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

DONG HYUK WOO
Georgia Institute of Technology
and
JOSHUA B. FRYMAN and ALLAN D. KNIES
Intel Corporation
and
HSIEN-HSIN S. LEE
Georgia Institute of Technology

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 part 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 that dynamically configures the acceleration cores between the hybrid mode and the prefetch-only mode by monitoring the effectiveness of the Chameleon cache mode. In our evaluation with 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 31% and 15% on average. When considering only memory-intensive applications, Chameleon improves the system performance by 50% and 26% for SPECint06 and SPECfp06, respectively.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Array and vector processors*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Heterogeneous multi-core, idle core, cache, prefetching

New Paper, Not an Extension of a Conference Paper

1. INTRODUCTION

Heterogeneous computing has emerged to address the growing concerns of energy efficiency [Woo and Lee 2008] and silicon area effectiveness [Hill and Marty 2008]. Small-scale heterogeneous MPSoCs have been used in embedded systems for years [Dutta et al. 2001; Artieri 2005]. 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) [Pham et al. 2005] integrated eight synergistic processing elements along with a superscalar PowerPC processor on the same die. CUDA [Buck 2007], CTM [Hensley 2007], RapidMind [McCool et al. 2006], OpenCL [Munshi 2008], PeakStream [Papakipos 2006], and Ct [Ghuloum et al. 2007] abstract the programming interface to enable heterogeneous computing based on a cooperative computing model between the CPU(s) and many-core graphics processing units (GPUs), shielding 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 [Moore 2007].

Unlike a symmetric multi-core processor in which all processor cores are identical, a heterogeneous computing platform is composed of distinct classes of processing cores: a high-performance host processor and an array of acceleration processing elements (PEs) [Pham et al. 2005; Moore 2007; Smith 2008; Woo et al. 2008; Singh et al. 2000; Yeh et al. 2007; Mahesri et al. 2008]. 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 processor 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, a data prefetcher, and variants of their combinations when running sequential programs. The main contributions of this work are as follows:

- We propose the Chameleon heterogeneous multi-core architecture, which virtualizes otherwise unused acceleration cores to enable prefetching and additional

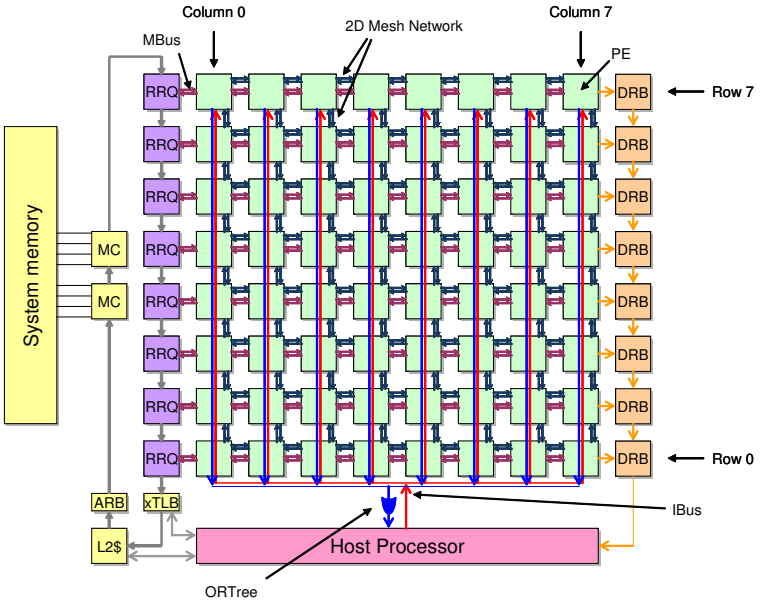


Fig. 1. Baseline Heterogeneous Multi-core Architecture

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 to virtualize the acceleration cores collectively for enhancing memory performance. In addition, we also propose an adaptive mode to change these modes dynamically to adapt the memory behavior of applications.
- We perform a case study using a heterogeneous multi-core processor that consists of a host processor integrated with an on-die massively parallel SIMD accelerator. We justify the performance benefits given by Chameleon and evaluate the hardware/power overheads and energy implications.

This paper is organized as follows: Section 2 explains the architectural features of our baseline heterogeneous multi-core processor. Section 3 enumerates different design issues for providing 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 of Chameleon's implementation, and Section 6 enumerates related work. Finally, Section 7 concludes.

2. BASELINE MULTI-CORE ARCHITECTURE

Although the core concept of Chameleon, utilizing idle acceleration cores for improving the performance of sequential workloads, can be generally applied to many

heterogeneous multi-core processors, the detailed design as well as its benefit and associated hardware overheads will vary for different heterogeneous architectures. In this article, we investigated a Chameleon architecture based on a recently proposed accelerator architecture [Woo et al. 2008] and evaluated its cost and performance potentials. We will later briefly discuss how Chameleon can be applied to other architectures in Section 5.

