Measuring Microarchitectural Details of Multi- and Many-Core Memory Systems through Microbenchmarking

ZHENMAN FANG, University of California Los Angeles
SANYAM MEHTA, PEN-CHUNG YEW and ANTONIA ZHAI, University of Minnesota
JAMES GREENSKY and GAUTHAM BEERAKA, Intel
BINYU ZANG, Shanghai Jiao Tong University

As multicore and many-core architectures evolve, their memory systems are becoming increasingly more complex. To bridge the latency and bandwidth gap between the processor and memory, they often use a mix of multilevel private/shared caches that are either blocking or nonblocking and are connected by high-speed network-on-chip. Moreover, they also incorporate hardware and software prefetching and simultaneous multithreading (SMT) to hide memory latency. On such multi- and many-core systems, to incorporate various memory optimization schemes using compiler optimizations and performance tuning techniques, it is crucial to have microarchitectural details of the target memory system. Unfortunately, such details are often unavailable from vendors, especially for newly released processors.

In this article, we propose a novel microbenchmarking methodology based on short elapsed-time events (SETEs) to obtain comprehensive memory microarchitectural details in multi- and many-core processors. This approach requires detailed analysis of potential interfering factors that could affect the intended behavior of such memory systems. We lay out effective guidelines to control and mitigate those interfering factors. Taking the impact of SMT into consideration, our proposed methodology not only can measure traditional cache/memory latency and off-chip bandwidth but also can uncover the details of software and hardware prefetching units not attempted in previous studies. Using the newly released Intel Xeon Phi many-core processor (with in-order cores) as an example, we show how we can use a set of microbenchmarks to determine various microarchitectural features of its memory system (many are undocumented from vendors). To demonstrate the portability and validate the correctness of such a methodology, we use the well-documented Intel Sandy Bridge multicore processor (with out-of-order cores) as another example, where most data are available and can be validated. Moreover, to illustrate the usefulness of the measured data, we do a multistage coordinated data prefetching case study on both Xeon Phi and Sandy Bridge and show that by using the measured data, we can achieve 1.3X and 1.08X performance speedup, respectively, compared to the state-of-the-art Intel ICC compiler. We believe that these measurements also provide useful insights into memory optimization, analysis, and modeling of such multicore and many-core architectures.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement Techniques

General Terms: Measurement, Performance

Additional Key Words and Phrases: Microbenchmarking, memory microarchitecture, prefetching, many-core, multicore

---

This article presents new work and is not an extension of a conference paper. This work was supported by NSF grants CNS-0834599, CCF-0708822, CCF-0916583, CCF-0903427, CSR-0834599, and CPS-0931931, and SRC grant SRC-2008-TJ-1819.

Authors' addresses: Z. Fang, Computer Science Department, UCLA; email: fangzhenman@gmail.com; S. Mehta, P.-C. Yew, and A. Zhai, Department of Computer Science and Engineering, University of Minnesota at Twin Cities; emails: {sanyam, yew, zhai}@cs.umn.edu; J. Greensky and G. Beeraka, Intel; emails: james.j.greensky@intel.com, beer0092@umn.edu; B. Zang, School of Software, Shanghai Jiao Tong University; email: binyu.zang@gmail.com.

DOI: http://dx.doi.org/10.1145/2687356

ACM Reference Format:
Zhenman Fang, Sanyam Mehta, Pen-Chung Yew, Antonia Zhai, James Greensky, Gautham Beeraka, and Binyu Zang. 2014. Measuring microarchitectural details of multi- and many-core memory systems through microbenchmarking. ACM Trans. Architec. Code Optim. 11, 4, Article 55 (December 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2687356

1. INTRODUCTION

Modern memory systems are becoming more and more complex. With the advent of multicore and many-core processors, the latency and bandwidth gap between the processor and memory continues to grow. To bridge this gap, many microarchitectural features have been developed and added that make the memory systems even more complex. Those features often include multilevel private/shared caches that are either blocking or nonblocking, fast interconnect-on-chip, hardware/software data prefetching, and simultaneous multithreading (SMT), to name just a few. For example, the newly released Intel® Xeon Phi™processor (codename Knights Corner) [Chrysos 2012] has up to 61 in-order cores on a single chip and incorporates many of those features.

It is crucial to know the microarchitectural details of such memory systems because compiler optimizations and performance tuning strategies depend heavily on how to exploit those microarchitectural features. For example, in addition to taking the latency at each level of cache hierarchy and the memory bandwidth into account, it is also useful for a compiler to know how many outstanding memory requests can be supported so that it will not generate too many prefetching instructions at a time, which can cause severe resource contention. It is also useful to know the behavior of the hardware prefetchers and how they are triggered, as well as the effect of SMT on data prefetching, so that programmers and compilers can take advantage of those microarchitectural supports. Unfortunately, many of such microarchitectural details are unavailable in the published data sheets, especially for newly released processors.

Prior studies [Fang et al. 2013; Molka et al. 2009; Babka and Tuma 2009; Juckeland et al. 2004; McCalpin 2014; LMbench 2014] mainly focused on measuring memory latency and bandwidth. Most have not considered other aspects of the memory microarchitecture such as data prefetching; however, data prefetching has become one of the crucial memory latency hiding techniques and is gaining more attention [Srinath et al. 2007; Ebrahimi et al. 2009; Son et al. 2009; Kamruzzaman et al. 2011; Jiménez et al. 2012; Lee et al. 2012]. Moreover, most prior work measured memory details using long elapsed-time events (LETEs). Such LETE-based approaches run a memory event (e.g., a cache miss) a large number of times, taking more than tens of thousands of cycles to tolerate timing variations. They then take the average and attribute it to that memory event (e.g., cache miss penalty). Those techniques are not appropriate for short elapsed-time events (SETEs), which can only be observed in tens or hundreds of clock cycles and thus cannot tolerate large variations. For example, they cannot observe *a latency spike that is tens or hundreds of cycles* after all miss status holding registers (MSHR) entries are filled,[1] or whether it is a cache hit or miss for a *specific* cache access.

