Chameleon: Virtualizing Idle Acceleration Cores of A Heterogeneous Multi-Core Processor for Caching and Prefetching

Dong Hyuk Woo[1] Joshua B. Fryman[2] Allan D. Knies[3] Hsien-Hsin S. Lee[1]

[1]School of Electrical and Computer Engineering
Georgia Institute of Technology
{dhwoo, leehs}@ece.gatech.edu

[2]Microprocessor Technology Labs
Intel Corporation
joshua.b.fryman@intel.com

[3]Intel Research Berkeley
allan.knies@intel.com

Technical Report GIT-CERCS-08-11

December 2008

Abstract

Heterogeneous multi-core processors have emerged as an energy- and area-efficient architectural solution to improving performance for domain-specific applications such as those with a plethora of data-level parallelism. These processors typically contain a large number of small, compute-centric cores for acceleration while keeping one or two high-performance ILP cores on the die to guarantee single-thread performance. Although a major portion of the transistors are occupied by the acceleration cores, these resources will sit idle when running unparallelized legacy codes or the sequential parts of an application. To address this under-utilization issue, in this paper, we introduce Chameleon, a flexible heterogeneous multi-core architecture to virtualize these resources for enhancing memory performance when running sequential programs. The Chameleon architecture can dynamically virtualize the idle acceleration cores into a last-level cache, a data prefetcher, or a hybrid between these two techniques. In addition, Chameleon can operate in an adaptive mode which dynamically configures the acceleration cores between the hybrid mode and the prefetch-only mode by monitoring the effectiveness of Chameleon caching scheme. In our evaluation using SPEC2006 benchmark suite, different levels of performance improvements were achieved in different modes for different applications. In the case of the adaptive mode, Chameleon improves the performance of SPECint06 and SPECfp06 by 33% and 22% on average. When considering only memory-intensive applications, Chameleon improves the system performance by 53% and 33%.

1 Introduction

Heterogeneous computing has emerged to address the growing concerns of energy efficiency and silicon area effectiveness. Small-scale heterogeneous MPSoCs have been used in embedded systems for years [13, 3]. Meanwhile, general-purpose processor designers are also advocating such heterogeneous architectures for future multi-core 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) [42] integrated eight synergistic processing elements along with a superscalar PowerPC processor on the same die. CUDA [6],

CTM [20], RapidMind [33], OpenCL [37], PeakStream [39], and Ct [15] serve as programming abstractions to enable heterogeneous computing based on a cooperative computing model between the CPU(s) 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 [35].

Unlike a symmetric multi-core processor where all processor cores are identical, a heterogeneous computing platform is composed of distinct classes of processing cores. One example is to have a state-of-the-art processor as the host integrated with an array of parallel processing elements (PEs) for accelerating certain parts of an application [42, 35, 48, 50, 47, 51, 32]. The high-performance host processor is mainly used for exploiting the sequential performance of an application. In contrast, the acceleration PE cores, typically not designed with complex ILP techniques, are integrated to deliver high throughput for parallelizable code with better energy- and area-efficiency. The rationale behind such a heterogeneous multi-core design is (1) to improve the energy efficiency and manage its ensuing thermal issues when running data-parallel workloads; and (2) to not lose sequential performance.

Unfortunately, while the host processor executes the sequential code of a parallelized workload or unparallelized legacy applications, the acceleration cores of a heterogeneous multi-core 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 paper, we introduce *Chameleon*, a flexible architecture with low-cost enabling techniques to provide several dynamic operation modes for better resource allocation. In addition to the parallel acceleration mode, the PE cores in Chameleon can be configured into a last-level cache, and a data prefetcher mode, and variants of their combinations when running sequential programs. The main contributions of this work are:

- We propose the Chameleon heterogeneous multi-core architecture which virtualizes otherwise unused acceleration cores to enable prefetching and additional cache space when running sequential workloads. We also show that the extra hardware cost to realize Chameleon is not significant.
- We propose several different operation modes including a caching mode, a data prefetching mode, and a hybrid mode that virtualize the acceleration cores collectively for enhancing memory performance. In addition, an adaptive mode is introduced to change these modes dynamically to adapt the memory behavior of a running application.
- We perform a case study using a heterogeneous multicore consisting of a host processor integrated with an on-die massively parallel SIMD accelerator. We justify the performance benefits of our Chameleon architecture and present their hardware/power overheads and energy implications in our evaluation.

This paper is organized as follows: Section 2 explains the architectural features of our baseline heterogeneous multi-core. Section 3 enumerates different design issues to provide a virtualized last-level cache and a virtualized prefetcher. It also describes the hybrid and the adaptive designs. Section 4 evaluates the characteristics of these different design choices and demonstrates single-thread performance improvement of SPEC 2006 benchmark suite. Section 5 discusses other issues regarding Chameleon's implementation, and Section 6 enumerates related work. Finally, Section 7 concludes.

2 Baseline Multi-Core Architecture

In this section, we briefly discuss our baseline heterogeneous multi-core architecture used throughout this paper before we introduce our Chameleon architecture. Note that, we use this baseline as a case study to demonstrate our techniques. The proposed ideas for Chameleon, however, are not limited to our baseline choice although they may require different design hooks. The main idea of Chameleon can be extended and applied to other heterogeneous multi-core processors such as the IBM Cell/BE [42], future integrated GPU [30] / Larrabee cores [45], or other similar acceleration-based multi-core architectures [51, 32]

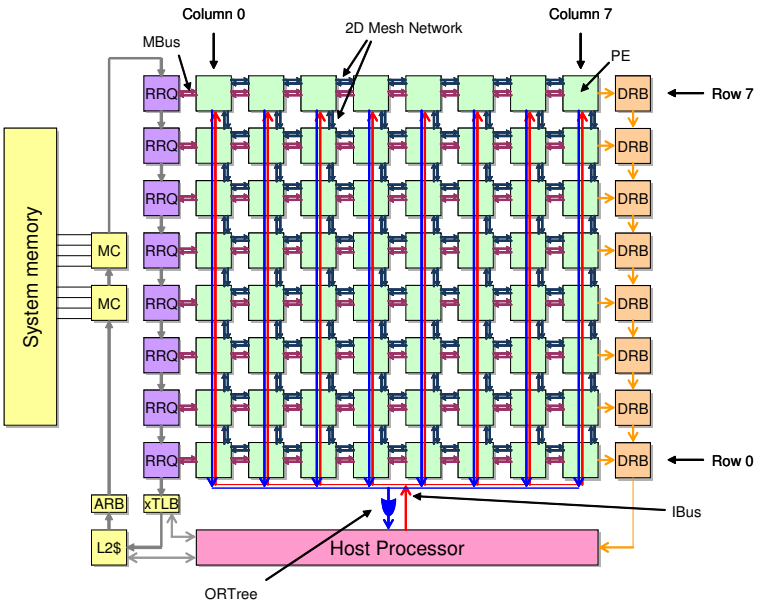


Figure 1: Baseline Heterogeneous Multi-core Architecture

Figure 1 illustrates our baseline heterogeneous multi-core processor which resembles the recently proposed Parallel-On-Demand architecture [50], MorphoSys [47], or early MPPs [4, 38] but on a single die. It consists of a state-of-the-art superscalar host processor and an array of 8 $\times$ 8 PE cores. The size of the PE array is not fixed, and will change for different process technology and different market requirements. The host processor is responsible for the master control of the entire computation. In addition to executing the sequential part of an application, the host processor also dispatches and orchestrates the instructions executed on the PE array to enable high throughput data parallel processing. Each PE in the array is a three-wide VLIW machine with a fixed instruction format composed of three pipelined operations: a *G* (Generic), an *X* (SSE), and an *M* (Memory) operation. To manage local data during the parallel execution phase, a 128KB scratch-pad memory is provided for each individual PE. To support *if-then-else* statements, masking instructions are also included to enable or disable each PE individually depending on the flag status.