Figure 1 illustrates our baseline heterogeneous multi-core processor used throughout this paper [Woo et al. 2008]. It consists of a state-of-the-art superscalar host processor and an array of 8 × 8 PE cores. The size of the PE array is not fixed and can change depending on the design budget, process technology and 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* conditional statements, masking instructions are also included to enable or disable each PE individually for given 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 (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 direct memory access (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 because all execution is fully orchestrated by the host processor, communication patterns are completely deterministic and exempt from 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 [Dally and Towles 2001; Siegel et al. 1984]

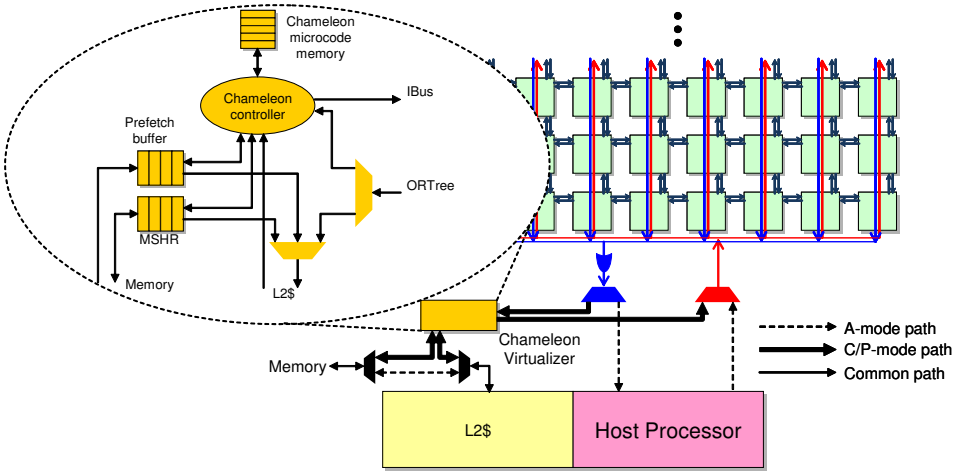


Fig. 2. Chameleon Virtualizer (not drawn to scale)

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. With 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 changes Chameleon among different modes to find the best possible performance by exploiting 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 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 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.

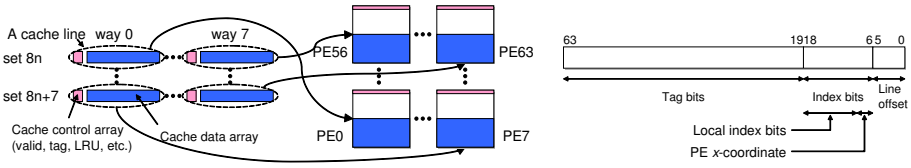
To control the PEs array and to have it function like a soft cache, we add a new

interface between the L2 cache of the host processor and the baseline PE array. As shown in Figure 2, 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 3.2. 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 that hits 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 will be 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 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. **(***HHL: I suggest to add the width of ORTree bus to Figure 2, the PE array tiles figure. ***)** 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 non-uniform cache architecture (NUCA) and discuss the required architectural support.

3.1.1 *Design for Way-Level Parallelism.* **(***HHL: Please proof-read this section carefully. I made changes for many missing info, but not sure they are absolutely correct. ***)** Our first design is to distribute multi-way cache lines of the same set across PEs in the same column. Figure 3 shows an example mapping of an eight-way set-associative 4MB cache. In this example, eight cache lines



(a) Layout (b) Indexing

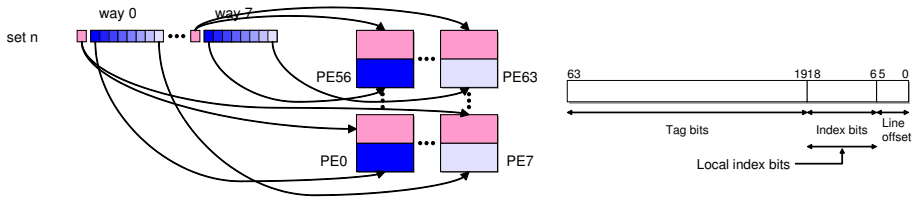
Fig. 3. Way-Level Parallelism (8-Way Cache, 4MB)

(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 LSBs are used to find the x-coordinate of 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, thus up to 1,024 cache lines can be stored inside each PE. 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* [Kadota et al. 1987] 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 desig.

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 different columns 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.



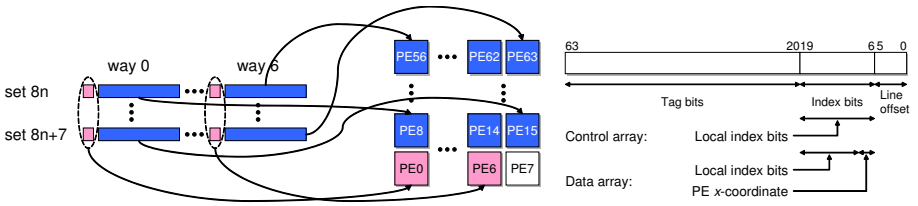
(a) Layout (b) Indexing

Fig. 4. Way- and Subblock-Level Parallelism (8-Way Cache, 4MB)

3.1.2 *Design for Way- and Subblock-Level Parallelism.* **(***HHL: Why can't we simply call this Subblock Level Parallelism, why WBLP? ***)** 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), 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 3. 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 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. 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.