In this article, we propose a novel SETE-based microbenchmarking methodology to measure memory microarchitectural details on multicore and many-core processors that exploit either in-order or out-of-order cores. We aim to provide guidance for the microarchitecture measurements and make the methodology viable to a wider audience,

[1]MSHR [Tuck et al. 2006] is a key hardware component for cache miss handling. It holds the cache miss requests and outstanding prefetches. It thus limits the number of outstanding memory requests to the next level of cache memory hierarchy.

including those outside the microarchitecture community. Our goals are to measure (1) cache/memory access latency at a cache hierarchy, latency at on-chip interconnect, and miss penalty of a translation lookaside buffer (TLB), as well as (2) effective off-chip bandwidth. Moreover, we aim at measuring many prefetching-related parameters not attempted in prior studies, which include (1) software prefetching parameters such as triggering conditions, maximum number of outstanding software prefetches allowed (i.e., the size of MSHRs), and shared and exclusive (for stores) prefetching behavior, and (2) hardware prefetching parameters such as triggering conditions, prefetching distances and prefetching degrees, maximum number of outstanding prefetching streams, and prefetching behavior with different page sizes. In all of those measurements, we also take the impact of SMT into consideration.

Considering that our SETE-based microbenchmarks run only a very short period of time (i.e., tens or hundreds of cycles), they require fine-grained timing measurements with very little tolerance in variation. In this work, we use the high-precision and low-overhead user-level *rdtsc* and *rdtscp* [Intel64IA32Manual 2014; RDTSC 2014] instructions to measure the elapsed time. More importantly, we have to eliminate most of the unintended perturbations, such as a level-1 TLB miss, which may be tolerable in the LETE-based microbenchmarks. We classify those potential interfering factors into three categories: (1) interference caused by *hardware resource constraints* such as cache pollution, unintended warmed-up cache effects, unintended TLB misses, and triggering of unintended hardware prefetching and unfinished nonblocking cache requests, (2) interference caused by *operating system effects* such as page allocation, context switching, and multithread interleaving and scheduling, and (3) interference caused by *unintended compiler optimizations*. By identifying all of those interfering factors, we further propose guidelines to control them precisely and minimize all possible perturbations.

To demonstrate how we can measure various memory microarchitectural features using our SETE-based microbenchmarks, we use the newly released Xeon Phi many-core processor (with in-order cores) as an example, where most data are undocumented and thus provides additional benefit to the community. To demonstrate the portability and validate the correctness of such a methodology, we also apply the methodology to the well-documented Intel Sandy Bridge multicore processor (with out-of-order cores), where most data are available and can be validated. Moreover, we open source our microbenchmarks for further validation by the community. To further illustrate the usefulness of our measured data, we present a case study on Xeon Phi and Sandy Bridge, where data are prefetched to the cache memory hierarchy in stages based on the measured resource availability (e.g., number of MSHRs and the hardware prefetching support). We find that our proposed multistage coordinated prefetching algorithm can achieve 1.3X and 1.08X performance speedup on Xeon Phi and Sandy Bridge, respectively, compared to the state-of-the-art Intel ICC [ICC 2014] compiler. We believe that these measurements also provide useful insights into memory optimization, analysis, and modeling of such multicore and many-core architectures.

In summary, this work makes the following contributions:

(1) A SETE-based microbenchmarking methodology is proposed to study the microarchitectural details of memory systems (cache memory in particular) on recent multi- and many-core processors. It uses high-precision low-overhead timing instructions such as *rdtsc* and *rdtscp* on the Intel x86. More importantly, it gives a *comprehensive* analysis of interfering factors that could affect the intended microbenchmark behavior. A set of design guidelines is provided to precisely control and mitigate those interfering factors.

(2) A set of open source SETE-based microbenchmarks is developed.[2] It can be used to study a comprehensive list of memory microarchitectural details on both out-of-order and in-order multi-/many-core processors. Those details include not only traditional cache/memory access latency and off-chip bandwidth but also many software and hardware prefetching related parameters not attempted in the past studies.
(3) A case study is conducted to show that some of the insights on effective software and hardware prefetching uncovered in our measurements can achieve significant performance improvement.

The rest of the article is organized as follows. In Section 2, we discuss related works and their limitations, which motivate our work. Section 3 presents a comprehensive analysis on the interfering factors in our SETE-based microbenchmarking. A list of general guidelines is then proposed to control those interfering factors in the design of our microbenchmarks. Section 4 briefly describes the Intel Xeon Phi many-core architecture and its software programming environment, which we use as a platform to prototype our methodology. In Section 5, we present specific issues related to microbenchmark design and our measured results on Xeon Phi. It also provides several useful insights from the measured results. In Section 6, we apply our microbenchmarks on Sandy Bridge to demonstrate their portability and validate their correctness. Section 7 presents a case study of multistage coordinated prefetching using our measured data to demonstrate their usefulness. Finally, Section 8 concludes the article.

2. RELATED WORK AND MOTIVATION

Using microbenchmarks is a common technique to uncover microarchitectural details of a target processor. In this section, we introduce some related works and identify their limitations, which motivate our SETE-based micro-benchmarking methodology. We also compare our work against performance analysis techniques using hardware performance counters in Section 2.1.

Traditional microbenchmarks, such as BenchIT [Juckeland et al. 2004], LMbench [LMbench 2014], and STREAM [McCalpin 2014], use a LETE-based methodology to measure the cache/memory latency and bandwidth. They use a system call to a timer to measure the time, which usually requires thousands of cycles overhead. Hence, those techniques do not control and mitigate interfering factors at a precision level required in our methodology. Their measurements are often not stable and can have variations in hundreds or thousands of cycles. To tolerate such variations, they run a single event a large number of times, which can take tens of thousands of cycles, and then take their average since the latency of each event is the same. A recent study on the Intel Xeon Phi many-core processor [Fang et al. 2013] used such LETE-based methodology to measure cache/memory latency and bandwidth parameters. In [Peng et al. 2008], it used a similar methodology to compare the memory performance of dual-core processors. It used a ping-pong scheme to measure the latency of cache-to-cache transfers. LETE-based microbenchmarks [Juckeland et al. 2004; LMbench 2014] are also used to measure the cache sizes, cache set associativity, TLB sizes, and access latencies.