To execute instructions in the PE array, the host processor needs to broadcast three-wide 96-bit VLIW instructions to the PEs via an instruction bus (IBus) shown in Figure 1. On the other hand, the host processor can monitor and obtain the status of PEs through an ORTree (128-bit wide) which logically combines the outcomes of PEs' flag registers (e.g., 64-bit RFLAGS and 64-bit MXCSR (MMX/SSE Control/Status Register)). The host processor can also read scalar values from PEs through a data return buffer (DRB) located on the rightmost column of the PE array. Furthermore, a PE can write its computation results back to the virtual memory space and the host processor can read them through the regular memory hierarchy.

PEs are connected to a DMA engine called row response queue (RRQ). An RRQ and all PEs in the same row are connected through a memory bus (MBus) consisting of two uni-directional buses. One bus streams data back from the main memory to the PEs in the row, and the other bus streams data from the PEs in the row to the main memory. PEs can also communicate with each other through a mesh network[1]. Communication is fully software-controlled by a communication instruction, which allows each PE to transfer 64-bit or 128-bit data to one of its four neighbor PEs. Because all communication is fully controlled by explicit instructions and all execution is fully orchestrated by the host processor, any communication pattern is completely deterministic without issues caused by bus arbitration, congestion, deadlock, and live-lock. Furthermore, each PE only needs a single 4:1 mux and 1:4 demux for its mesh communication. Therefore, no area- and power-hungry router is required.

[1]The mesh network can be reconfigured to a folded torus network using simple switches [12, 46]

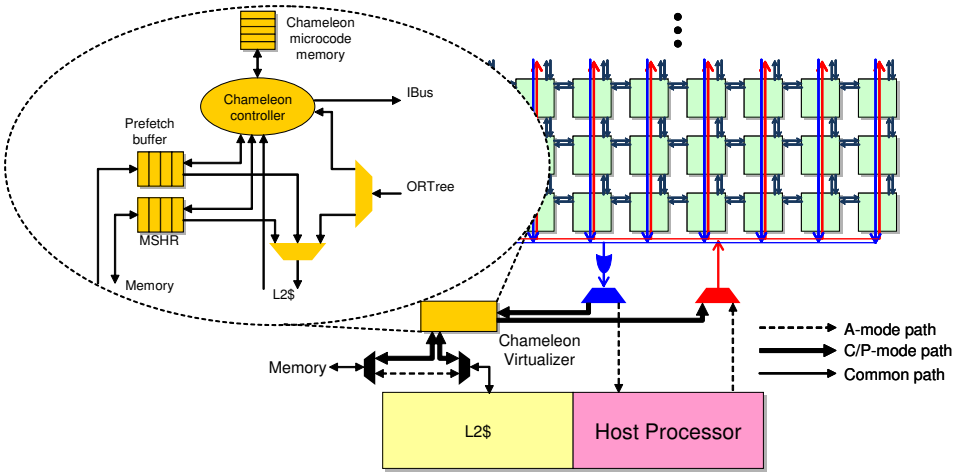


Figure 2: Chameleon Virtualizer (not scaled)

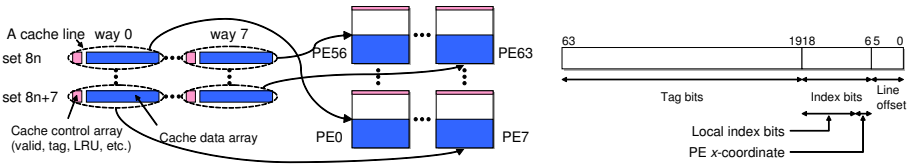
3 Chameleon Architecture

Here we describe our proposed architecture called *Chameleon* for future heterogeneous multi-core processors. To achieve better utilization of on-die resources, we added low-cost configuration hardware that virtualizes idle acceleration cores dynamically to improve sequential performance. 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 based on the dynamic behavior of an application.

3.1 C-Mode: Virtualizing Idle Cores for Caching

As shown in Section 2, the original purpose of integrating the heterogeneous PE array onto general-purpose processor cores is to accelerate the data-parallel part of an application to obtain better 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. Note that this virtualization must be simple, low-overhead, and not affect the efficiency when the A-mode is turned on. The idea is to configure the unused local scratch-pad memories collectively into a last-level cache while 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 IBus, performing corresponding local computation, and routing the results back. For example, to calculate the cache index bits, the PE is instructed to perform an SHR (logical shift-right) to eliminate the cache line offset and an AND (logical and) to mask out non-index bits. Using this calculated index bits, the PE can read data from its local 128KB scratch-pad memory with a load instruction.

To achieve this, a new interface between the L2 cache of the host processor and the baseline PE array is required in order to control the PE array and have it function like a soft cache. As shown in Figure 2, this new interface, called *Chameleon Virtualizer*, is in charge of orchestrating the memory management operations needed for the virtualized last-level cache using microcode stored inside the Chameleon microcode memory. The microcode is written in the original PE ISA and consists of 10s of PE instructions. Upon a cache read miss in the L2 cache, for example, the miss address is forwarded to the Chameleon controller. Upon 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, the following tasks are performed by PEs: (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,



(a) Layout (b) Indexing

Figure 3: Way-Level Parallelism (8-Way Cache, 4MB)

and (6) updating the 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 3.2. Note that the overhead of the Chameleon Virtualizer is only incurred in the C-mode as 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 ORTree bus which was originally designed for obtaining the flag status of the PE array, but is idle when operating in the C-mode. Hence, we hijack this bus to send hit/miss signals and transfer requested cache lines. However, we need to introduce a new instruction called xferortree to 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.

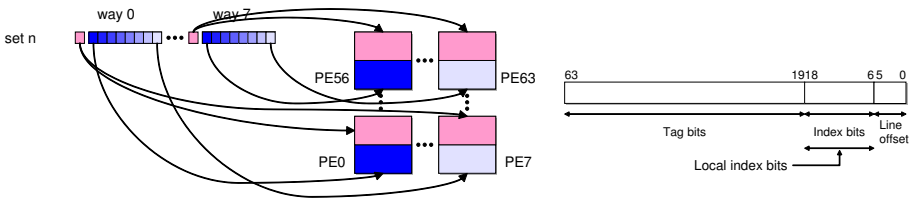
In the following sections, we will address the challenges with respect to the styles of cache line layout across the PE array. We also detail these design alternatives and evaluate and quantify their trade-offs in our experiments. Furthermore, we investigate how can we optimize their access latency by adopting NUCA (Non-Uniform Cache Architecture) models and discuss the required architectural support.

3.1.1 Design for Way-Level Parallelism

Our first design is to distribute multi-way cache lines of the same set across the PEs in the same column. Figure 3 shows an example of an eight-way set-associative 4MB cache. In this example, eight cache lines mapped to the same set are distributed over eight PEs in the same column (e.g., PE0, PE8, to PE56). Also shown in the figure is how to index the cache. Out of the global index bits, three LSBs are used to generate the x-coordinate of the PE location on the same row. The rest of the index bits (10 bits) are used as the local index to look up the cache line in the eight local scratch-pad memories on the indexed column. Each PE in this example can store up to 1,024 cache lines. 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 perform their tag matching 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, we chose to implement the *counter LRU algorithm* [25] and have the replacement performed by microcode stored inside the Chameleon microcode memory. 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 takes longer than a hardware-based LRU update, we found that properly scheduled microcode can hide much of this latency.

In terms of area utilization, this WLP design has a space overhead for cache control bits. To implement an eight-way



(a) Layout (b) Indexing

Figure 4: Way- and Subblock-Level Parallelism (8-Way Cache, 4MB)

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 per 64B 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 to make cache indexing practical. As a result, in our WLP design example, each PE will accommodate 1,024 cache lines.