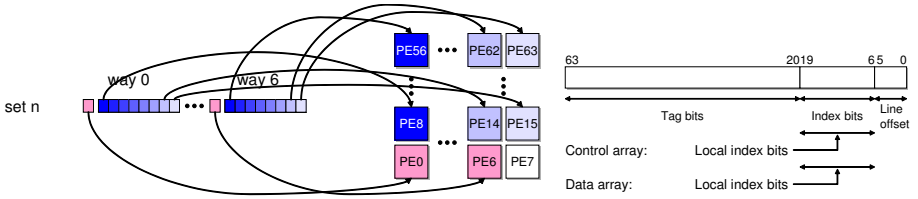
The primary challenge of such a WBLP design is the area overhead in keeping the cache control bits. As a cache line is split into eight subblocks, all eight PEs that keep a subblock of the same cache line need to have redundant valid, tag, LRU and coherence bits. Otherwise, more delay will incur for communicating this information. We found that each PE can accommodate this redundant information without sacrificing the overall cache capacity. As explained in Section 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 (i.e., each 128KB scratchpad memory is quite under-utilized.) In the WBLP design, at most 64b overhead is

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



(a) Layout (b) Indexing

Fig. 5. Decoupled WLP Cache (7-Way Cache, 7MB)



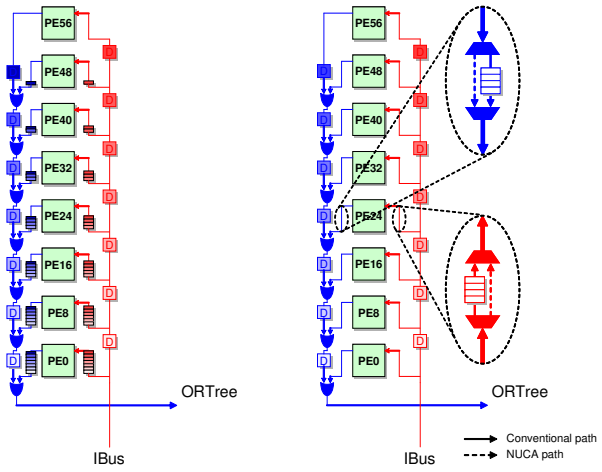
(a) Layout (b) Indexing

Fig. 6. Decoupled WBLP Cache (7-Way Cache, 7MB)

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 capacity.

3.1.3 *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.

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 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 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



(a) Conventional, Synchronized Model

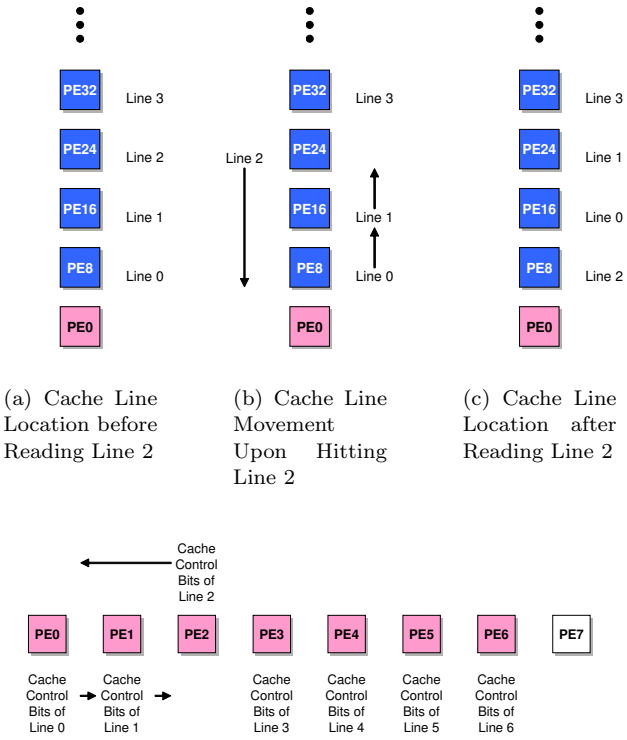
(b) Pipelined Model to Support NUCA

Fig. 7. Execution Model (Only Column 0 is shown.)

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 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 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 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.

3.1.4 *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 7(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



(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

Fig. 8. LRU Management of a NUCA C-Mode (Decoupled WLP cache)

be beneficial to have non-uniform access latencies, i.e., accessing each PE row out-of-sync. As shown in previous studies [Kim et al. 2002; Huh et al. 2005; Chishti et al. 2003], 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 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, 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 directly bypasses and 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 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 as 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 no data is corrupted between distinct memory accesses. The northbound transfer instruction can be synchronized by adding one more pipeline register in the northbound output mesh driver while the southbound transfer instruction can be synchronized by architecting the latency of this instruction as two cycles. More details on the time-zone effect can be found in [Woo et al. 2008].

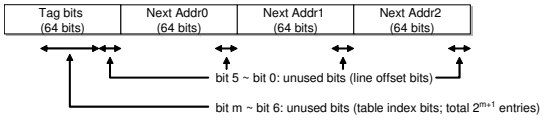
Another NUCA design issue is with respect to how to implement the LRU policy efficiently. Figure 8 illustrates an instance for our decoupled WLP cache. In this example, the host processor issues to read *line 2*, and seven PEs have seven different cache lines mapped to the same set as shown in Figure 8(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 8(b)). This movement allows PEs to maintain more recently used cache lines closer to the Chameleon Virtualizer as shown in Figure 8(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). **(***HHL: I was lost when reading this paragraph. I do not understand the explanation of this instruction. I was also lost of the figure 8(d) and its purpose. ***)** 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.

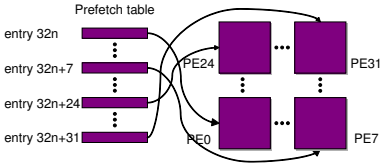
The final design consideration of our NUCA C-mode is the placement of the cache control array. In a decoupled design, the cache control array is located in row 0, nearest to the Chameleon Virtualizer, reducing the tag lookup latency significantly. Our NUCA design adopts a decoupled design as its base 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 8(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 3, the Chameleon Virtualizer can find corresponding data line in row 4.

