{Utilization}) = {} & (\#\text{ of COMPASS shader execution}) \\
& \times (\#\text{ of required threads per COMPASS shader execution}) \\
& \times (\#\text{ of required VLIW bundles per thread}) \\
& /(\text{total execution time in cycle}) \\
& /(\text{the number of VLIW cores})
\end{aligned}
$$

Table 10 shows the utilization of different COMPASS shaders. Here, the gray boxes represent the best performing COMPASS shaders as explained earlier. As shown in the table, the utilization of COMPASS is extremely low. The highest utilization is 1.49%, and the average (geometric mean) utilization is found to be 0.20%. Because a COMPASS shader is executed upon an L2 cache miss, which is a very rare event, and because a COMPASS shader relies on a very small subset of available hardware threads, utilization is very low.

Based on these utilization results, we also estimated the average power of the GPU. Here, we modeled the leakage power of the total 1MB register file using CACTI 5.3 [119]. According to this method, we estimated that the 1MB register file will consume 0.96W of leakage power. On the other hand, we estimated the average dynamic

Table 10: GPU utilization.

%	SPECint06					SPECfp06					
	429. mcf	462. lib-quan-tum	471. omnet-pp	473. astar	483. xalan-cbmk	410. bwaves	433. milc	437. leslie-3d	450. soplex	454. cal-culix	459. Gems-FDTD
Stride (d=16)	0.0616	0.3906	0.1496	0.0620	0.0733	0.5656	0.1780	0.1435	0.2197	0.0370	0.2670
Markov (w=3)	0.0039	0.0066	0.0085	0.0031	0.0060	0.0186	0.0084	0.0051	0.0108	0.0019	0.0095
Delta (d=16)	0.2336	0.4931	0.3884	0.1862	0.2148	1.4880	1.9442	0.4245	0.6123	0.1105	1.1845
Region	0.3289	1.3398	0.1420	0.1715	0.2368	1.1980	0.6575	0.3712	0.8548	0.0986	0.3109
Custom (429.mcf) (d=4)	0.0411	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 11: GPU average power.

W	SPECint06					SPECfp06					
	429. mcf	462. lib-quan-tum	471. omnet-pp	473. astar	483. xalan-cbmk	410. bwaves	433. milc	437. leslie-3d	450. soplex	454. cal-culix	459. Gems-FDTD
Stride (d=16)	0.99	1.17	1.04	0.99	1.00	1.27	1.06	1.04	1.08	0.98	1.11
Markov (w=3)	0.96	0.96	0.97	0.96	0.96	0.97	0.97	0.96	0.97	0.96	0.97
Delta (d=16)	1.09	1.23	1.17	1.06	1.08	1.77	2.01	1.19	1.29	1.02	1.60
Region	1.14	1.69	1.04	1.05	1.09	1.61	1.32	1.16	1.42	1.01	1.13
Custom (429.mcf) (d=4)	0.98	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

power of the GPU based on TDP number of a real product, ATI Radeon 4650, which is 55W. To estimated the average dynamic power, we subtracted the leakage power number from the TDP number and multiplied this result by the utilization of the GPU. Note that this is a conservative approach because COMPASS shaders rely on simple integer operations rather than heavy floating point operations. Based on our model, as shown in Table 11, we found that GPU average power is very low due to the aforementioned low utilization. Note that leakage power accounts for most of the GPU power. Based on these results, we calculated energy additionally consumed by COMPASS shaders as shown in Table 12.

Furthermore, we also estimated energy consumed by the off-chip bus between an L2 cache and DRAM. For this estimation, we assumed that 32bit off-chip data transfer consumes 1300 pJ based on measurement data from Stanford University [97]. Based on this assumption, we estimated the energy consumption of the off-chip bus as shown in Table 13. From these results, we made a few interesting observations. First

Table 12: GPU energy consumption.

J	SPECint06					SPECfp06					
	429. mcf	462. libquantum	471. omnetpp	473. astar	483. xalancbmk	410. bwaves	433. milc	437. leslie3d	450. soplex	454. calculix	459. GemsFDTD
Stride (d=16)	0.88	0.37	0.42	0.56	1.28	0.23	0.66	0.21	0.42	0.20	0.36
Markov (w=3)	0.91	1.13	0.37	0.70	0.98	0.44	0.80	0.36	0.58	0.24	0.59
Delta (d=16)	0.76	0.92	0.47	0.60	1.36	0.36	0.35	0.24	0.51	0.21	0.34
Region	0.53	0.41	1.90	0.58	1.61	0.36	0.60	0.24	0.51	0.21	1.39
Custom (429.mcf) (d=4)	0.57	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 13: Off-chip bus energy consumption.