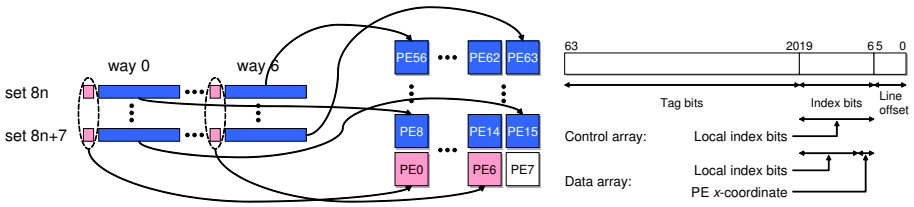
In this design, once the set is determined, only one corresponding column is enabled to complete one cache operation. In other words, if the Chameleon Virtualizer can provide eight instruction streams, and decode eight returning messages, we can build a virtualized eight-bank cache. To implement it,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

3.1.2 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 4 and 8 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) need to be used to load each 16B chunk into the XMM registers (or store 8B chunk to general purpose registers), adding 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 3. To alleviate this issue, we investigate another design option where a 64B cache line is split across eight PEs on the same row as shown in Figure 4. 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 instruction 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. To implement it, 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.

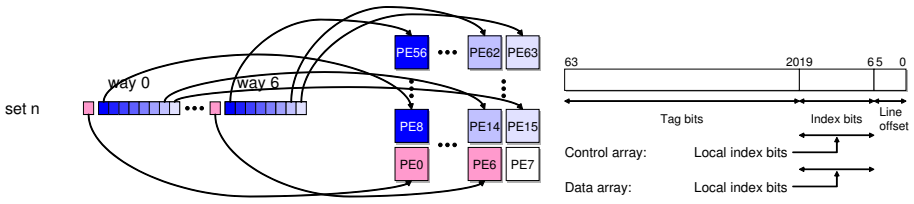
The primary challenge of such a WBLP design is the area overhead in keeping the cache control bits. As a cache line is split into eight subblocks, all eight PEs that keep a subblock of the same cache line need to have redundant valid, tag, LRU and coherence bits. Otherwise, more delay will incur for communicating this information. We found that each PE can accommodate this redundant information without sacrificing the overall cache capacity. As explained in Section 3.1.2, 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. 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

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



(a) Layout (b) Indexing

Figure 5: Decoupled WLP Cache (7-Way Cache, 7MB)



(a) Layout (b) Indexing

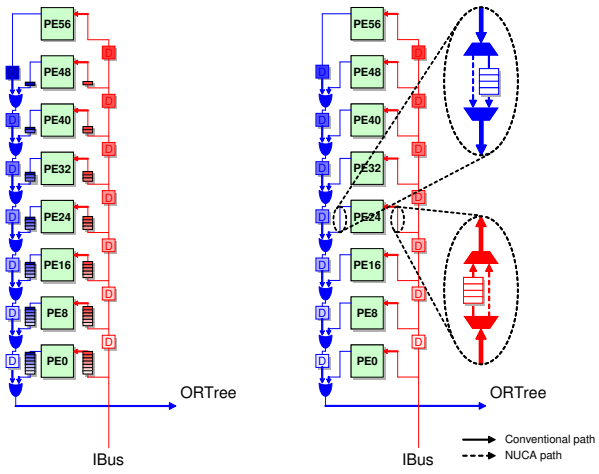
Figure 6: Decoupled WBLP Cache (7-Way Cache, 7MB)

capacity.

3.1.3 Decoupled Design

The two designs discussed previously place the cache control array and data array in the same PE, so that each PE can locally detect whether the request is a hit or 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.

Instead, we study an alternative design style where 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 storing the cache control array, and then it needs to request the target PE storing the hit line to route the line back to the Chameleon Virtualizer. Figure 5 and Figure 6 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 cache control array for the data array of PEs in row $n + 1$. For example, in case of a decoupled WBLP cache (Figure 6), 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 the 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 stores the cache data array of 2k lines (the decoupled WLP cache (Figure 5)) or the 8B subblock of 16k lines (the decoupled WBLP cache (Figure 6)). Note that these 63 PEs fully utilize their 128KB local scratchpad memory. The only unused space is the local memory space of PE7 as shown in the figures.



(a) Conventional, Synchronized Model

(b) Pipelined Model to Support NUCA

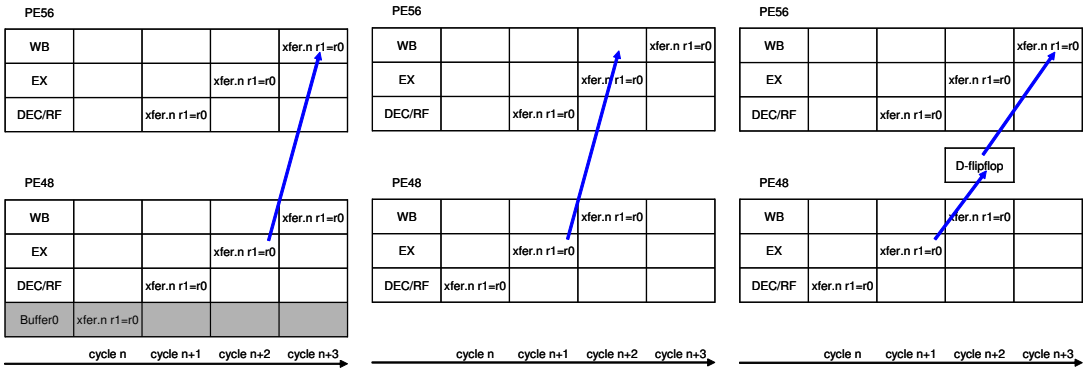
Figure 7: Execution Model (Only Column 0 is shown.)

3.1.4 NUCA Cache Design

In the conventional A-mode, to synchronize the computation for each PE, a PE located at row i (where i ranges from 0 to 7) in an 8x8 PE array will contain an instruction queue with $7 - i$ entries as shown in Figure 7(a). Instructions are broadcast through IBus and are 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. Sucn 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 research [27, 22, 11], a non-uniform cache architecture (NUCA) helps reduce the average cache access latency, thereby improving the overall performance. As such, it will be more desirable to keep high temporal-locality data in a nearby memory bank of a large NUCA structure. Although our baseline PE array already has a partitioned array of 64 PEs using mesh topology, 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 7(b), the NUCA path does not buffer any incoming cache access 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 the values using 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 8(a)), the r0 value of PE48 reaches PE56 at cycle $n + 3$ when the xfer.n instruction being executed