To reduce the timing variation, in [Molka et al. 2009], it used the high-precision low-overhead *rdtsc* instruction to measure cache/memory latency and bandwidth on Intel Nehalem processors. It takes only a few cycles to read the timer. Similarly, in [Babka and Tuma 2009], it also used the *rdtsc* instruction to investigate TLB and cache associativity on Intel Core and AMD Opteron processors. In [Paoloni 2014], it

[2]Our microbenchmarks can be downloaded from https://sites.google.com/site/fangzhenman/home/ubench.20140502.tar.gz.

further used the similar instruction *rdtscp* to reduce the timing variation in Intel out-of-order processors, which has just around 56 cycles overhead and 4-cycle variation.[3] They can reduce timing variation of the measured events due to the low-overhead timer accesses. However, they still needed a LETE-based methodology because they needed to aggregate the measured events to offset the effect of out-of-order execution and, more importantly, the variation caused by other interfering factors listed in Section 3.[4]

There are two challenges in the SETE-based microbenchmarking methodology. First, most SETEs cannot be aggregated and then averaged when measuring prefetching-related parameters because the latency of the events will change. For example, to identify the triggering conditions of hardware prefetching in cache memories, we need to know precisely whether each specific memory access is an L1/L2 cache hit or miss. LETE-based microbenchmarks can only tell the average latency from aggregated measurements. They cannot be used to measure *the exact latency of each specific access*. As another example, when we measure the size of MSHRs, we expect to see a latency spike when all MSHR entries are filled. It occurs after only tens or hundreds of cycles, as most MSHRs are quite small. Aggregating such measurements will obscure such *small latency spikes*. Second, many interfering factors that may be tolerable in LETE-based microbenchmarks can no longer be tolerated in SETE-based microbenchmarks because of their short elapsed time in tens or hundreds of cycles. For example, a TLB miss will significantly perturb the latency of a single cache access. To address those challenges, we propose a SETE-based methodology in Section 3 to measure the microarchitectural details of the memory system in multi- and many-core processors.

Similar SETE-based methodologies have been used to measure GPU performance in [Wong et al. 2010] and [Volkov and Demmel 2008], including the cache size, set associativity, instruction latency, and control flow behavior. However, it was not intended to study the data prefetching aspect of the cache memory, which is very important in most memory latency hiding strategies.

2.1. Using Hardware Performance Counters

To enable efficient low-level performance analysis and tuning, commodity processors usually provide a set of hardware performance counters for users to get some statistical performance data such as cycle breakdown, cache miss rate, and bandwidth.

There are mainly two types of hardware performance counter working mechanisms: event-based sampling[5] (EBS, widely used in Intel machines) [Levinthal 2014] and instruction-based sampling (IBS, widely used in AMD machines) [Drongowski 2014]. In EBS, it configures a hardware performance counter to monitor an event (e.g., L2 cache miss) and interrupts every Nth time the event happens. When the interrupt occurs, the program counter (PC) is reported. However, in out-of-order execution, the reported PC may be quite far (tens of instructions) from the actual instruction that triggers the interrupt. To tackle this problem, precise event-based sampling (PEBS) is further introduced (for only a small subset of events) to guarantee that the reported PC is within one instruction of the actual instruction and report the entire architectural state. On the other hand, IBS is designed to associate with an instruction instead of an event. In IBS, every N cycles, it tags a random instruction and records useful events caused by the instruction as it proceeds through the pipeline.

As a result, hardware performance counters can report how frequent an event (e.g., L2 cache miss) happens, identify instructions that most frequently cause a specific

[3]We will discuss the limitation of *rdtscp* and how to overcome it in Section 3.1.
[4]Although [Molka et al. 2009] did point out some interfering factors that could affect the intended microbenchmark behavior, it was not comprehensive enough.
[5]Here, we treat time-based sampling as a special case of EBS, where the event is time.

event (EBS/PEBS), or identify the most frequent events that an instruction causes (IBS). Based on this information, researchers can further compute other useful statistical profiles of a running program. For example, in [Levinthal 2014], Intel demonstrated how to estimate cycle breakdown using hardware performance counters. In [Eyerman et al. 2006, 2011], researchers used hardware performance counters to estimate the cycle per instruction (CPI) breakdown (in different miss events such as cache and TLB miss) of a program and proposed an interval analysis model to further reduce event overlap effects in out-of-order processors. In [Fields et al. 2003, 2004], they collected miss event information within hotspots using specialized hardware performance counters and modeled the interaction cost between multiple instructions to enable bottleneck analysis. In [Ferdman et al. 2012], it used hardware performance counters to characterize various performance metrics such as CPI, cache miss rate, and bandwidth consumption for large-scale workloads.

However, both EBS/PEBS and IBS have their limitations. They use sampling-based mechanisms to reduce the high overhead of using hardware performance counters, and the sampling rate is usually thousands or tens of thousands of instructions. This works fine with statistical measurement but makes it inaccurate to measure the behavior of a single specific event. In [Demme and Sethumadhavan 2011], it proposed a precise and lightweight technique to use on-chip performance counters. However, it is different from our work and is more like an extension to *rdtsc* and *rdtscp* instructions. It makes reading other on-chip performance counters more lightweight yet precise. But it does not consider applying it to measure the memory microarchitectural details or analyze interfering factors to the microbenchmarks.