J	SPECint06					SPECfp06					
	429. mcf	462. libquantum	471. omnetpp	473. astar	483. xalancbmk	410. bwaves	433. milc	437. leslie3d	450. soplex	454. calculix	459. GemsFDTD
Nextline (baseline)	0.28	0.54	0.59	0.22	0.55	0.37	0.59	0.13	0.40	0.04	0.43
GHB (d=16)	0.26	0.54	0.22	0.23	0.33	0.37	0.90	0.13	0.38	0.04	0.53
Stride (d=16)	0.26	0.54	0.22	0.24	0.33	0.37	0.91	0.13	0.44	0.04	0.53
Markov (w=3)	0.32	0.54	0.17	0.19	0.33	0.56	0.49	0.14	0.59	0.04	0.51
Delta (d=16)	0.26	0.54	0.18	0.26	0.31	0.37	0.50	0.13	0.41	0.04	0.44
Region	0.79	0.54	8.82	1.34	6.98	0.37	2.08	0.15	0.99	0.04	6.21
Custom (429.mcf) (d=4)	0.53	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

of all, region COMPASS may consume a significant amount of energy in some cases (471.omnetpp, 483.xalancbmk, and 459.GemsFDTD), but interestingly enough, they are not the best performing configurations (represented in gray boxes). In these cases, region COMPASS clearly prefetches lots of unused data consuming lots of energy in the bus. However, those unused data not only consumes more energy but also pollutes the L2 cache seriously, which results in bad performance. This explains why those cases that consume a significant amount of energy in the bus is not a favorable configuration for the target application. Secondly, due to more accurate prefetching, COMPASS consumes less energy in the bus in some applications (471.omnetpp and 433.milc). In other words, the baseline prefetchers bring more unused data than the best performing COMPASS.

Additionally, we also modeled dynamic energy consumption of a CPU and L2 cache using the Wattch model [14] as shown in Table 14. Note that we did not model

Table 14: CPU and L2 cache energy consumption.

J	SPECint06					SPECfp06					
	429. mcf	462. lib-quan-tum	471. omnet-pp	473. astar	483. xalan-cbmk	410. bwaves	433. milc	437. leslie-3d	450. soplex	454. cal-culix	459. Gems-FDTD
Nextline (baseline)	16.16	16.51	15.05	12.76	19.92	15.39	17.99	10.51	14.58	10.33	14.69
GHB (d=16)	16.22	21.66	11.17	13.13	18.01	18.82	22.17	11.33	15.60	10.51	18.42
Stride (d=16)	15.54	14.03	11.15	12.91	17.95	14.65	20.62	10.25	14.08	10.19	15.47
Markov (w=3)	16.47	17.33	10.81	13.03	17.04	17.63	17.17	11.11	16.65	10.49	16.33
Delta (d=16)	14.74	16.00	10.72	13.12	17.72	14.81	14.43	10.26	13.98	10.21	14.00
Region	20.56	13.55	117.60	26.56	101.35	19.78	38.41	10.78	21.13	10.24	86.61
Custom (429.mcf) (d=4)	17.08	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 15: CPU, GPU, L2 cache, and off-chip bus energy consumption.

J	SPECint06					SPECfp06					
	429. mcf	462. lib-quan-tum	471. omnet-pp	473. astar	483. xalan-cbmk	410. bwaves	433. milc	437. leslie-3d	450. soplex	454. cal-culix	459. Gems-FDTD
Nextline (baseline)	16.44	17.05	15.65	12.97	20.47	15.76	18.58	10.63	14.98	10.37	15.12
GHB (d=16)	16.47	22.20	11.39	13.36	18.34	19.18	23.08	11.45	15.98	10.55	18.95
Stride (d=16)	16.67	14.94	11.79	13.71	19.56	15.24	22.20	10.59	14.94	10.43	16.36
Markov (w=3)	17.70	19.00	11.34	13.92	18.35	18.63	18.46	11.60	17.82	10.77	17.43
Delta (d=16)	15.76	17.76	11.37	13.98	19.40	15.54	15.27	10.63	14.90	10.46	14.78
Region	21.88	14.51	128.33	28.48	109.94	20.51	41.09	11.17	22.62	10.48	94.21
Custom (429.mcf) (d=4)	18.18	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

the leakage energy consumption of the CPU and L2 cache, but this is a conservative approach because COMPASS reduces execution time, which is helpful to reduce leakage energy consumption (compared to the baseline) in the CPU and L2 cache. Also note that, in our baseline model, we assumed that the GPU is completely turned off without consuming any leakage energy. Based on these results along with aforementioned GPU and bus energy results, we estimated overall energy consumption as shown in Table 15. As shown in the table, compared to the baseline, COMPASS often consumes more energy due to aggressive prefetching while it often consumes less energy due to more accurate prefetching.