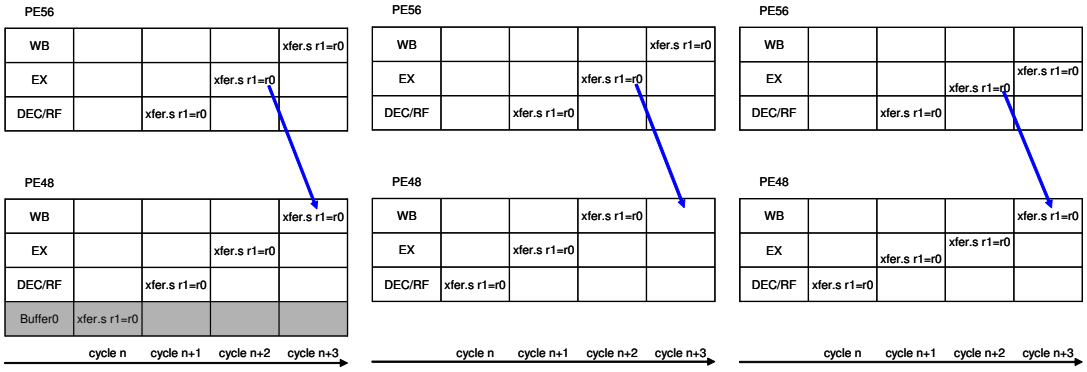


(a) Conventional, Strictly Synchronized Execution Model

(b) Pipelined Execution Model: Time-Zone Effect of xfer.n

(c) Pipelined Execution Model: Solution for Time-Zone Effect of xfer.n

Figure 8: Conventional and NUCA Execution Model of xfer.n Instruction



(a) Conventional, Strictly Synchronized Execution Model

(b) Pipelined Execution Model: Time-Zone Effect of xfer.s

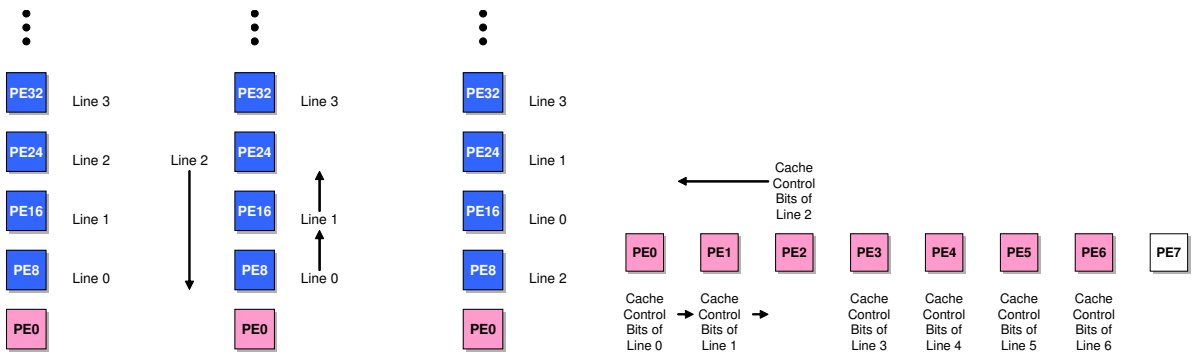
(c) Pipelined Execution Model: Solution for Time-Zone Effect of xfer.s

Figure 9: Conventional and NUCA Execution Model of xfer.s Instruction

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 8(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 8(c)).

A similar problem is present in synchronizing the southbound transfer xfer.s instruction. Figure 9(a) shows the synchronized execution model. However, in the pipelined execution model (Figure 9(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, the r1 of PE48 will not be correctly updated. Fortunately, as shown in Figure 9(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 how to make the LRU management efficient. Figure 10 illustrates an instance for



(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 10: LRU Management of a NUCA C-Mode (Decoupled WLP cache)

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 10(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 10(b)). This movement allows PEs to maintain more recently used cache lines closer to the Chameleon Virtualizer as shown in Figure 10(c).

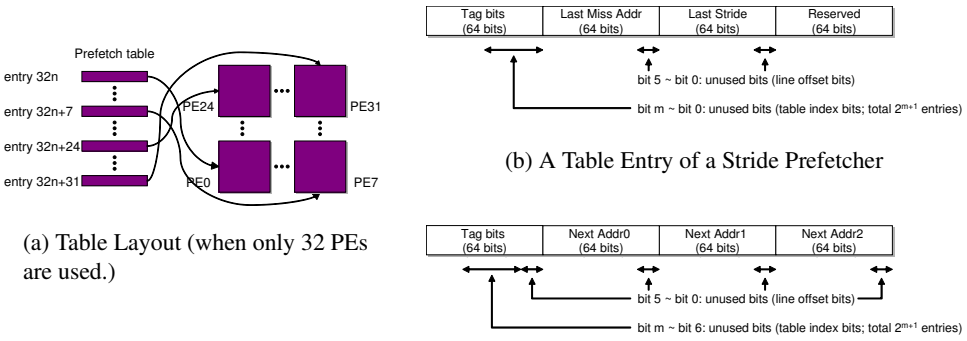
Additionally, we added one instruction called sampleortree to allow the PEs in row 1 to access the ORTree bus for obtaining the hit line directly when the line is migrated down to the MRU position (row 1). This is another special move instruction that drives ORTree data into a register. Otherwise, the 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.

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 10(d), we can force the y-coordinate of the PE caching the target data line to be always bigger by one than the x-coordinate of the PE caching the target control bits. For example, if the x-coordinate of the PE that has the control bits of the requested cache line is 3, the Chameleon Virtualizer can find corresponding data line in row 4.

3.2 P-Mode: Virtualizing Idle Cores as a Prefetcher

In addition to the C-mode that supplies a virtualized last-level cache, we also investigate the 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 multi-core 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).

The P-mode is particularly useful when the application running on the host processor demonstrates streaming behavior or contains a working set much larger than the extra cache capacity that C-mode can accommodate. In this work, we evaluate two types of data prefetchers: a PC-indexed stride prefetcher and a Markov prefetcher. We chose these



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

(b) A Table Entry of a Stride Prefetcher

(c) A Table Entry of a Markov Prefetcher

Figure 11: P-Mode Prefetcher

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 the simpler next-line prefetcher [19]. We will demonstrate that Chameleon can realize such area-consuming schemes by virtualizing the resources in P-mode.

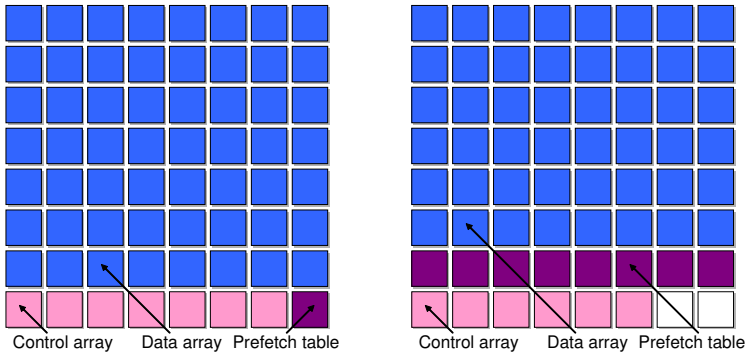
In the P-mode, the prefetch table is virtually laid out across PEs. Upon an L2 cache miss, the Chameleon Virtualizer checks its prefetch buffer first (Figure 2). If the requested line is not found, then the Chameleon Virtualizer broadcasts microcode to look up the virtualized prefetch table. This microcode 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 (e.g., the last miss address or the last stride in a stride prefetcher) and generates prefetch requests. To support P-mode, we added a small data prefetch buffer (a 32-entry buffer in this paper) in the Chameleon Virtualizer as shown in Figure 2. A prefetched cache line is temporarily stored in this buffer, which is checked upon every L2 cache miss.

3.2.1 Virtualized Stride Prefetcher

Figure 11(b) shows one entry of the prefetch table of a virtualized strider prefetcher. The size of each entry is 32B: 8B for tag bits, 8B for the last miss address, 8B for the last stride, and 8B unused. Clearly, this layout is not perfect in terms of the number of bits due to those unused bits in the table index bits, line offset bits, etc. However, the overall table size will not be larger even if we compact the table entry, simply because the number of entries per PE has to be a power-of-two to make PE indexing simple. A compact design will require unaligned memory accesses, introducing overhead at runtime. In our proposed design, 4,096 entries can be stored in each PE's local scratch-pad memory space.

The P-mode stride prefetcher is indexed by taking several LSBs (e.g., 17 bits on 32 PEs) of the PC. These index bits consists of PE ID bits (e.g., 5 LSBs on 32 PEs) and local index bits (e.g., 12 MSBs on 32 PEs). The PE ID bits are used to match 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 11(a). 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.

Upon an L2 cache miss, the Chameleon Virtualizer broadcasts the instruction's PC address 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 last miss address and the last stride stored in this table entry. In this paper, the P-mode stride prefetcher prefetches the four next addresses, that is, the prefetch depth is four. We also evaluated a stride prefetcher with a depth of eight in our experiments, but found that it only improves one application, 462.libquantum. Therefore, we do not show these results due to space limit.