3.2 P-Mode: Virtualizing Idle Cores as a Prefetcher

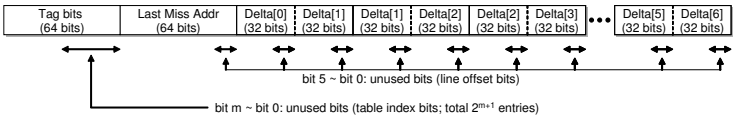
In addition to the C-mode that supplies a virtualized last-level cache, we also investigate the enabling mechanisms to reconfiguring idle PEs to work as a data prefetcher. The rationale behind this is from the following observation— the off-chip bandwidth of a heterogeneous multi-core processor is typically 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).



(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

Fig. 9. P-Mode Prefetcher

In this article, we evaluate two data prefetchers for Chameleon: a Markov prefetcher [Joseph and Grunwald 1997] and a program counter (PC)-indexed delta correlation prefetcher [Kandiraju and Sivasubramaniam 2002; Nesbit and Smith 2004]. 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 [Hegde 2008] or a stride prefetcher [Tendler et al. 2001]. 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 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.

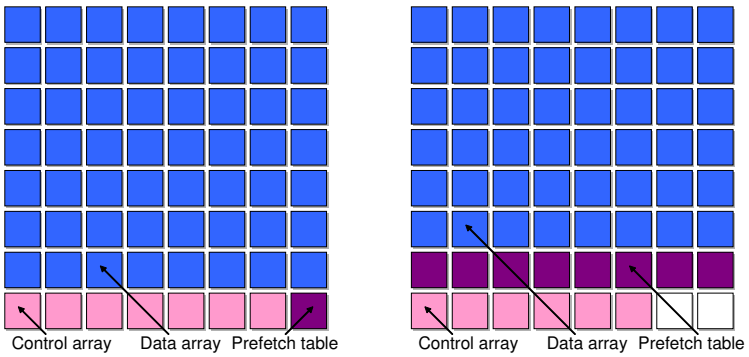
3.2.1 *Virtualized Markov Prefetcher.* Figure 9(a) shows the prefetch table design for a virtualized Markov prefetcher. Although the original Markov prefetcher paper [Joseph and Grunwald 1997] showed that a prefetch table with four next-miss addresses provides a reasonable balance between coverage and 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 9(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 on 32 PEs) from a PC. 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 9(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 paper, we vary the size of the prefetch table (1, 2, 4, and 8 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.

3.2.2 *Virtualized Delta Correlation Prefetcher.* In addition to a Markov prefetcher, we also evaluate a delta correlation prefetcher [Kandiraju and Sivasubramaniam 2002; Nesbit and Smith 2004] 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 9(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. **(***HHL: In the figure, to be less confusing, why don't you just draw all the 6 delta pairs, because there are only four missing from the figure: 3, 4, 4, 5. ***)** 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 9(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 9(c)).



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

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

Fig. 10. Hybrid Design (Cache + Prefetcher)

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 (e.g., 17 bits on 32 PEs). As in the P-mode Markov prefetcher, these index bits are 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.

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 10(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 10(b) shows another example of the HybridCP-mode

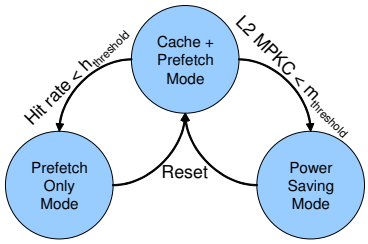


Fig. 11. AdaptiveCP-Mode (MPKC: Misses Per Kilo Cycles)

in which a 6-way 6MB cache is co-located with a 1MB stride prefetcher table. In this example, the virtualized last-level cache is shrunk, but the prefetch table is enlarged. Note that certain partioning such as a 4MB cache and a 3MB prefetch table may be infeasible due to PE indexing.

3.4 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 11 shows an example of mode transition for 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 4 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. Furthermore, we can perform such sampling regularly by reseting the state diagram to find a better mode when

Table I. 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	4 64-bit channels, 800MHz double data rate, 350 cycle latency

a program phase changes.

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 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 [Renau et al. 2005], 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[3] information measured by the baseline SIMD simulator and uses them to simulate the entire heterogeneous architecture. Table I lists the configuration of the simulated host processor. Unless otherwise stated, the capacity of our baseline L2 cache is 512KB, which is the capacity of the L2 cache used in the IBM Cell/BE [Pham et al. 2005] (We also show simulation results with 1MB and 2MB baseline models later). **(***HHL: Why all of a suddent you mentioned IBM capacity here? It makes it sound like you will compare performance against CELL. If there is no particular reason, I suggest we remove it. ***)** Throughout this article, the baseline performance is measured with this host processor model without any Chameleon capability.

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

[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.

Table II. 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*									

and simulated next two billion instructions.