3. METHODOLOGY FOR MEASURING MEMORY MICROARCHITECTURE

In this section, we present our microbenchmarking methodology using SETE. We first describe how we measure the time using high-precision, low-overhead, user-level *rdtsc* and *rdtscp* instructions with our precision enhancement on in-order and out-of-order processors. We then analyze the crucial factors that could interfere with the intended microbenchmark behavior such as hardware resource constraints, operating system effects, and compiler optimizations. We follow it by proposing some general design guidelines to mitigate those effects. We also uncover some interfering factors unique to SETE-based microbenchmarks and propose techniques to deal with them. For example, we need to ensure that an out-of-order instruction or a nonblocking cache request has been completed before a measurement. Each thread also has to execute its concurrent code region at least 0.1ms (mainly to mitigate thread creating overhead) to ensure that all thread execution is indeed overlapped in a multithreaded environment. Those specific design issues to each measurement are discussed in Section 5.

3.1. Measuring Short Elapsed-Time Events

Many-core architectures such as Xeon Phi use in-order cores to simplify core design and to save power. They usually provide a very stable *rdtsc* instruction, which costs only a few cycles of overhead (six cycles on Xeon Phi). On these in-order cores, we can get the elapsed time by comparing the end and the beginning time of an event, and then subtract the *rdtsc* overhead.

Other multicore processors such as Sandy Bridge use out-of-order cores to boost their performance. Measuring event elapsed time on out-of-order processors is much more challenging. Due to the nature of out-of-order execution, the leading *rdtsc* instruction might be executed ahead of some irrelevant instructions before it; similarly, the trailing *rdtsc* instruction might also be executed ahead of some target instructions. This could lead to a variation of up to hundreds of cycles. To ensure that irrelevant instructions before the leading *rdtsc* are not included, and that all target instructions before the trailing *rdtsc* are included, we need to insert a serializing instruction such as *cpuid* [Intel64IA32Manual 2014; RDTSC 2014] right before each *rdtsc*. Unfortunately, such a serializing instruction could incur an overhead of hundreds of cycles and makes the measurement unstable.

To overcome this problem, we use the *rdtscp* instruction to read the end time instead. The *rdtscp* instruction does the serialization first to ensure that all instructions before *rdtscp* have been completed and then reads the timestamp counter. However, it does not guarantee that instructions after *rdtscp* are not executed before the time is read. To guarantee this, we use a *cpuid* instruction right after *rdtscp* to block the instructions after it from being executed before the end time is read. Compared to *rdtsc*, *rdtscp* has a much higher overhead of around 56 cycles with a 4-cycle variation on processors such as Sandy Bridge [Paoloni 2014].

Although [Paoloni 2014] has already greatly reduced the *rdtscp* overhead and variation, there is still one problem. Some target instructions might overlap their execution with the serializing part (around 50 cycles) of *rdtscp*—that is, the time for those overlapped target instructions cannot be measured. To overcome this problem, we fill some data-dependent instructions after the target event to mitigate this effect—in other words, we use those filled data-dependent instructions to overlap with the serializing part of *rdtscp* so that we can more accurately measure the time of the entire target event.

In summary, to measure the time on out-of-order cores, we use *rdtsc* to read the beginning time and put a serializing instruction *cpuid* right before it. Similarly, we use *rdtscp* to read the end time and put a serializing instruction *cpuid* right after it. Moreover, we fill some data-dependent instructions after the target event to overlap the serialization latency of *rdtscp*. Finally, we calculate the elapsed time by comparing the end and the beginning time of the event, then subtract the overhead of *rdtsc* and *rdtscp*. We find that this technique mostly eliminates the interference caused by out-of-order execution and can produce quite stable timing results for SETEs.

The hardware timestamp counter on multicore and many-core processors is per-core based. It cannot measure the global time for all on-chip cores (i.e., the system-wide time). Fortunately, the only measurement that needs a global timer is for off-chip bandwidth, which can be measured using a traditional LETE-based approach.

3.2. Interfering Factors and General Design Guidelines

There are many other factors that can interfere with the measurements, which makes it quite challenging. In this section, we present the first comprehensive analysis of those interfering factors and classify them into three categories: (1) hardware resource constraints, (2) operating system effects, and (3) compiler optimizations. We also propose techniques to control or mitigate them.

3.2.1. Hardware Resource Constraints. All hardware resources have limited capacities. They include caches, TLBs, and off-chip bandwidth. They could affect the intended microbenchmark behavior if left unchecked. There are other constraints as well. For example, when measuring software prefetching parameters, hardware prefetchers could interfere with their measurements because hardware prefetchers on machines such as Xeon Phi are always on (Intel does not disclose how to turn them off on the Xeon Phi). We use the following techniques to control and avoid those unintended effects.

(1) *Warm up instruction cache and TLB.* In SETE-based measurements, instruction cache misses and TLB misses can cause the measurements to vary significantly in different runs. To reduce such effects and get more stable time, we repeat the measured code regions five times in each run and use the *last* measured time of the five as our measurement time, as suggested in [RDTSC 2014]. This allows the

instruction cache and TLB to be fully warmed up before the intended measurement is taken. In addition, we take the average of three such runs to further reduce possible time variations. We also check their standard deviations (all within 1%) to ensure that the measurement of each run is stable enough to be useful.

(2) *Flush unintended warmed-up data cache lines.* As we repeat the same measurements five times in each run, the cache lines could have been unintentionally warmed up. For example, when we try to measure L1 cache miss penalty, the intended cache line may have already been brought into the L1 cache by earlier execution. To avoid using warmed-up cache lines, we need to explicitly flush the cache each time before taking the measurement.

(3) *Avoid data cache pollution.* Cache pollution could replace the cache lines of interests during a measurement and cause unintended cache misses. To avoid cache pollution, we need to use different cache lines in different sets during a measurement.

(4) *Avoid data TLB misses.* Data TLB misses will incur additional latency and affect the measured time. To avoid unintended TLB misses, we use as few pages as possible to allow their page table entries to be held in the level-1 TLB. We also include a warm-up phase in each run to touch all required pages to ensure that no page fault occurred during the run and that the TLB is also sufficiently warmed up.