(a) 7-Way 7MB NUCA WBLP Cache + 128KB Stride Prefetcher

(b) 6-Way 6MB NUCA WBLP Cache + 1MB Stride Prefetcher

Figure 12: Hybrid Design (Cache + Prefetcher)

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, for example in the case of a PC-indexed prefetcher, it would be better off to enable only eight PEs in the first row to reduce the lookup latency. In this paper, we vary the size of the prefetch table (1, 2, 4, and 8 rows) and perform a sensitivity study in our result section.

3.2.2 Virtualized Markov Prefetcher

Figure 11(c) shows the prefetch table design for a virtualized Markov prefetcher. Again, the size of the prefetch table entry is 32B, but containing the 8B current-miss address in the tag and three 8B next-miss addresses. The original Markov prefetcher [24] showed that a prefetch table with four next-miss addresses provides a reasonable balance between coverage and accuracy. To compromise with the 32B alignment issue, we evaluate a Markov prefetcher with three (instead of four) next-miss addresses. Similarly, although there exists some redundant information in this table as shown in Figure 11(c), we aligned each field at an 8B boundary, to make lookup microcode efficient.

Unlike a stride prefetcher, this table is indexed by a miss address of the cache line missing in the L2. 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.

To make the prefetch depth larger, the Chameleon Virtualizer needs to perform several lookups. For instance, to implement a prefetcher whose prefetch depth is two, the Chameleon Virtualizer needs to execute three more times (once per each of three next miss addresses). This iterative process consumes a large amount of the Chameleon Virtualizer bandwidth. Therefore, deep prefetching may degrade the overall performance. We evaluated a Markov prefetcher with a depth of two, but found its impact similar to that of depth one. Thus, we did not show them in Section 4 for brevity.

3.3 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 12(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 conventional NUCA WBLP cache. In this example, upon an L2 cache miss, the Chameleon Virtualizer

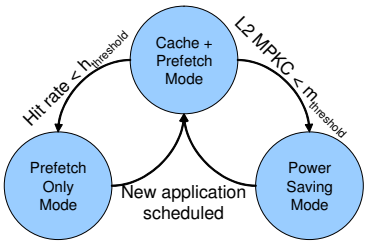


Figure 13: AdaptiveCP-Mode (MPKC: Misses Per Kilo Cycles)

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 12(b) shows another example of the HybridCP-mode, where a 6-way 6MB cache is co-located with a 1MB stride prefetcher table. In this example, the virtualized last-level cache is shrinked, but the prefetch table is enlarged. Note that arbitrary partioning between them, such as a 4MB cache and a 3MB prefetch table, may not be practical to not make PE indexing complicated.

3.4 AdaptiveCP-Mode: Mode Adaptation in Chameleon

Although there exists a wide spectrum of Chameleon design space, it is unlikely that any one 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 is running an application with good locality and a reasonable size of working set. However, when a streaming application is running, additional lookup latency of a cache will make a HybridCP-mode to perform worse than the one with the P-mode. Furthermore, some applications are purely computation-intensive, thus Chameleon will not help to improve the overall performance but may lead to more power consumption.

To obtain the best blend of all, we propose an adaptive mode based on 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 13 shows an example mode transition of an 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 uses only its data prefetching functionality. If the application does not have many L2 cache misses (i.e., low 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 shown in Section 4 as it does not pollute the regular cache hierarchy. To implement an adaptive mechanism, two performance counters need to be added: an L3 cache hit counter and an L3 cache access counter (which is 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 would be more appropriate for the running application.

4 Experimental Results

4.1 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 microarchitecture pipeline, it models latency and bandwidth of the

Table 1: Host Processor Configuration

Clock frequency	3.0 GHz
Processor model	out-of-order
Machine width	3 (fetch) / 3 (issue) / 4 (retire)
The number of pipeline stages	1 (fetch) / 4 (decode) / 2 (rename) / 4 (wakeup) / 1 (schedule)
ROB size	140
Physical register file size	80 (INT) / 64 (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 cache with 64B line; 15 cycle hit latency; 3 cycle throughput
Memory	350 cycle latency, 24 GBps bandwidth

Table 2: 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*									

interconnection network among PEs including the IBus, ORTree, MBus, and mesh network. Additionally, we implemented the Chameleon functionality integrated with this simulator. The second simulator is SESC [44], 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 information measured by the SIMD PE simulator,[3] and uses them to simulate the entire heterogeneous architecture. Table 1 lists the configuration of the simulated host processor. Throughout this article, 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, 436.cactusADM, 465.tonto, and 470.lbm, which incurred 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.

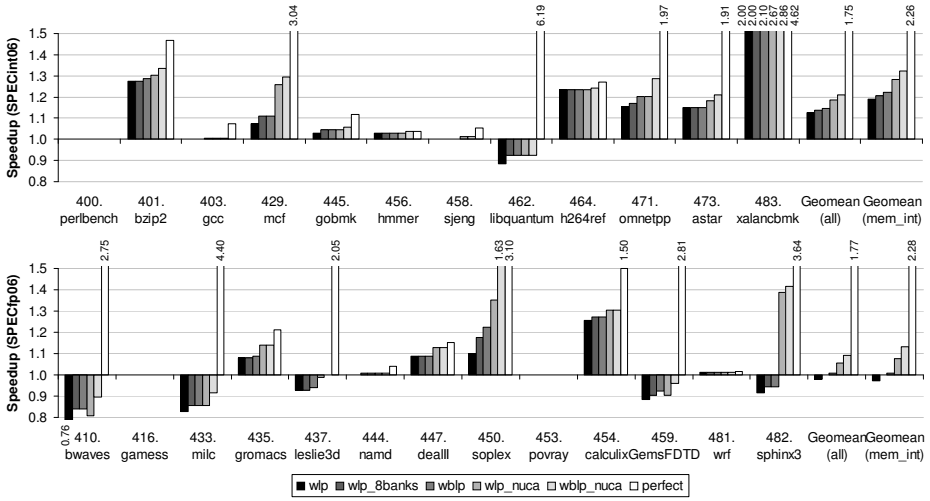


Figure 14: Relative Performance of Different C-Mode Designs

4.2 Evaluation of C-Mode

First of all, we measured the latency and throughput for each C-mode design. Unlike a conventional cache where its latency and throughput is solely determined by the characteristics of transistors and wires, the latency and throughput of a C-mode cache is also determined by the number of instructions to control PEs and their order. For example, for a cache read operation, the read latency can be reduced if the instructions routing a read-hit line back to the Chameleon Virtualizer are scheduled earlier than the instructions updating the LRU bits. On the other hand, the number of the 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 2 summarizes each cache design and their latency/throughput studied in this section. As shown in Table 2, the latency and throughput of NUCA models vary depending on which row an access hits.[4] 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, there is no need to update the LRU bits, thus 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 at 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.

[3]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.

[4]In this paper, 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.