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 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 their order. **(***HHL: What does "their order" mean? ***)** For example, for a cache read operation, a read latency can be reduced if instructions that route a read-hit line back to the Chameleon Virtualizer are scheduled earlier than instructions that update 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 II summarizes each cache design and their latency/throughput studied in this section. As shown in Table II, the latency and throughput of NUCA models vary depending on which row an access hits[4]. Furthermore, even in non-NUCA

[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

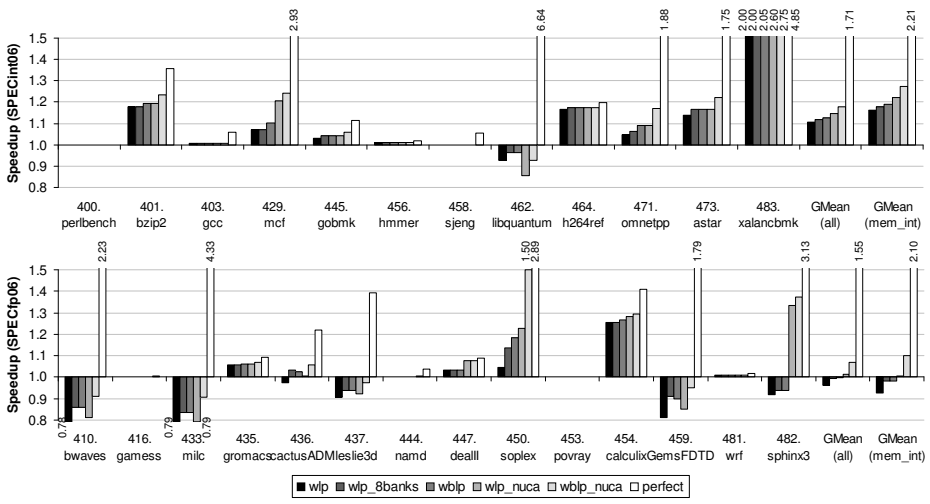


Fig. 12. Relative Performance of Different C-Mode Designs

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 multi-core processor. Figure 12 shows the performance impact of different C-mode designs. To show the theoretical limit, 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

which the hit PE belongs.

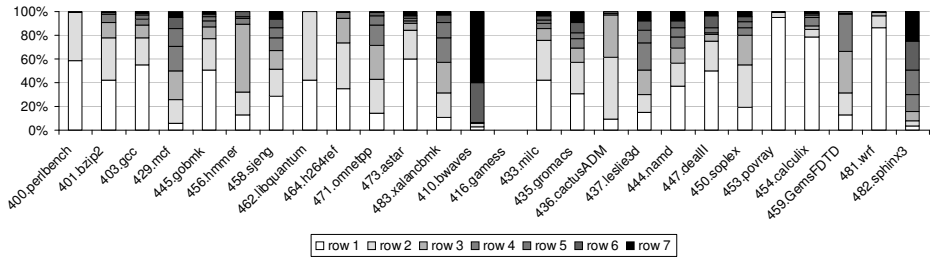


Fig. 13. Distribution of Hit Rows

(*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 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 13 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 13 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 implementing time-zoning. **(***HHL: I think we discussed under the name "time-zone effect" as an issue. Did we ever christen the name "time-zoning", a technique to improve it? ***)**

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

Table III. 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	

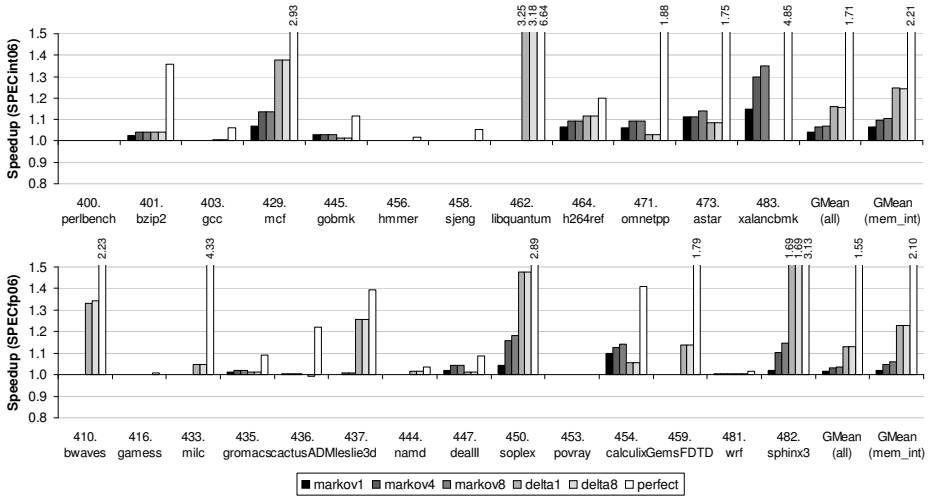


Fig. 14. Relative Performance of Different P-Mode Designs

4.3 Evaluation of P-Mode

Table III 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 14. 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-