In addition to the energy itself, we also calculated performance per joule to estimate energy efficiency. Normalized performance per joule is shown in Table 16. As shown in the table, in most cases, the best-performing COMPASS shader improves

Table 16: CPU, GPU, L2 cache, and off-chip bus energy efficiency (normalized perf/joule).

	SPECint06					SPECfp06					
	429. mcf	462. lib-quan-tum	471. omnet-pp	473. astar	483. xalan-cbmk	410. bwaves	433. milc	437. leslie-3d	450. soplex	454. cal-culix	459. Gems-FDTD
Nextline (baseline)	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
GHB (d=16)	0.89	3.68	1.58	1.02	1.04	1.48	0.88	1.18	1.07	1.09	0.88
Stride (d=16)	1.04	3.07	1.53	1.01	0.97	1.91	0.96	1.28	1.35	1.11	1.03
Markov (w=3)	0.90	0.67	1.69	0.77	1.31	0.61	0.87	0.63	0.72	0.88	0.52
Delta (d=16)	1.39	1.10	1.59	0.98	0.98	1.63	5.13	1.26	1.32	1.09	1.74
Region	1.50	4.10	0.03	0.50	0.15	1.12	0.72	1.17	0.97	1.11	0.05
Custom (429.mcf) (d=4)	1.46	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

performance per joule mainly due to its superior performance. We found two exceptional cases, 450.soplex and 473.astar. In case of 450.soplex, while region COMPASS improves performance a lot, it consumes much energy in the L2 cache. On the other hand, in case of 473.astar, performance gain is marginal. In this case, we may want to develop a better prefetching method. Note that, in both cases, the OS may select stride COMPASS to improve performance per joule when better energy efficiency is desired. Overall, we found that, for the configurations that provide the best performance, COMPASS can improve performance per joule by 59% on average (geometric mean). Furthermore, if the OS opts for optimizing the system performance for energy efficiency (better performance per joule), COMPASS can improve performance per joule by 74% on average.

6.6 Summary

In this chapter, we proposed COMPASS, a compute shader-assisted prefetching scheme, to improve the memory performance of an integrated chip while the on-chip GPU is idle. With very lightweight architectural support, COMPASS shaders can emulate various hardware prefetchers for improving performance of single-thread applications. Moreover, due to its programmability and flexibility, one can implement or select the best performing prefetching algorithm specially tailored for a specific application to

exploit its particular memory access behavior. We designed, evaluated, and analyzed different COMPASS shaders and also performed a case study to demonstrate a custom-designed COMPASS shader. Our simulation results showed that COMPASS can improve the single-thread performance of memory-intensive applications by 68% on average. With COMPASS, we envision that application vendors can supply an application-specific COMPASS shader bundled with their software in the future and have it loaded at runtime to improve memory performance.

CHAPTER VII

CONCLUSION

In this dissertation, we propose and evaluate different heterogeneous many-core processor designs to provide higher performance under the limited chip power budget. Achieving such a goal is often highly application-dependent, thus we propose various architectural techniques to further utilize on-chip resources even when an application that runs on a many-core processor is not necessarily a target application. The contribution of this dissertation includes the followings.

First of all, we proposed an analytical framework that helps computer architects to understand the energy efficiency of different types of many-core processor design styles in an early design stage, so that the large design space of a many-core processor can be effectively reduced. From this study, we found that, as the number of cores scales in the future, a heterogeneous many-core processor that consists of many energy-efficient cores along with one high-performance processor can achieve the best energy efficiency because it can provide highly energy-efficient parallel computing capability without losing its sequential computing capability.

Second, based on the previous observation, we proposed a heterogeneous many-core acceleration layer, called POD, that can be snapped on top of a conventional x86 processor layer with an emerging 3D-integration technology. By snapping such an acceleration layer on demand, industry can easily provide the acceleration functionality in addition to its general-purpose capability. Such a novel design reveals several challenges such as binary compatibility, efficient recovery from misspeculation, and an energy-efficient interconnection network. With these new challenges in mind, we

designed and evaluated the POD layer. From this study, we found that a heterogeneous many-core processor can outperform a symmetric, homogeneous many-core processor when they are constrained by the same single-chip power budget.

Third, while such a heterogeneous many-core processor can improve the performance of data-parallel applications, the heterogeneous many-core processor itself may not be highly utilized when the host processor executes the sequential code of a parallelized workload or unparallelized legacy applications. To address this issue, we propose Chameleon, low-cost architectural techniques that can virtualize the idle POD layer into a last-level cache memory, a prefetcher, or a hybrid between these two memory-enhancement techniques. By providing these additional functionalities with extremely low-cost hardware, we found that Chameleon can significantly improve the performance of memory-intensive sequential code in a very cost-effective manner.