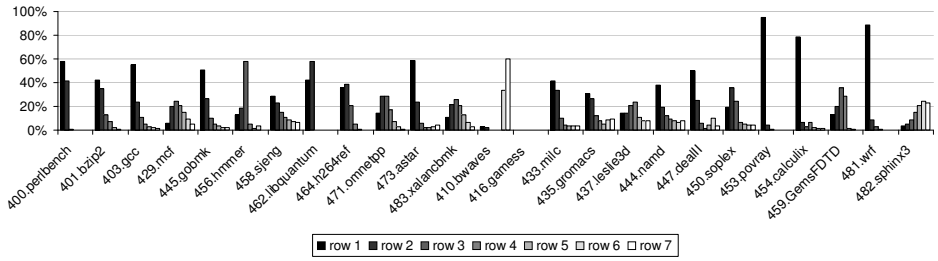


Figure 15: Distribution of Hit Rows

Table 3: Latency and Throughput of Different P-Mode Designs

Legend	Description	Latency	Throughput
stride1	1MB stride prefetcher table on 8 PEs in row 0	23	15
stride2	2MB stride prefetcher table on 16 PEs in row 0 and 1	25	
stride4	4MB stride prefetcher table on 32 PEs in row 0 to 3	29	
stride8	8MB stride prefetcher table on 64 PEs in row 0 to 7	37	
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	

Now we evaluate and quantify the performance potential for single-thread applications by using the C-mode on a heterogeneous multicore processor. Figure 14 shows the performance impact of different C-mode designs. To show the theoretical limit, we also simulated a *perfect* memory model where the L2 cache is assumed perfect. This model also reveals those benchmark programs which are memory-intensive. In this article, we define memory-intensive applications as applications whose performance can be improved more than 10% with the perfect L2 model.

Not surprisingly, the C-mode improves the performance of memory-intensive applications, such as 401.bzip2, 429.mcf, 464.h264ref, 483.xalancbmk, 450.soplex, and 454.calculix. For example, the NUCA WBLP-style (*wblp_nuca*) C-mode improves the performance of 483.xalancbmk by 186%. 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 21% and 9%, repsectively. For the memory-intensive application category, the average performance improvements for them are 32% and 13%, respectively.

However, the performance of some memory-intensive applications was degraded including 462.libquantum, 410.bwaves, 433.milc, and 459.GemsFDTD. We found that the hit rates in the C-mode cache were very low when running these applications, thus an additional cache level will only introduce extra latency to bring data back.

Apparently, the NUCA models are effective despite their longer latency when a hit PE is located far from the Chameleon Virtualizer. Figure 15 shows the distribution of hit rows. Note that 416.gamess is extremely computation-intensive and generates a small number of cold misses, resulting in a 100% L3 miss rate. Therefore, there is no bar shown in Figure 15 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 to implement time-zoning.

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 14, 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.

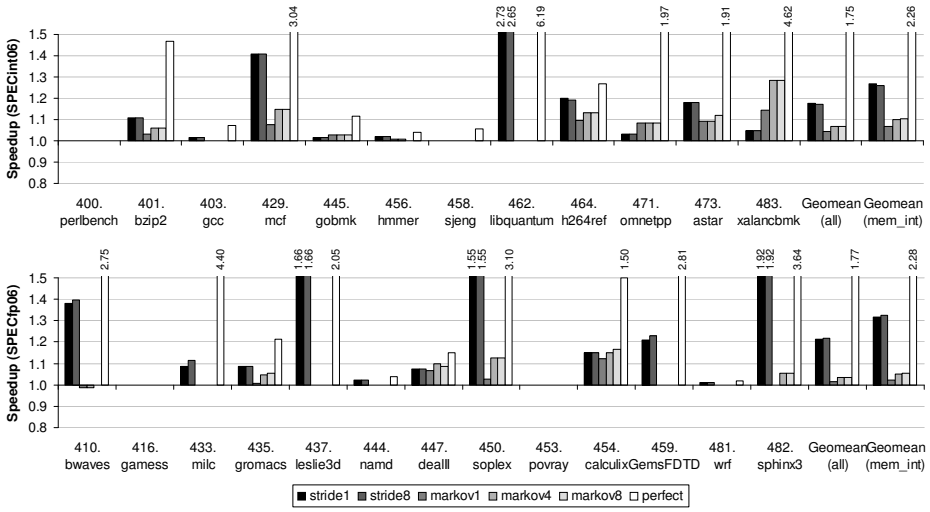


Figure 16: Relative Performance of Different P-Mode Designs

4.3 Evaluation of P-Mode

Table 3 describes each prefetcher design used in this section, and shows its table lookup latency and throughput. As expected, there is a trade-off between the table lookup latency and the table size. For example, it takes 23 cycles for the Chameleon Virtualizer to look up a 1MB stride prefetcher table while it takes 37 cycles for an 8MB table. This trade-off is represented in the overall performance graphs shown in Figure 16. 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. A larger table will be able to cover more miss addresses. However, we found a 1MB P-mode stride prefetcher is sufficiently large, for it is indexed by the program counter.

In most cases, a P-mode stride prefetcher performs better than a P-mode Markov prefetcher. Four exceptions are 471.omnetpp, 483.xalancbmk, 447.dealII, and 454.calculix where a P-mode Markov prefetcher surpasses the others. 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 stride prefetcher. Perez *et al.* also made the conclusion that a PC-indexed stride prefetcher can deliver higher performance than a Markov prefetcher [40]. On average, the 1MB P-mode stride prefetcher improved the performance of SPECint06 and SPECfp06 applications by 17% and 21%, respectively. Their average improvements for memory-intensive applications are 27% and 32%.

4.4 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 186% but only 5% improvement is obtained with a 1MB P-mode stride prefetcher. In contrast, the 1MB P-mode stride prefetcher improves the performance of 462.libquantum by 173% but using the NUCA WBLP-style C-mode cache adversely degrades it by 8%. More interestingly, Figure 17 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 shows the performance impact of the best performing C-mode (*wblp_nuca*) and P-mode (*stride1*).

In this section, we evaluate two HybridCP-mode designs: a hybrid design with a 7MB NUCA WBLP cache and a 128KB stride prefetcher (*hybrid_7MB_128KB* of Figure 12(a)), and a hybrid design with a 6MB NUCA WBLP cache

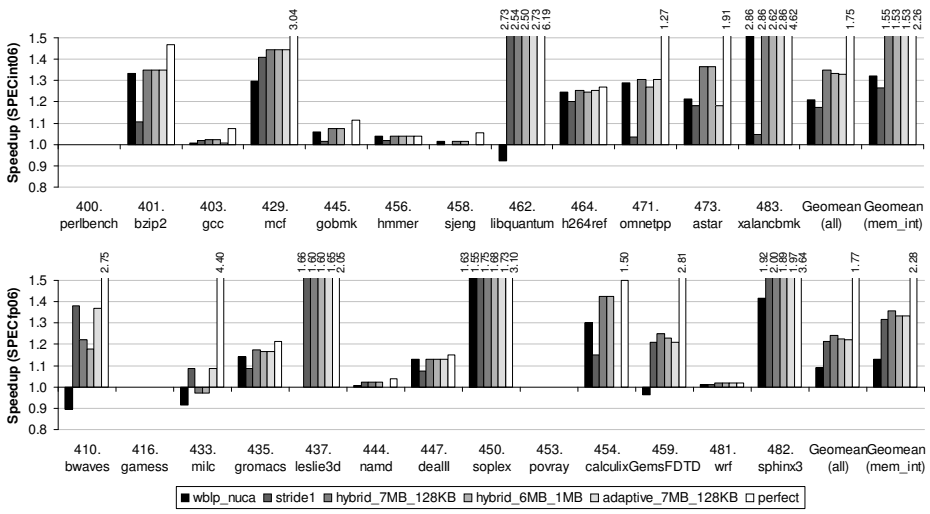


Figure 17: Relative Performance of HybridCP-Mode and AdaptiveCP-Mode

and a 1MB strider prefetcher (*hybrid_6MB_1MB* of Figure 12(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 14, 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 strider prefetcher. On average, this hybrid design improves the performance of SPECint06 and SPECfp06 by 36% and 24%. The average performance improvements for memory-intensive ones are 55% and 36%. (For the remaining of the article, the HybridCP-mode refers to the hybrid design with a 7MB cache and 128KB prefetcher, unless explictly 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 stride prefetcher, but can disable its cache functionality to behave as a 128KB stride prefetcher based on the algorithm shown in Figure 13. In this evaluation, we modeled the AdaptiveCP-mode to make decisions after warming up the cache during the first 100 million cycles and then monitoring the number of cache accesses and hits for the next 100 million cycles. The model in Figure 17 uses 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). We also studied the sensitivity of different cache hit rate thresholds from 0.1 to 0.4 for the AdaptiveCP mode and found the performance of most applications was not sensitive to them. The only exception was 429.mcf, which benefits a lot from both the C-mode and the P-mode. The performance was not sensitive to the MPKC threshold value around 0.5 either, because Chameleon is not accessed frequently.