(5) *Avoid approaching bandwidth limit.* The memory latency will increase significantly when approaching the bandwidth limit. In all cases except bandwidth measurements, we only use a small number of memory operations. This is guaranteed to keep them way below the bandwidth limit.

(6) *Avoid triggering hardware prefetchers during software prefetching measurements.* To distinguish software and hardware prefetching effects, we need to avoid triggering hardware prefetchers when measuring software prefetching parameters. On Sandy Bridge, we can turn off all hardware prefetchers in the BIOS. However, the Intel data sheets do not disclose how to turn them off on Xeon Phi. To avoid triggering hardware prefetchers, we access random cache lines in all software prefetching measurements, which can prevent hardware prefetchers from detecting any streaming access pattern to trigger them. Interestingly, it was found later that software prefetching will not trigger hardware prefetchers on Xeon Phi—that is, hardware prefetchers will not affect software prefetching.

(7) *Ensure that a nonblocking cache request is completed.* On out-of-order processors such as Sandy Bridge, all cache requests are nonblocking. Even on in-order processors such as Xeon Phi, software and hardware prefetching, as well as cache flushing instructions, are nonblocking. To ensure that those nonblocking cache requests are completed before a measurement, we use the serializing instruction *cpuid* to ensure that all of the instructions in the pipeline have been completed before the target instructions, as described earlier.

3.2.2. Operating System Effects. Another type of interferences comes from the operating system. Paging, context switching, multithread interleaving, and scheduling could affect the intended microbenchmark behavior.

(1) *Ensure physical page allocation.* Considering that data caches use physical addresses, it is important that physical pages are allocated when they are accessed. The operating system often uses copy-on-write mechanism to optimize the physical page management, so we need to initialize (i.e., *write*) the pages before we *access* them to guarantee that physical pages have been allocated before we run the measurements.

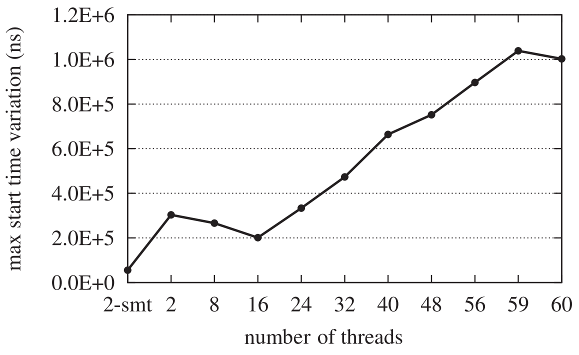


Fig. 1. Maximum variations of the thread start time.

(2) *No context switching.* When measuring SETEs, context switching is usually not an issue because of the short elapsed time.
(3) *Avoid thread migration.* The operating system might migrate a thread to a different core with a cold cache and produce unexpected behavior. To avoid such thread migration during a run, we pin each thread to a specific core. We also pin each thread to a specified hardware thread context when we measure the SMT effects.
(4) *Avoid thread synchronization.* Thread synchronization could incur significant overhead for SETE-based measurements. In our microbenchmarks, we avoid any thread synchronization in the measured code regions.
(5) *Ensure concurrent multithreaded execution.* In multithreaded microbenchmarks, we have to guarantee that when we take the measurements, all threads are actually running concurrently. For example, Figure 1 shows the maximum variation of the *start time* of each thread when a multithreaded microbenchmark is being run on Xeon Phi. Each thread is bound to a different core (except for the *2-smt* case, where threads are bound to two hardware threads on the same core). Figure 1 shows that each thread has to execute at least 0.1ms in the concurrent code regions to guarantee that all threads are actively executing concurrently. This variation is mainly due to the thread creating overhead and is system specific.

    This variation can make SETE-based multithreaded microbenchmarking quite challenging when we want to guarantee that two threads actually overlap in their execution. Fortunately, memory bandwidth measurement is the only case that requires concurrent multithreaded execution, and we measure the bandwidth using LETEs. In all other multithreaded measurements, we only care about the time of each individual thread during the measurements. We keep the first thread running indefinitely, then start the second thread to ensure that these two threads are overlapped. We then take the SETE-based measurements in the second thread.

3.2.3. Compiler Optimizations. Finally, we should avoid unintended compiler optimizations. The compiler may optimize the code in an unexpected way. To avoid compiler optimizations on the microbenchmarks, we use assembly code to implement the measured code regions. We generate other parts of the program by compiling the C code to assembly code and check the assembly codes to make sure that they are what we need.

4. AN OVERVIEW OF INTEL XEON PHI MANY-CORE ARCHITECTURE

In this section, we give a brief overview of the architecture and software programming environment of the newly released Intel Xeon Phi (codename Knights Corner) many-core processor, which we use as an example to demonstrate how to determine various (many undocumented) memory microarchitectural features.

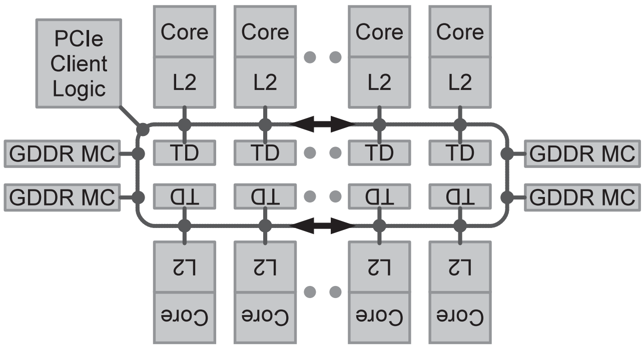


Fig. 2. Overview of the Intel Xeon Phi (codename Knights Corner) many-core architecture.