Last but not least, without being satisfied with such a win in our POD architecture, we extended the idea of Chameleon to a more general architecture platform, a platform with integrated CPUs and GPUs, which is the most promising future heterogeneous many-core processor. Unlike the POD architecture, GPUs do not have a large cache memory or a scratch-pad memory, thus techniques proposed by Chameleon are not very useful for the integrated GPU. To address this issue, we proposed COMPASS, a compute-shader assisted prefetching scheme, in which we use the idle GPU into a programmable prefetcher by utilizing a large register file of a GPU, which is originally designed for heavy multi-threading for graphics or high-performance computation. From this study, we found several unique features of GPUs such as throughput-oriented pipeline may make prefetching less effective. However, we found that those problems can be mitigated by several architectural techniques such as deeper prefetching and using different indexing functions. Furthermore, we found that the programmability of COMPASS allows us to predict miss

addresses more accurately than a conventional hardware prefetchers. As a result, we found that such a software-hardware cooperative approach can utilize a GPU to improve the memory performance of an integrated CPU even providing the capability of application-specific prefetching.

REFERENCES

[1] Advanced Micro Devices Inc., "R700-Family Instruction Set Architecture," March 2009. http://developer.amd.com/gpu_assets/R700-Family_Instruction_Set_Architecture.pdf Date accessed: August 2009.

[2] Agarwal, M., Malik, K., Woley, K. M., Stone, S. S., and Frank, M. I., "Exploiting postdominance for speculative parallelization," in *Proceedings of the International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), pp. 295–305, IEEE Computer Society, 2007.

[3] AGEIA, "Physx product overview." http://www.ageia.com Date accessed: March 2008.

[4] Ahn, J. H., Dally, W. J., Khailany, B., Kapasi, U., and Das, A., "Evaluating the Imagine System Architecture," in *Proceedings of the International Symposium on Computer Architecture*, 2004.

[5] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," in *AFIPS Spring Joint Computer Conference*, 1967.

[6] Annavaram, M., Patel, J., and Davidson, E., "Data Prefetching by Dependence Graph Precomputation," in *Proceedings of the International Symposium on Computer Architecture*, 2001.

[7] Arevalo, A., Matinata, R. M., Pandian, M., Peri, E., Ruby, K., Thomas, F., and Almond, C., *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. IBM Publications Center, 2008.

[8] Artieri, A., "Nomadik: an MPSoC solution for advanced multimedia," in *Proceedings of the 5th International Forum on Application-Specific Multi-Processor SoC*, 2005.

[9] Bahl, L., Cocke, J., Jelinek, F., and Raviv, J., "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. 20, no. 2, pp. 284–287, 1974.

[10] Blank, T., "The MasPar MP-1 Architecture," in *Proceedings of COMPCON*, Spring 1990.

[11] Borkar, S., "Networks for Multi-core Chip–A Controversial View," in *2006 Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems*, 2006.

[12] Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P. S., Sutton, J., Urbanski, J., and Webb, J., "iWarp: An integrated solution to high-speed parallel computing," in *Supercomputing '88*, pp. 330–339, 1988.

[13] Bouknight, W. J., Denenberg, S. A., McIntyre, D. F., Randall, J. M., Sameh, A. H., and Slotnick, D. L., "The Illiac IV System," in *Proceedings of IEEE*, April 1972.

[14] Brooks, D., Tiwari, V., and Martonosi, M., "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the International Symposium on Computer Architecture*, pp. 83–94, 2000.

[15] Buck, I., "GPU computing with NVIDIA CUDA," in *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, (New York, NY, USA), p. 6, ACM, 2007.

[16] Callahan, D., Kennedy, K., and Porterfield, A., "Software Prefetching," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[17] Cao, Y., Sato, T., Orshansky, M., Sylvester, D., and Hu, C., "New paradigm of predictive MOSFET and interconnect modeling for early circuit simulation," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 201–204, 2000.

[18] Chang, J. and Sohi, G. S., "Cooperative caching for chip multiprocessors," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, (Washington, DC, USA), pp. 264–276, IEEE Computer Society, 2006.

[19] Chen, T.-F. and Baer, J.-L., "Reducing Memory Latency via Non-blocking and Prefetching Caches," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[20] Chen, T., Zhang, T., Sura, Z., and Tallada, M. G., "Prefetching irregular references for software cache on cell," in *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, (New York, NY, USA), pp. 155–164, ACM, 2008.

[21] Chen, W. Y., Mahlke, S. A., Chang, P. P., and Hwu, W.-m. W., "Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching," in *Proceedings of the International Symposium on Microarchitecture*, 1991.

[22] Cheriton, D. R., Slavenburg, G. A., and Boyle, P. D., "Software-controlled caches in the vmp multiprocessor," in *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, (Los Alamitos, CA, USA), pp. 366–374, IEEE Computer Society Press, 1986.