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 33% and 22%, repectively. For memory-intensive applications, it improves by 53% and 33% 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 3% degradation.) However, there are two applications that the AdaptiveCP-mode fails to adapt to the better-performing mode: 473.astar and 454.calculix. To understand this behavior, we sampled the cache hit rate and the L2 MPKC every 100 million cycles using the HybridCP-mode only. We observed that the cache hit rate of 473.astar suddenly jumps up after the first 1.6 billion cycles, and stays high till the end. This is the reason why the AdaptiveCP-mode fails to select the HybridCP-mode, but selects the prefetch-only mode. In 454.calculix, the L2 MPKC is initially low, but it arises after the first three hundred million cycles, the main cause leading to the power-saving mode. Constant monitoring and dynamic mode selection will be needed to achieve the best performance for these applications, which lies in our future work.

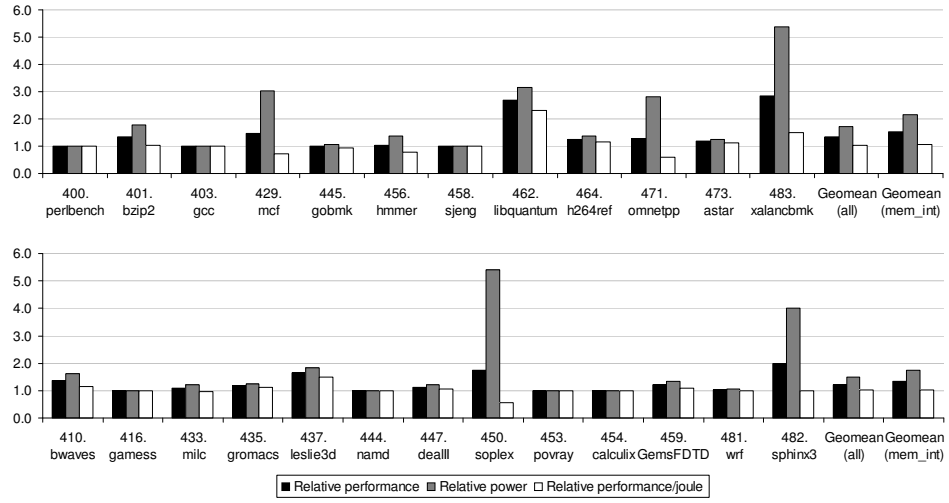


Figure 18: Performance, Power, and *Performance per Joule* of the AdaptiveCP-Mode

Up to this point, our baseline host processor has a 512KB L2. 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 30% and 20%, respectively. When increasing it to 2MB, it improved the performance of them by 25% and 19%, respectively.

4.5 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 [17] 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, xferortree and sampleortree. Third, to provide a NUCA model, we added two sets of mux and demux as shown in Figure 7(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, counting the number of accesses and the number of hits in the Chameleon cache.

We also evaluated the extra power dissipation for the AdaptiveCP-mode using Wattch [5]. We additionally modeled the global interconnect (IBus, ORTree, Mesh) power consumption using the Berkeley Predictive Technology Model [7]. 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, for convenience, we conservatively modeled that all 56 PEs are active regardless of the LRU state of a hit line.

First of all, we found that a C-mode read operation of the AdaptiveCP-mode consumes about 327 times more energy compared to a cache read operation of 2-way 32KB, dual-ported L1 cache. This result is not surprising because a C-mode read operation looks up a total of 8MB memory (64 PEs, 128KB per PE), along with tens of other instructions that consumes energy in the ALUs and register files of all PEs. Although not all 64 PEs are active after tag matching,

the control instructions for C-mode can still consume a considerable amount of energy. For a prefetch table lookup, it consumes 8 times more energy compared to a 32KB cache read operation. Unlike a cache operation, the prefecher of the AdaptiveCP-mode only uses one PE (128KB prefetch table) and fewer instructions, resulting in less energy.

Figure 18 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. Fortunately, as Chameleon is accessed very infrequently (on L2 miss), Chameleon will consume only 71% (SPECint06) and 48% (SPECfp06) more power (Figure 18) 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 [1, 31]. 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 18 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 improves the *performance per joule* of SPECint06 and SPECfp06 by 3% and 1% (1.03x and 1.01x in the geomeans). In other words, the Chameleon will not lose its energy efficiency. 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, 471.omnetpp, and 450.soplex, is degraded as shown in the figure. We found that they prefer the HybridCP-mode, thus 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 article, and it remains as our future work.

5 Implications to the Overall Design

In this section, we discuss several issues of Chameleon that are not discussed in the main text, but necessary for completeness.

Coherence support. Although it is not completely evaluated in this paper, cache coherence can be supported by the Chameleon Virtualizer. Upon receiving a coherence message, the Chameleon Virtualizer broadcasts microcode to look up coherence bits stored in one of the PEs. A state machine needs to be implemented in the Chameleon Virtualizer, so that it can perform a correct coherence action upon receiving coherence bits from a PE.

Usage model. Although the baseline PE array is typically used as an accelerator for data-parallel applications, Chameleon enhances it so that it can work better than a conventional superscalar processor, especially for memory intensive applications. If Chameleon is used in an accelerator board on a system bus, e.g., PCIe, or integrated onto a CPU chip, a smart OS can schedule memory-intensive workloads on Chameleon rather than on a conventional CPU if it fails to find a data-parallel application. Moreover, Chameleon can enhance the performance of data-parallel applications as well, because it improves the performance of their sequential portion of the code.

6 Related Work

Software caching techniques have been used by many systems. To improve the programmability of SPEs on IBM Cell/BE, IBM provided a software cache library as a part of their SDK [2, 14]. The MIT Raw processor used software caching to emulate both instruction [34] and data cache [36], while the Stanford VMP multiprocessor handled cache misses using software techniques [10]. Furthermore, with advanced compiler techniques, a software cache memory can be better managed [49, 26, 9]. The goal of all these prior works is to improve the programmability or performance of

a processor with local scratch-pad memory, while our work paper is to provide a virtualized last-level cache to the host processor to improve single-thread performance of the host processor.

To use on-chip memory resources more efficiently, researchers have focused on managing shared cache memories [28, 52, 8, 18, 21, 43, 16]. Zhang and Asanovic proposed a new cache management policy called victim replication, which combines the advantage of private and shared schemes in a tiled CMP [52]. Chang and Sohi used private cache memories for both dynamic sharing and performance isolation [8]. Harris proposed a synergetic caching policy, which groups neighboring cores into a cluster to have shared memory space among them [18]. These prior works try to address the problems of shared cache management, while our work try to address the under-utilization issue of a heterogeneous multi-core processors by virtualizing the unused PEs.

On the other hand, a NUCA cache structure has been studied to tackle a long latency problem of the last-level cache. Kim *et al.* proposed an adaptive, non-uniform cache structure [27]. Based on a NUCA model, Huh *et al.* studied an optimal degree of cache sharing [22]. Unlike the original NUCA proposal, NuRAPID decouples tag and data array to enable flexible data placement [11]. Although this paper adopts an idea of a NUCA cache, we propose an architectural solution called *time-zoning* to provide non-uniform cache access latencies on a SIMD PE engine with a strictly synchronized execution model.