4.1. Intel Xeon Phi Architecture

Figure 2 shows an overview of Intel Xeon Phi architecture [Chrysos 2012]. It is connected to the host processor (e.g., Intel Xeon processor) through a PCI Express (PCIe) bus. To support high parallelism, the Intel Xeon Phi processor provides up to 61 in-order cores on a single chip. The core used in this study has a clock rate of 1GHz, and Turbo Boost technology [XeonPhiManual 2014] is not supported. Each core has a private L1 cache (32KB) and a private L2 cache (512KB). They are *inclusive*—that is, all data in L1 also have a copy in L2. Each L2 cache has a streaming hardware prefetcher. Each core also has a two-level data TLB that supports two page sizes: 4KB and 2MB. There is an on-chip bidirectional ring that connects the L2 caches, memory controllers (MCs), and distributed tag directories (TDs). The distributed TDs maintain global coherence states for each cache line. Table I shows the basic architectural parameters available in Intel data sheets [XeonPhiManual 2014] for our preproduction engineering sample of the Xeon Phi processor donated by Intel.

4.2. Software Programming Environment

Being an x86-64 many-core processor running Linux, Xeon Phi offers full capability of using the same software tools, programming languages, and programming models as in other general-purpose Intel Xeon processors [XeonPhiManual 2014]. There are two major modes to run an application on Xeon Phi: (1) *offload mode*, in which an application runs on the host and offloads selected parts of the application to Xeon Phi, and (2) *native mode*, in which an application runs on Xeon Phi natively and independently. It can communicate with the host processor or other coprocessors. In this work, we run all microbenchmarks on Xeon Phi in *native mode*. We use Linux *pthreads* to implement our multithreaded microbenchmarks. Xeon Phi's Linux running kernel is version 2.6.38.8. The Linux kernel uses 4KB page size as default, and we use 4KB page size throughout this article unless otherwise specified. To allocate physical pages in 2MB size, we use an *mmap* system call to map a local file onto 2MB pages. To avoid unintended compiler optimizations, Intel ICC 13.0.1 compiler [ICC 2014] with the -O0 option is used to compile the microbenchmarks.

5. MICROBENCHMARK IMPLEMENTATION AND EXPERIMENTAL RESULTS ON XEON PHI

In this section, we show how we design microbenchmark programs to measure key memory microarchitectural details using Xeon Phi as an example, since many of the parameters on Xeon Phi are undocumented in Intel data sheets. They include not only traditional cache latencies and off-chip bandwidth but also microarchitectural

Table I. Basic Parameters of Xeon Phi Preproduction Engineering Sample Available in Intel Data Sheets [XeonPhiManual 2014]

In-order core parameters	
number of cores	60
number of HW threads/core	4
core frequency	always 1GHz (no Turbo Boost)
Cache hierarchy parameters (per core)	
L1 instruction cache size	32KB
L1 data cache size	32KB
L2 unified cache size	512KB
L1/L2 inclusive	yes
line size of L1/L2	64B
associativity of L1/L2	8-way
L2 HW prefetcher	1 streaming HW prefetcher
TLB hierarchy parameters (per core)	
associativity in all cases	4
4KB small page size	
level-1 instruction TLB	32 entries, maps 128KB memory
level-1 data TLB	64 entries, maps 256KB memory
level-2 unified TLB	64 entries, maps 128MB memory
2MB huge page size	
level-1 instruction TLB	not supported
level-1 data TLB	8 entries, maps 16MB memory
level-2 unified TLB	64 entries, maps 128MB memory
GDDR5 memory parameters	
memory size	4GB
peak bandwidth	153.6GB/s

details related to software and hardware prefetching. We also present some insights into software and hardware prefetching using our measured data. Considering that bandwidth measurements use LETEs, we will present them last in Section 5.4. Finally, we summarize all of our measurements on Xeon Phi in Table V.

5.1. Latency Measurements

We measure the latency at different levels of cache hierarchy, on-chip ring interconnect, and TLBs. Moreover, we discuss the effect of blocking and nonblocking caches.

5.1.1. Latency at Different Levels of Cache Hierarchy. We stage an L1 (or L2) cache hit by first prefetching the cache line into the L1 (or L2) cache, then measure the latency of reading the same cache line. For DRAM memory access latency, we directly measure the latency of reading a line from memory (i.e., an L2 cache miss).

We first measure the hit latency of a single thread. The L1 cache hit latency is 1 or 2 cycles. The variation of 1 cycle is caused by the fact that 2 instructions can be issued *every other cycle* for each hardware thread on a Xeon Phi core. Memory access latency varies in different runs and usually falls between 318 and 346 cycles. These measurements agree with those provided in the Intel data sheets [XeonPhiManual 2014]. Our measured L2 cache hit latency is 22 or 23 cycles, which is different from the 11 cycles listed in the Intel data sheets. Another technical report showed independently that L2 cache hit latency was around 22 cycles in their measurements [Fang et al. 2013]. Moreover, we verify the results using both *scalar* and *vector* loads, and the hit latency remains the same.

We then measure the hit latency when SMT is turned on. We find that the hit latency increases about 10 cycles due to the thread interleaving in SMT. Hence, in the

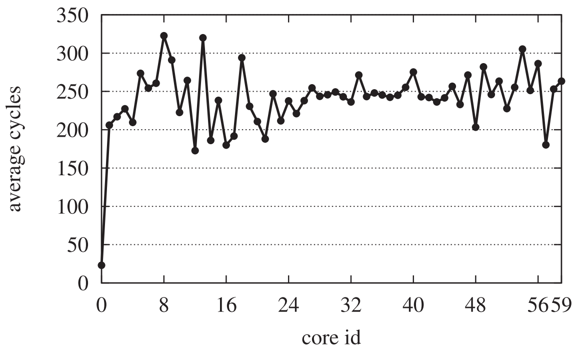


Fig. 3. Latency of reading data from another core's L2 cache.

Table II. Latency (in Clock Cycles) of a Level-1 or Level-2 TLB Miss and a Page Fault

cycles	level-1 TLB miss	level-2 TLB miss	page fault
4KB page	29	91	2,261
2MB page	10	86	2,898,353

rest of article, we will not measure latency using SMT unless we need to study whether hardware threads will cause contention at some shared resources.