[23] Chishti, Z., Powell, M. D., and Vijaykumar, T. N., "Distance associativity for high-performance energy-efficient non-uniform cache architectures," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), p. 55, IEEE Computer Society, 2003.

[24] Cho, S. and Melhem, R., "Corollaries to amdahl's law for energy," *IEEE Compututer Architecture Letters*, vol. 7, no. 1, pp. 25–28, 2008.

[25] Collins, J., Wang, H., Tullsen, D., Hughes, C., Lee, Y., Lavery, D., and Shen, J., "Speculative Precomputation: Long-range Prefetching of Delinquent Loads," in *Proceedings of the International Symposium on Computer Architecture*, 2001.

[26] Cooksey, R., Jourdan, S., and Grunwald, D., "A Stateless, Content-Directed Data Prefetching Mechanism," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[27] Dally, W. J. and Towles, B., "Route Packets, Not Wires: On-Chip Interconnection Networks," in *Proceedings of the 38th Design Automation Conference*, 2001.

[28] Dimitrov, M. and Zhou, H., "Combining Local and Global History for High Performance Data Prefetching," in *The Journal of Instruction-Level Parallelism Data Prefetching Championship*, 2009.

[29] Duller, A., Panesar, G., and Towner, D., "Parallel processing-the picochip way," *Communicating Processing Architectures*, pp. 125–138, 2003.

[30] Dundas, J. and Mudge, T., "Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss," in *Proceedings of the International Conference on Supercomputing*, 1997.

[31] Dutta, S., Jensen, R., and Rieckmann, A., "Viper: A multiprocessor soc for advanced set-top box and digital tv systems," *IEEE Des. Test*, vol. 18, no. 5, pp. 21–31, 2001.

[32] Eichenberger, A. E., O'Brien, K., O'Brien, K., Wu, P., Chen, T., Oden, P. H., Prener, D. A., Shepherd, J. C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., and Gschwind, M., "Optimizing compiler for the cell processor," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 161–172, IEEE Computer Society, 2005.

[33] Espasa, R., Ardanaz, F., Emer, J., Felix, S., Gago, J., Gramunt, R., Hernandez, I., Juan, T., Lowney, G., Mattina, M., and A., S., "Tarantula: a vector extension to the alpha architecture," in *Proceedings of the International Symposium on Computer Architecture*, 2002.

[34] Fedorova, A., Seltzer, M., and Smith, M., "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[35] Fedorova, A., Seltzer, M., Small, C., and Nussbaum, D., "Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[36] Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *Proceedings of the International Symposium on Microarchitecture*, 2007.

[37] Ganusov, I. and Burtscher, M., "Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[38] Gebis, J., Williams, S., Patterson, D., and Kozyrakis, C., "VIRAM1: A Media-Oriented Vector Processor with Embedded DRAM," *41st DAC Student Design Contest*, 2004.

[39] Ghuloum, A., Smith, T., Wu, G., Zhou, X., Fang, J., Guo, P., So, B., Rajagopalan, M., Chen, Y., and Chen, B., "Future-Proof Data Parallel Algorithms and Software on Intel^{TM} Multi-Core Architecture," in *Intel Technology Journal*, Vol. 11, Issue 4 2007.

[40] Gonzalez, R. and Horowitz, M., "Energy dissipation in general purpose microprocessors," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 9, pp. 1277–1284, 1996.

[41] Grammatikakis, M. D., Hsu, D. F., Kraetzl, M., and Sibeyn, J. F., "Packet routing in fixed-connection networks: A survey," *Journal of Parallel and Distributed Computing*, vol. 54, no. 2, pp. 77–132, 1998.

[42] Guo, F., Solihin, Y., Zhao, L., and Iyer, R., "A framework for providing quality of service in chip multi-processors," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 343–355, IEEE Computer Society, 2007.

[43] Halfhill, T., "Massively Parallel Digital Video." Microprocessor Report, January 2006.

[44] Hammond, L., Nayfeh, B. A., and Olukotun, K., "A Single-Chip Multiprocessor," vol. 30, no. 9, pp. 79–85, 1997.

[45] Hammond, L., Willey, M., and Olukotun, K., "Data Speculation Support for a Chip Multiprocessor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[46] Hamzaoglu, F., Zhang, K., Wang, Y., Ahn, H., Bhattacharya, U., Chen, Z., Ng, Y., Pavlov, A., Smits, K., Bohr, M., and others, "A 153Mb-SRAM Design with Dynamic Stability Enhancement and Leakage Reduction in 45nm High-K Metal-Gate CMOS Technology," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pp. 376–621, 2008.

[47] Harris, S., *SYNERGISTIC CACHING IN SINGLE-CHIP MULTIPROCESSORS*. PhD thesis, stanford university, 2005.