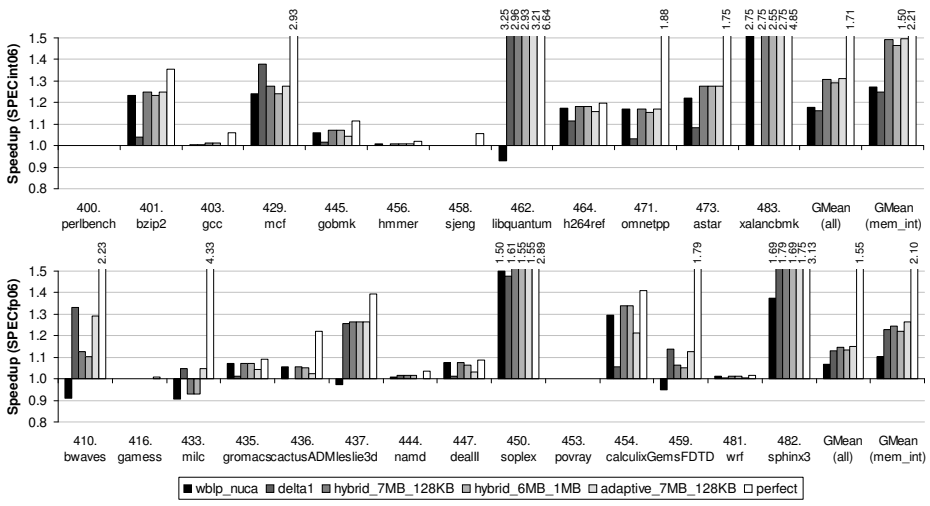


Fig. 15. Relative Performance of HybridCP-Mode and AdaptiveCP-Mode

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

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 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 15 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*).

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 10(a)) and a hybrid design with a 6MB NUCA WBLP cache and a 1MB delta correlation prefetcher (*hybrid_6MB_1MB* of Figure 10(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 12, 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 31% and 15%. The average performance improve-

ments 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 stride prefetcher. However, we disable its cache functionality to behave as a 128KB delta correlation prefetcher based on the algorithm shown in Figure 11. 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 15 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).

4.5 Hardware and Power Overhead

(*HHL: Not clear to me if the power is only dynamic power and no leakage power was evaluated. Please clarify in the paper. ***)** 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 [Hamzaoglu et al. 2008] 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 instruc-

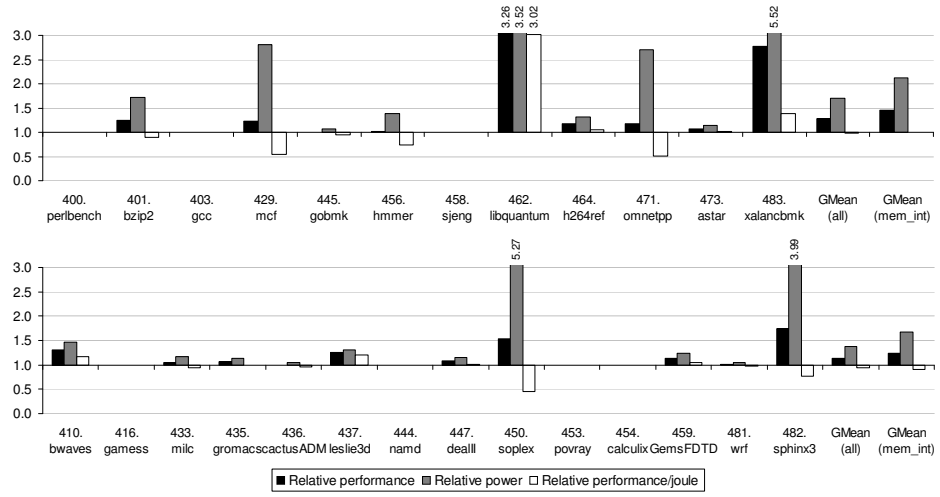


Fig. 16. Performance, Power, and *Performance per Joule* of the AdaptiveCP-Mode

tion. 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 power dissipation for the AdaptiveCP-mode using Wattch [Brooks et al. 2000] model. We additionally modeled the global interconnect (IBus, ORTree, Mesh) power consumption using the Berkeley Predictive Technology Model [Cao et al. 2000]. 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 326 times more energy compared to a cache read operation of 2-way 32KB, dual-ported L1 cache. **(***HHL: 326 times sound too scary if the baseline is already large. Say, if the baseline is 10 watts, 326 times, well.. can't do. What is the absolute joules estimated? ***)** 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 18 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, which results in less energy.

Figure 16 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 an L2 miss), Chameleon will consume only 69% (SPECint06) and 37% (SPECfp06) more power (Figure 16) 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 [Agarwal et al. 2007; Liu et al. 2006]. 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 16 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 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.

Context switching. The C-mode cache evaluated in this paper is based on a write-through policy. By using a write-through policy, we can avoid the overhead of context swtiching, a massive data movement from scrach-pad memories to the main memory when a newly scheduled data-parallel application wants to use scratch-pad memories. Because ample memory bandwidth is left unused due to idleness of acceleration cores, one traditional disadvantage of a write-through policy, consuming more bus bandwidth, is not critical in the Chameleon architecture. All our simulation results account for all write-through traffic.

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 (PEs in row 0 in the NUCA design.) 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.

Dynamic voltage and frequency scaling. Instead of using Chameleon, while PEs are idle, the host processor can exploit remaining power budget by dynamically increasing its clock frequency and supply voltage [Kim et al. 2008; Kumar and Hinton 2009]. Because idle PEs consume a small amount of idle power, the host processor can run at a higher clock frequency without exceeding the overall power budget of the chip. Nonetheless, this method will not be able to improve the performance of memory-intensive, single-thread applications, which are typically unscalable and insensitive to the clock frequency. Chameleon provides an attractive alternative to improve the single-thread performance of memory-intensive applications as shown in previous sections.