5.1.2. Latency at On-Chip Ring Interconnect. According to Intel data sheets, Xeon Phi uses an on-chip ring interconnect to maintain its MESI-like cache coherence protocol. We measure the latency on the ring interconnect by first prefetching a cache line to core i's L2 cache, then measure the latency of reading the same cache line from core 0.

Figure 3 shows the latency of reading core i's L2 cache on core 0. Although the latency varies when reading from different cores, the latency does not increase linearly with the increase of the distance between two cores on the ring. This is due to the "evenly" distributed tag directory design and the ring-based cache coherence protocol. The average latency is about 243 cycles, and the standard deviation is 32 cycles.

A useful insight here is that as the average access latency from a remote L2 is less than the off-chip memory latency (about 25% faster), we could use other idle cores to perform data prefetching for hiding more memory latency if its own L2 data prefetchers are fully subscribed.

5.1.3. TLB Miss Penalty. In this section, we measure the penalty of level-1 TLB miss, level-2 TLB miss, and page fault in both 4KB and 2MB page sizes. To instigate a level-1 or level-2 TLB miss, we use twice the number of pages that a level-1 or level-2 TLB can hold. We touch all of those pages so that later accesses to earlier pages will cause a level-1 or level-2 TLB miss. To trigger a page fault, we simply do not initialize the page before accessing it. Since page fault penalty is much larger than a cache or TLB miss, the results are quite stable.

As Table II shows, the latency of a level-1 TLB miss in accessing a 2MB page (10 cycles) is much smaller than that in accessing a 4KB page (29 cycles). This is mainly because for 4KB pages, a level-2 TLB caches only enough page directory information that requires another level of table lookup, whereas in a 2MB page, a level-2 TLB caches the entire physical address. The level-2 TLB miss latency in accessing a 2MB page is also slightly better. The page fault incurs a significant penalty, which is around 2,300 cycles if the page is cached in the main memory (in the case of a 4KB

Table III. Software Prefetching Triggering Conditions

type	level-1 TLB miss	level-2 TLB miss
Y/N	Yes	Yes
type	page fault	across page boundary
Y/N	No	Yes

page), and around 3ms if the page is on a local file on Xeon Phi (in the case of a 2MB page).

A useful insight here is that one can use the 2MB page size to reduce the penalty of TLB misses if data accesses are concentrated in certain large memory regions.

5.1.4. Blocking and Nonblocking Cache. We also study the effects of a blocking and a nonblocking cache on regular *loads* and *stores*, and on *software prefetching* instructions. We investigate whether they can overlap their latency with the following independent loads, stores, software prefetching instructions, and long-latency computing instructions such as floating-point multiplications.

The results confirm that a load or store miss could block the entire pipeline in an in-order instruction pipeline during a single-thread execution, whereas they can overlap with instructions from another thread if SMT is turned on. For software prefetching instructions, they are nonblocking and can overlap with other independent instructions. The prefetched data returning to the cache can be out of order—in other words, they are not necessary in the order in which prefetching instructions are issued.[6]

5.2. Software Prefetching

In this section, we look at the triggering conditions of software prefetching, the number of outstanding prefetches supported with and without SMT turned on, and the behavior of prefetching in the *exclusive* mode (i.e., data prefetching for later *store* instructions).

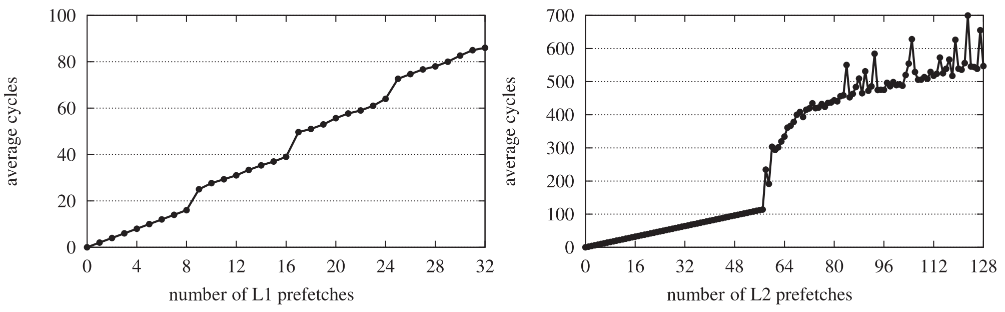
5.2.1. Software Prefetching Triggering Conditions. We first check whether software prefetching will be dropped if (1) it will trigger a level-1/level-2 TLB miss, (2) it will trigger a page fault, or (3) it will cross a page boundary. To verify these cases, we first prefetch a cache line under each of those scenarios. We then measure the latency of reading the same cache line *immediately* after the prefetching instruction is completed. If it is a cache hit, then clearly software prefetching has been triggered.

Table III summarizes our observations. Except for the page fault, software prefetching will be triggered in all other cases. In Section 5.3, we will notice that hardware prefetching has very different triggering conditions—that is, it will not prefetch across a page boundary.

5.2.2. Number of Outstanding Software Prefetches. We also measure the maximum number of outstanding software prefetches supported at L1 and L2 caches—that is, the number of MSHRs available at L1 and L2 caches. We increase the number of issued software prefetches and measure the prefetch latencies. We expect to see a large latency increase when all MSHR entries are filled (so the next prefetch has to wait until one of the previous prefetches is completed and an MSHR entry is freed up). When measuring the number of MSHRs at the L1 cache, we ensure that all prefetched cache lines are already in the L2 cache.

Figure 4(a) shows the results of measuring L1 MSHRs in a single thread. The latency increases sharply when the number of L1 cache prefetches increases from eight to nine,

[6]Due to space constraints, detailed results are not shown in this article but can be obtained using our open source microbenchmarks.