[48] Hegde, R., " Optimizing Application Performance on Intel® Core™ Microarchitecture Using Hardware-Implemented Prefetchers." Intel Software Network http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers Date accessed: August 2009.

[49] Held, J., Bautista, J., and Koehl, S., "From a Few Cores to Many: A Tera-scale Computing Research Overview," *Intel Whitepaper*, 2006.

[50] Hensley, J., "AMD CTM overview," in *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, (New York, NY, USA), p. 7, ACM, 2007.

[51] Hill, M. and Marty, M., "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[52] Hofstee, H. P., "Power Efficient Processor Architecture and The Cell Processor," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2005.

[53] Hsu, L. R., Reinhardt, S. K., Iyer, R., and Makineni, S., "Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, (New York, NY, USA), pp. 13–22, ACM, 2006.

[54] Hu, Z., Kaxiras, S., and Martonosi, M., "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior," in *Proceedings of the International Symposium on Computer Architecture*, 2002.

[55] Huddy, R., "ATI Radeond™ HD 2000 SeriesTechnology Overview," in *AMD Technical Day, The Develop Conference & Expo*, 2007.

[56] Huh, J., Kim, C., Shafi, H., Zhang, L., Burger, D., and Keckler, S. W., "A nuca substrate for flexible cmp cache sharing," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, (New York, NY, USA), pp. 31–40, ACM, 2005.

[57] Hwu, W.-M., Ryoo, S., Ueng, S.-Z., Kelm, J. H., Gelado, I., Stone, S. S., Kidd, R. E., Baghsorkhi, S. S., Mahesri, A. A., Taso, S. C.,

Navarro, N., Lumetta, S. S., Frank, M. I., and Patel, S. J., "Implicitly Parallel Programming Models for Thousand-Core Microprocessors," 2007.

[58] Intel Corporation, "http://processorfinder.intel.com/details.aspx?sSpec5=SLANR Date accessed: March 2008."

[59] Intel Corporation, "Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7-900 Desktop Processor Series," October 2009.

[60] Joseph, D. and Grunwald, D., "Prefetching using markov predictors," in *Proceedings of the International Symposium on Computer Architecture*, (New York, NY, USA), pp. 252–263, ACM, 1997.

[61] Kadota, H., Miyake, J., Okabayashi, I., Maeda, T., Okamoto, T., Nakajima, M., and Kagawa, K., "A 32-bit CMOS microprocessor with on-chip cache and TLB," *IEEE Journal of Solid-State Circuits*, vol. 22, no. 5, pp. 800–807, 1987.

[62] Kandemir, M., Ramanujam, J., Irwin, M., Vijaykrishnan, N., Kadayif, I., and Parikh, A., "Dynamic management of scratch-pad memory space," in *Design Automation Conference, 2001. Proceedings*, pp. 690–695, 2001.

[63] Kandiraju, G. B. and Sivasubramaniam, A., "Going the Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the International Symposium on Computer Architecture*, 2002.

[64] Kapasi, U. J., Dally, W. J., Rixner, S., Owens, J. D., and Khailany, B., "The Imagine Stream Processor," in *Proceedings of the International Conference on Computer Design*, 2002.

[65] Kim, C., Burger, D., and Keckler, S., "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 211–222, 2002.

[66] Kim, J. S., Taylor, M. B., Miller, J., and Wentzlaff, D., "Energy Characterization of a Tiled Architecture Processor with On-Chip Networks.," in *Proceedings of the 8th International Symposium on Low Power Electronics and Design*, 2003.

[67] Kim, S., Chandra, D., and Solihin, Y., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," pp. 111–122, Sept. 2004.

[68] Knauerhase, R., Brett, P., Hohlt, B., Li, T., and Hahn, S., "Using OS Observations to Improve Performance in Multicore Systems," *IEEE Micro*, vol. 28, no. 3, pp. 54–66, 2008.

[69] Kumar, M., Baransky, Y., and Denneau, M., "The GF11 Parallel Computer," *Parallel Computing*, vol. 19, no. 12, pp. 1393–1412, 1993.

[70] Kung, H. T., "Why Systolic Architectures," *IEEE Computer*, vol. 15, no. 1, pp. 37–46, 1982.

[71] Lee, C., Potkonjak, M., and Mangione-Smith, W. H., "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *Proceedings of the International Symposium on Microarchitecture*, 1997.

[72] Leighton, F. T., *Introduction to parallel algorithms and architectures : arrays, trees, hypercubes*. Morgan Kaufmann, 1992.

[73] Liao, S. S., Wang, P. H., Wang, H., Hoflehner, G., Lavery, D., and Shen, J. P., "Post-Pass Binary Adaptation for Software-Based Speculative Precomputation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[74] Lin, W., Reinhardt, S., and Burger, D., "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2001.