Extension to other architecture. Although we used a heterogeneous multicore processor integrated with an acceleration PE array to demonstrate our techniques, the general idea of virtualzing idle cores is not limited to such platforms but applicable to other similar types of multicore or many-core architectures. Although it is impossible to address all different architecture-specific design issues in a single paper, here we briefly describe how can our idea be applied to other architectures. For example, to provide a virtualized last-level cache in the IBM Cell/BE, an interface similar to the Chameleon Virtualizer can be implemented between a power processing element (PPE) and synergetic processing elements (SPEs), forwarding L2 miss traffic to the SPEs instead of sending it directly to the off-chip DRAMs. The new interface may need to broadcast only an L2 miss address along with a small amount of control signals since IBM Cell SPEs store their code internally. Depending on the lookup outcome, the new interface can route the return message either to the PPE L2 or to the main memory. As Cell SPEs can be used for MIMD processing, they are more suitable to have a highly concurrent multi-banked cache than SIMD PEs. The current capacity of the the Cell PPE's L2 is 512KB while the aggregate capacity of all local stores of eight SPEs is 2MB. When the number of SPEs scales in the future, the aggregate capacity of local stores will be proportionally larger, making Chameleon on the future Cell processors more attractive. Another potential platform to apply Chameleon is on-chip integrated GPUs. An interface similar to the Chameleon Virtualizer can be added between an L2 cache and the integrated GPU, and it can forward an L2 miss address to the GPU. Although a GPU does not have much cache or scratch-pad memory space, it has very large register files to enable massive multithreading for hiding a cache miss latency.

Thus, instead of relying on a scratch-pad memory space, we can record prefetch history in the register file of the GPU.

6. RELATED WORK

Prior studies investigated speculative multi-threading or helper thread type of techniques [Sohi et al. 1995; Hammond et al. 1998; Dundas and Mudge 1997; Collins et al. 2001; Luk 2001; Annavaram et al. 2001; Liao et al. 2002; Mutlu et al. 2003] to boost single-thread performance by utilizing idle cores. However, these techniques require accleration cores to be completely re-designed to support them, which could lead to severe performance degradation when running conventional data-parallel applications. An event-driven helper thread emulates the behavior of a hardware prefetcher on a closely-coupled homogeneous idle core [Ganusov and Burtscher 2006]. Others implemented technique that exploits remaining power budget by scaling the clock frequency and the supply voltage of an active core while the other cores are idle [Kim et al. 2008; Kumar and Hinton 2009].

On the other hand, Cong *et al.* proposed core spilling when resources on one core are exhausted [Cong et al. 2007]. Similarly, Core Fusion [Ipek et al. 2007] was proposed to group independent cores to form a larger CPU dynamically as needed by applications. A flexible heterogeneous multi-core processor [Pericas et al. 2007] dynamically adds or removes a processor from the system to adapt to the requirement of the applications.

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 [Arevalo et al. 2008; Eichenberger et al. 2005]. The MIT Raw processor used software caching to emulate both instruction [Miller and Agarwal 2006] and data cache [Moritz et al. 1999] while the Stanford VMP multiprocessor handled cache misses using software techniques [Cheriton et al. 1986]. Furthermore, with advanced compiler techniques, a software cache memory can be better managed [Udayakumaran et al. 2006; Kandemir et al. 2001; Chen et al. 2008]. 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. On the other hand, several studies proposed a reconfigurable memory architecture that can be configured to behave either as a cache memory or as a scratch-pad memory [Mai et al. 2000; Sankaralingam et al. 2003]. These memory architectures require both a data and a tag array to function as a cache memory. **(***HHL: I know there is another Smart Memory paper from Stanford in this ISCA 2009. Please check out to see if there is anything to cite. ***)**

To use on-chip memory resources more efficiently, researchers have focused on managing shared cache memories [Kim et al. 2004; Zhang and Asanovic 2005; Chang and Sohi 2006; Harris 2005; Hsu et al. 2006; Qureshi and Patt 2006; Guo et al. 2007]. 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 [Zhang and Asanovic 2005]. Chang and Sohi used private cache memories for both dynamic sharing and performance isolation [Chang and Sohi 2006]. Harris proposed a synergistic caching policy, which groups neighboring cores

into a cluster to have shared memory space among them [Harris 2005]. These prior studies 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 [Kim et al. 2002]. Based on a NUCA model, Huh *et al.* studied an optimal degree of cache sharing [Huh et al. 2005]. Unlike the original NUCA proposal, NuRAPID decouples tag and data array to enable flexible data placement [Chishti et al. 2003]. 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 [Kistler et al. 2006]. 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.

7. CONCLUSION

In this article, 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 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 a heterogeneous multi-core processor that consists of one high performance processor core and an array of SIMD-capable processing elements for the case study to demonstrate our techniques in this article. 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 50% and 26% for SPECint06 and SPECfp06, respectively.

REFERENCES

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

Annavaram, M., Patel, J., and Davidson, E. 2001. Data prefetching by dependence graph precomputation. In *Proceedings of the International Symposium on Computer Architecture*. 52–61.

Arevalo, A., Matinata, R. M., Pandian, M., Peri, E., Ruby, K., Thomas, F., and Almond, C. 2008. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. IBM Publications Center (http://www.redbooks.ibm.com/abstracts/sg247575.html).

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

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

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

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

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

Chen, T., Zhang, T., Sura, Z., and Tallada, M. G. 2008. 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*. ACM, New York, NY, USA, 155–164.

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

Chishti, Z., Powell, M. D., and Vijaykumar, T. N. 2003. 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*. IEEE Computer Society, Washington, DC, USA, 55.

Collins, J., Wang, H., Tullsen, D., Hughes, C., Lee, Y., Lavery, D., and Shen, J. 2001. Speculative precomputation: long-range prefetching of delinquentloads. In *Proceedings of the International Symposium on Computer Architecture*. 14–25.