Memory-Mapped I/O (MMIO) is a well-known technique that allows a CPU to assign a part of its memory space to an I/O device, and maps it to the memory space of the I/O device. MMIO is mainly used for communication between CPU and I/O devices. The PPE of IBM Cell/BE also has limited capability of accessing the memory space mapped to the local store of SPEs through its MMIO interface, but this is far less efficient than using DMA, and this operation is not synchronized with SPE execution [29]. Chameleon is completely different from MMIO. Chameleon is targeted to improve the performance of the host processor by virtualizing idle cores collectively into a cache and/or a prefetcher. There is no static address mapping involved in Chameleon. More recently, Core Fusion [23] was proposed to group independent cores to form a larger CPU dynamically as needed by applications. A Flexible Heterogeneous MultiCore processor [41] dynamically adds or removes a processor from the system to adapt to the requirement of the applications. Chameleon is different from these works, as we virtualize idle resources to provide better memory performance with minimal hardware modifications.

7 Conclusion

In this paper, we propose Chameleon, a flexible heterogeneous multi-core 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 exploiting data-level parallelism, Chameleon can virtualize these idle cores collectively into a last-level cache (C-mode), a table-based data prefetcher (P-mode), or a hybrid memory enhancing scheme of these two 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 stride prefetcher and a Markov 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 a heterogeneous multi-core processor consisting of one high performance processor core and an array of SIMD-capable processing elements 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 21% and 9% while the Chameleon P-mode can improve them by 17% and 21%. Furthermore, our hybrid mode shows a 35% and 24% improvement, respectively. In the adaptive mode, 33% and 22% were observed for SPECint06 and SPECfp06. Finally, when accounting for memory-intensive applications only from the suite, the average speedups of the adaptive mode were increased to 53% and 33% for SPECint06 and SPECfp06, respectively.

References

[1] Mayank Agarwal, Kshitiz Malik, Kevin M. Woley, Sam S. Stone, and Matthew I. Frank. Exploiting postdominance for speculative parallelization. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 295–305, Washington, DC, USA, 2007. IEEE Computer Society.

[2] Abraham Arevalo, Ricardo M. Matinata, Maharaja Pandian, Eitan Peri, Kurtis Ruby, Francois Thomas, and Chris Almond. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. IBM Publications Center (http://www.redbooks.ibm.com/abstracts/sg247575.html), 2008.

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

[4] Tom Blank. The MasPar MP-1 Architecture. In *Proceedings of COMPCON*, Spring 1990.

[5] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the International Symposium on Computer Architecture*, pages 83–94, 2000.

[6] Ian Buck. GPU computing with NVIDIA CUDA. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 6, New York, NY, USA, 2007. ACM.

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

[8] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 264–276, Washington, DC, USA, 2006. IEEE Computer Society.

[9] Tong Chen, Tao Zhang, Zehra Sura, and Mar Gonzales Tallada. 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*, pages 155–164, New York, NY, USA, 2008. ACM.

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

[11] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 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*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[12] William J. Dally and Brian Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings of the 38th Design Automation Conference*, 2001.

[13] Santanu Dutta, Rune Jensen, and Alf Rieckmann. Viper: A multiprocessor soc for advanced set-top box and digital tv systems. *IEEE Des. Test*, 18(5):21–31, 2001.

[14] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the cell processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Anwar Ghuloum, Terry Smith, Gansha Wu, Xin Zhou, Jesse Fang, Peng Guo, Byoungro So, Mohan Rajagopalan, Yongjian Chen, and Biao Chen. Future-Proof Data Parallel Algorithms and Software on Intel™ Multi-Core Architecture. In *Intel Technology Journal*, Vol. 11, Issue 4 2007.

[16] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. 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*, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

[17] F. Hamzaoglu, K. Zhang, Y. Wang, H.J. Ahn, U. Bhattacharya, Z. Chen, Y.G. Ng, A. Pavlov, K. Smits, M. Bohr, et al. 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*, pages 376–621, 2008.

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

[19] Ravi Hegde. Optimizing Application Performance on Intel® Core™ Microarchitecture Using Hardware-Implemented Prefetchers, 2008. Intel Software Network (http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers).

[20] Justin Hensley. AMD CTM overview. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 7, New York, NY, USA, 2007. ACM.

[21] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. 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*, pages 13–22, New York, NY, USA, 2006. ACM.

[22] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A nuca substrate for flexible cmp cache sharing. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 31–40, New York, NY, USA, 2005. ACM.

[23] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, 2007.

[24] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the International Symposium on Computer Architecture*, pages 252–263, New York, NY, USA, 1997. ACM.

[25] H. Kadota, J. Miyake, I. Okabayashi, T. Maeda, T. Okamoto, M. Nakajima, and K. Kagawa. A 32-bit CMOS microprocessor with on-chip cache and TLB. *IEEE Journal of Solid-State Circuits*, 22(5):800–807, 1987.

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

[27] C. Kim, D. Burger, and S.W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *ACM SIGPLAN Notices*, 37(10):211–222, 2002.

[28] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, September 2004.

[29] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE MICRO*, pages 10–23, 2006.

[30] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[31] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. Posh: a tls compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167, New York, NY, USA, 2006. ACM.

[32] Aqeel Mahesri, Daniel Johnson, Neal Crago, and Sanjay J. Patel. Tradeoffs in Designing Accelerator Architectures for Visual Computing. In *Proceedings of the International Symposium on Microarchitecture*, 2008.

[33] Michael D. McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance evaluation of gpus using the rapidmind development platform. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 181, New York, NY, USA, 2006. ACM.

[34] Jason E. Miller and Anant Agarwal. 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*, pages 293–302, New York, NY, USA, 2006. ACM.

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

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

[37] Aaftab Munshi. Opencl: Parallel computing on the gpu and cpu. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, 2008.

[38] John R. Nickolls. The Design of the MasPar MP-1: a Cost Effective Massively Parallel Computer. In *Compcon Spring 90'*, 1990.

[39] Matthew Papakipos. PeakStream Platform, 2006. SUPERCOMPUTING 2006 Tutorial on GPGPU, Course Notes (http://www.gpgpu.org/sc2006/slides/12.papakipos.peakstream.pdf).

[40] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the International Symposium on Microarchitecture*, pages 43–54, Washington, DC, USA, 2004. IEEE Computer Society.

[41] Miquel Pericas, Adrian Cristal, Francisco J. Cazorla, Ruben Gonzalez, Daniel A. Jimenez, and Mateo Valero. A flexible heterogeneous multi-core architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

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

[43] Moinuddin K. Qureshi and Yale N. Patt. 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*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[44] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[45] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[46] Howard Jay Siegel, Thomas Schwederski, IV Nathaniel J. Davis, and James T. Kuehn. Pasm: a reconfigurable parallel system for image processing. *SIGARCH Comput. Archit. News*, 12(4):7–19, 1984.

[47] H. Singh, M.H. Lee, G. Lu, FJ Kurdahi, N. Bagherzadeh, and EM Chaves Filho. MorphoSys: an integrated reconfigurable system for data-paralleland computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[48] Stephen L. Smith. Intel Roadmap Overview. In *Intel Developer Forum*, 2008.

[49] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. on Embedded Computing Sys.*, 5(2):472–511, 2006.

[50] Dong Hyuk Woo, Joshua B. Fryman, Allan D. Knies, Marsha Eng, and Hsien-Hsin S. Lee. POD: A 3D-integrated Broad-Purpose Acceleration Layer. *IEEE Micro*, July/August, 2008.

[51] Thomas Y. Yeh, Petros Faloutsos, Sanjay J. Patel, and Glenn Reinman. ParallAX: An Architecture for Real-Time Physics. In *Proceedings of the International Symposium on Computer Architecture*, 2007.

[52] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 336–345, Washington, DC, USA, 2005. IEEE Computer Society.