[75] Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., and Torrellas, J., "Posh: a tls compiler that exploits program structure," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 158–167, ACM, 2006.

[76] Lucas, B. and Kanade, T., "An iterative image registration technique with an application to stereo vision," in *International joint conference on artificial intelligence*, vol. 3, pp. 674–679, 1981.

[77] Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., and Lefohn, A., "GPGPU: general purpose computation on graphics hardware," in *Proceedings of the conference on SIGGRAPH 2004 course notes*, ACM Press New York, NY, USA, 2004.

[78] Luk, C.-K., "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," in *Proceedings of the International Symposium on Computer Architecture*, 2001.

[79] Madan, N. and Balasubramonian, R., "Leveraging 3D Technology for Improved Reliability," in *Proceedings of the International Symposium on Microarchitecture*, 2007.

[80] Mantor, M., "Radeon R600, a 2nd Generation Unified Shader Architecture," in *Proceedings of the 19th Hot Chips Conference, August*, 2007.

[81] Mantor, M., "Entering the Golden Age of Heterogeneous Computing," in *Performance Enhancement on Emerging Parallel Processing Platforms*, 2008.

[82] MasPar, "Maspar programming language (ansi c compatible mpl) reference manual."

[83] McCool, M. D., Wadleigh, K., Henderson, B., and Lin, H.-Y., "Performance evaluation of gpus using the rapidmind development platform," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 181, ACM, 2006.

[84] Miller, J. E. and Agarwal, A., "Software-based instruction caching for embedded processors," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 293–302, ACM, 2006.

[85] Moore, C., "The Role of Accelerated Computing in the Multi-core Era," in *Workshop on Manycore and Multicore Computing: Architectures, Applications And Directions*, 2007.

[86] Moritz, C., Frank, M., Lee, W., and Amarasinghe, S., "Hot Pages: Software Caching for Raw Microprocessors," 1999.

[87] Mudge, T., "Power: a First-class Architectural Design Constraint," *IEEE Computer*, vol. 34, pp. 52–58, 2001.

[88] Munshi, A., "Opencl: Parallel computing on the gpu and cpu," in *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, 2008.

[89] Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2003.

[90] Mysore, S., Agrawal, B., Srivastava, N., Lin, S., Banerjee, K., and Sherwood, T., "Introspective 3D chips," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 264–273, ACM Press New York, NY, USA, 2006.

[91] Narayanan, R., Ozisikyilmaz, B., Zambreno, J., Pisharath, J., Memik, G., and Choudhary, A., "MineBench: A Benchmark Suite for Data Mining Workloads," in *Proceedings of the International Symposium on Workload Characterization*, 2006.

[92] Nesbit, K. and Smith, J., "Data Cache Prefetching Using a Global History Buffer," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2004.

[93] Nickolls, J. R., "The Design of the MasPar MP-1: a Cost Effective Massively Parallel Computer," in *Compcon Spring 90'*, 1990.

[94] Nikolic-Popovic, J., "Implementing a MAP Decoder for cdma2000^{TM} Turbo Codes on a TMS320C62x^{TM} DSP Device." Texas Instruments Application Report, May 2000.

[95] NVIDIA, "Tesla - GPU Computing Solution for HPC." http://www.nvidia.com/object/tesla_computing_solutions.html Date accessed: March 2008.

[96] Papakipos, M., "PeakStream Platform," 2006. SUPERCOMPUTING 2006 Tutorial on GPGPU, Course Notes.

[97] Patti, R., "3D-ICs: The Evolution and Direction of a New Technology, *IEEE International 3D Systems Integration Conference*." 2009.

[98] Peh, L.-S., "Chip-scale networks: Power and thermal impact." http://www.princeton.edu/ peh/talks/stanford_nws.pdf Date accessed: March 2008.

[99] Perez, D. G., Mouchard, G., and Temam, O., "Microlib: A case for the quantitative comparison of micro-architecture mechanisms," in *Proceedings of the International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 43–54, IEEE Computer Society, 2004.

[100] Pham, D., Asano, S., Bolliger, M., Day, M. N., Hofstee, H. P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., and Yazawa, K., "The Design and Implementation of a First-Generation CELL Processor," in *Proceedings of the 2005 IEEE International Solid-State Circuits Conference*, 2005.

[101] Pollack, F. J., "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (Keynote address)," 1999.

[102] Qureshi, M. K. and Patt, Y. N., "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 423–432, IEEE Computer Society, 2006.

[103] Rafique, N., Lim, W.-T., and Thottethodi, M., "Architectural Support for Operating System-Driven CMP Cache Management," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[104] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C., "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2007.

[105] Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., and Montesinos, P., "SESC simulator," January 2005. http://sesc.sourceforge.net Date accessed: August 2009.