Cong, J., Guoling, H., Jagannathan, A., Reinman, G., and Rutkowski, K. 2007. Accelerating sequential applications on CMPs using core spilling. *IEEE Transactions on Parallel and Distributed Systems 18,* 8, 1094–1107.

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

Dundas, J. and Mudge, T. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing*. ACM New York, NY, USA, 68–75.

Dutta, S., Jensen, R., and Rieckmann, A. 2001. Viper: A multiprocessor soc for advanced set-top box and digital tv systems. *IEEE Des. Test 18,* 5, 21–31.

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.

2005. Optimizing compiler for the cell processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Washington, DC, USA, 161–172.

Ganusov, I. and Burtscher, M. 2006. Efficient emulation of hardware prefetchers via event-driven helper threading. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 144–153.

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

Guo, F., Solihin, Y., Zhao, L., and Iyer, R. 2007. 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*. IEEE Computer Society, Washington, DC, USA, 343–355.

Hammond, L., Willey, M., and Olukotun, K. 1998. Data speculation support for a chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

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

Harris, S. 2005. SYNERGISTIC CACHING IN SINGLE-CHIP MULTIPROCESSORS. Ph.D. thesis, stanford university.

Hegde, R. 2008. Optimizing Application Performance on Intel® Core^{TM} 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).

Hensley, J. 2007. AMD CTM overview. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*. ACM, New York, NY, USA, 7.

Hill, M. and Marty, M. 2008. Amdahl's Law in the Multicore Era. *Computer 41,* 7, 33–38.

Hsu, L. R., Reinhardt, S. K., Iyer, R., and Makineni, S. 2006. 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*. ACM, New York, NY, USA, 13–22.

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

Ipek, E., Kirman, M., Kirman, N., and Martinez, J. F. 2007. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*.

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

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

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

Kandiraju, G. B. and Sivasubramaniam, A. 2002. Going the distance for tlb prefetching: an application-driven study. In *Proceedings of the International Symposium on Computer Architecture*.

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

Kim, S., Chandra, D., and Solihin, Y. 2004. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 111–122.

Kim, W., Gupta, M., Wei, G., and Brooks, D. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the International Symposium on High Performance Computer Architecture*.

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

Kumar, R. and Hinton, G. 2009. A Family of 45nm IA Processors. In *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference*. 58–59.

Liao, S. S., Wang, P. H., Wang, H., Hoflehner, G., Lavery, D., and Shen, J. P. 2002. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 117–128.

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

Luk, C.-K. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the International Symposium on Computer Architecture*. 40–51.

Mahesri, A., Johnson, D., Crago, N., and Patel, S. J. 2008. Tradeoffs in Designing Accelerator Architectures for Visual Computing. In *Proceedings of the International Symposium on Microarchitecture*.

Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W. J., and Horowitz, M. 2000. Smart memories: a modular reconfigurable architecture. In *Proceedings of the International Symposium on Computer Architecture*. Vol. 28.

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

Miller, J. E. and Agarwal, A. 2006. 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*. ACM, New York, NY, USA, 293–302.

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

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

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

Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. 2003. 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*.

Nesbit, K. and Smith, J. 2004. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 96–106.

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

Pericas, M., Cristal, A., Cazorla, F. J., Gonzalez, R., Jimenez, D. A., and Valero, M. 2007. A flexible heterogeneous multi-core architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Washington, DC, USA, 13–24.

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.

2005. The Design and Implementation of a First-Generation CELL Processor. In *Proceedings of the 2005 IEEE International Solid-State Circuits Conference*.

Qureshi, M. K. and Patt, Y. N. 2006. 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*. IEEE Computer Society, Washington, DC, USA, 423–432.

Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., and Montesinos, P. 2005. SESC simulator. http://sesc.sourceforge.net.

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

Siegel, H. J., Schwederski, T., Nathaniel J. Davis, I., and Kuehn, J. T. 1984. Pasm: a reconfigurable parallel system for image processing. *SIGARCH Comput. Archit. News 12,* 4, 7–19.

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

Smith, S. L. 2008. Intel Roadmap Overview. In *Intel Developer Forum*.

Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. 1995. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture*. 414–425.

Tendler, J., Dodson, S., Fields, S., Le, H., and Sinharoy, B. 2001. POWER4 system microarchitecture. IBM Technical White Paper.

Udayakumaran, S., Dominguez, A., and Barua, R. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. on Embedded Computing Sys. 5,* 2, 472–511.

Woo, D. H., Fryman, J. B., Knies, A. D., Eng, M., and Lee, H.-H. S. 2008. POD: A 3D-integrated Broad-Purpose Acceleration Layer. *IEEE Micro 28,* 4 (July/August), 28–40.

Woo, D. H., Fryman, J. B., Knies, A. D., and Lee, H.-H. S. 2008. Chameleon: Virtualizing Idle Acceleration Cores of A Heterogeneous Multi-Core Processor for Caching and Prefetching. Tech. rep., Georgia Institute of Technology. GIT-CERCS-08-11.

Woo, D. H. and Lee, H.-H. S. 2008. Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era. *IEEE Computer 41,* 12, 24–31.

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

Zhang, M. and Asanovic, K. 2005. 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*. IEEE Computer Society, Washington, DC, USA, 336–345.

Received December 2008; revised Month Year; accepted Month Year