[106] Rixner, S., Dally, W. J., Kapasi, U., Khailany, B., Lopez-Lagunas, A., Mattson, P., and Owens, J. D., "A Bandwidth-Efficient Architecture for Media Processing," in *Proceedings of the International Symposium on Microarchitecture*, 1998.

[107] Rubin, N., "Issues And Challenges In Compiling for Graphics Processors (Keynote speech)," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2008.

[108] Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S. W., and Moore, C. R., "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *Proceedings of the International Symposium on Computer Architecture*, 2003.

[109] Sharif, A. and Lee, H.-H. S., "Data Prefetching Mechanism by Exploiting Global and Local Access Patterns," in *The Journal of Instruction-Level Parallelism Data Prefetching Championship*, 2009.

[110] Siegel, H. J., Schwederski, T., Nathaniel J. Davis, I., and Kuehn, J. T., "Pasm: a reconfigurable parallel system for image processing," *SIGARCH Computer Architecture News*, vol. 12, no. 4, pp. 7–19, 1984.

[111] Slotnick, D. L., Borck, W. C., and McReynolds, R. C., "The Solomon Computer," in *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 22, pp. 97–107, 1962.

[112] Smith, S. L., "Intel Roadmap Overview," in *Intel Developer Forum*, 2008.

[113] Sohi, G. S., Breach, S. E., and Vijaykumar, T. N., "Multiscalar Processors," in *Proceedings of the International Symposium on Computer Architecture*, 1995.

[114] Solihin, Y., Lee, J., and Torrellas, J., "Using a User-Level Memory Thread for Correlation Prefetching," in *Proceedings of the International Symposium on Computer Architecture*, 2002.

[115] Taveniku, M., Ahlander, A., Jonsson, M., and Svensson, B., "The VEGA Moderately Parallel MIMD, Moderately Parallel SIMD, Architecture for High Performance Array Signal Processing," in *International Parallel Processing Symposium*, 1998.

[116] Taylor, M., Amarasinghe, S., and Agarwal, A., "Scalar Operand Networks," *IEEE Transactions on Parallel and Distributed Systems*, 2005.

[117] Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., and Agarwal, A., "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs," *IEEE Micro*, Mar/Apr, 2002.

[118] Tendler, J., Dodson, S., Fields, S., Le, H., and Sinharoy, B., "POWER4 System Microarchitecture." IBM Technical White Paper, October 2001.

[119] Thoziyoor, S., Muralimanohar, N., Ahn, J., and Jouppi, N., "CACTI 5.1," tech. rep., HP Laboratories, Palo Alto, 2008.

[120] Tilera Corporation, "TILE64 Processor Family." http://www.tilera.com/products/processors.php Date accessed: March 2008.

[121] Tuck, N. and Tullsen, D., "Initial Observations of the Simultaneous Multithreading Pentium 4 Processor," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[122] Tucker, L. W. and Robertson, G. G., "Architecture and Applications of the Connection Machine," *IEEE Computer*, August 1988.

[123] Udayakumaran, S., Dominguez, A., and Barua, R., "Dynamic allocation for scratch-pad memory using compile-time decisions," *Trans. on Embedded Computing Sys.*, vol. 5, no. 2, pp. 472–511, 2006.

[124] Wang, H., Peh, L.-S., and Malik, S., "Power-driven Design of Router Microarchitectures in On-Chip Networks," in *Proceedings of the International Symposium on Microarchitecture*, 2003.

[125] Woo, D. H., Fryman, J. B., Knies, A. D., Eng, M., and Lee, H.-H. S., "POD: A 3D-Integrated Broad-Purpose Acceleration Layer," *IEEE Micro*, vol. 28, no. 4, pp. 28–40, 2008.

[126] Woo, D. H., Fryman, J. B., Knies, A. D., and Lee, H.-H. S., "Chameleon: Virtualizing Idle Acceleration Cores of A Heterogeneous Multi-Core Processor for Caching and Prefetching," *ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 1, 2010.

[127] Woo, D. H. and Lee, H.-H. S., "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era," *Computer*, vol. 41, no. 12, pp. 24–31, 2008.

[128] Woo, D. H. and Lee, H.-H. S., "COMPASS: a programmable data prefetcher using idle GPU shaders," in *Proceedings of the International Conference on*

Architectural Support for Programming Languages and Operating Systems, pp. 297–310, 2010.

[129] Yeh, T. Y., Faloutsos, P., Patel, S. J., and Reinman, G., "ParallAX: An Architecture for Real-Time Physics," in *Proceedings of the International Symposium on Computer Architecture*, 2007.

[130] Zhang, M. and Asanovic, K., "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Proceedings of the International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 336–345, IEEE Computer Society, 2005.