(a) number of L1 MSHRs (b) number of L2 MSHRs

Fig. 4. Latency with different number of outstanding L1 and L2 prefetches, respectively.

and this pattern repeats after every eight L1 cache prefetches. This indicates that the number of L1 MSHRs is eight (i.e., eight outstanding L1 prefetches are supported). We also measure this number when SMT is on and find that *all hardware threads* on the same core share the same eight L1 MSHRs. By eliminating all interfering factors for this SETE as described earlier, the measured results as shown in Figure 4(a) are quite stable—in other words, there is always a jump in latency from eight to nine outstanding software prefetches.

Figure 4(b) shows the results of measuring L2 MSHRs in a single thread. The number of L2 MSHRs is between 56 and 60. Unlike those in L1 cache, this pattern does not repeat because memory latency fluctuates due to the ring interconnect and DRAM latencies. L2 prefetches are completed totally out of order (i.e., prefetched data return in an order different from when they are issued). We also measure it when SMT is on and find that hardware threads on the same core also share the same L2 MSHRs.

A useful insight here is that the number of MSHRs are shared among hardware threads and are quite limited, especially for L1 MSHRs. Prefetching directly from memory to the L1 cache will require much longer prefetch distance and hence more outstanding prefetches that likely will exceed the number of L1 MSHRs. Therefore, it is better to use a *multistage coordinated software prefetching*—that is, first prefetch data from off-chip memory to the L2 cache, then from the L2 cache to the L1 cache with a different prefetch distance. We find that this strategy can significantly improve the cache performance, as demonstrated in Section 7.

5.2.3. Exclusive Prefetching Behavior. Xeon Phi allows a cache line to be prefetched into L1/L2 caches with a *shared* or an *exclusive* coherence state for a later *store* operation. The behavior is quite different when other L1/L2 caches also have a copy of the same cache line due to the invalidation-based coherence protocol. To measure the differences in access latency, we first prefetch the target cache line into a controlled number of remote L2 caches. (We omit the case of prefetching to other cores' L1 caches because it is quite similar to the case of prefetching to other cores' L2 caches.) We then measure the latency of a *store* operation immediately after prefetching the target cache line with a *shared* and an *exclusive* state in its L2 cache, respectively.

As shown in Figure 5, the *store* latency after an *exclusive* prefetching is the same as that of a cache *hit* because the invalidation of the copies in other L2 caches has already been done by the *exclusive* prefetching, whereas after a *shared* prefetching, a *store* operation will incur a large invalidation overhead. However, the invalidation overhead does not increase linearly with the increased number of L2 caches holding the same cache line due to the coherence protocol on the ring interconnect and the distributed tag directories that it uses.

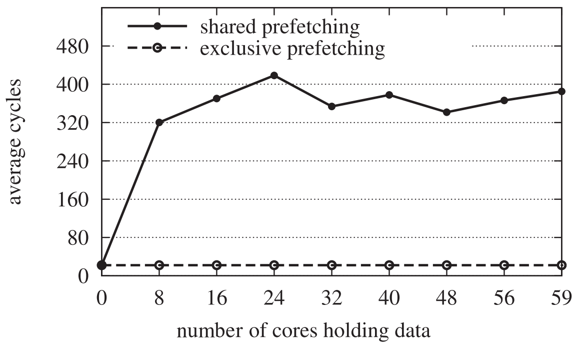


Fig. 5. Latency of a *store* after using a *shared* or an *exclusive* prefetch with different number of other cores holding the same data.

Table IV. Hardware Prefetching Triggering Conditions

access type	triggering condition
regular loads	3 ascending or descending line misses; corner cases need 1 or 2 line misses
regular stores	3 ascending or descending line misses; corner cases need 1 or 2 line misses
SW prefetching	does not trigger HW prefetcher
across page boundary	stop at page boundary

5.3. Hardware Prefetching

Xeon Phi has a hardware prefetcher on the L2 cache but not on the L1 cache. Each L2 hardware prefetcher is a streaming prefetcher [Srinath et al. 2007] that can prefetch 16 different data streams. We look at how a hardware prefetcher is triggered, the number of cache lines a single stream is allowed to prefetch, whether a hardware prefetcher can distinguish memory access streams from different hardware SMT threads, and the hardware prefetcher's behavior in huge (i.e., 2MB) page size.

5.3.1. Hardware Prefetching Triggering Conditions. In this section, we study when and how loads, stores, and software prefetching can trigger hardware prefetching. We also look at the number of misses needed to trigger hardware prefetching, and whether a hardware prefetcher can prefetch across page boundary or not. To do this, we measure the latency of a *load* operation to a target cache line immediately after those conditions have been met, then check to see whether it is an L2 cache hit or not.

Table IV summarizes the observed triggering conditions for hardware prefetching. First, both loads and stores can trigger hardware prefetching. In general, 3 *consecutive* cache misses to 3 *different* cache lines on the *same* page are needed to trigger hardware prefetching. Moreover, the starting addresses of those 3 cache lines must be in an increasing order (positive distance), or a decreasing order (negative distance), to trigger a prefetching in a forward or backward direction, respectively. There are some corner cases that do not need 3 misses: (1) if the first cache miss occurs in the 0th (or 63rd because a page contains 64 cache lines) cache line of a page, then this single miss will trigger the hardware prefetcher in a forward (or backward) direction, as there is only one possible direction for the hardware prefetching stream, and (2) if the first cache miss occurs in the 1st (or 62nd) cache line of a page, then two misses are needed to trigger the hardware prefetcher in a forward (or backward) direction, as these two misses will determine the direction of the hardware prefetching stream.

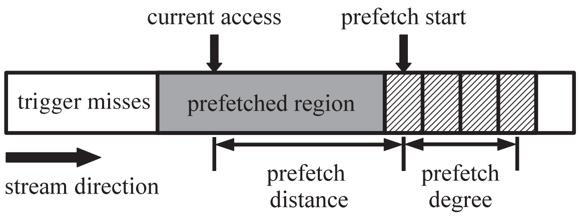


Fig. 6. Overview of a single-stream hardware prefetching behavior.

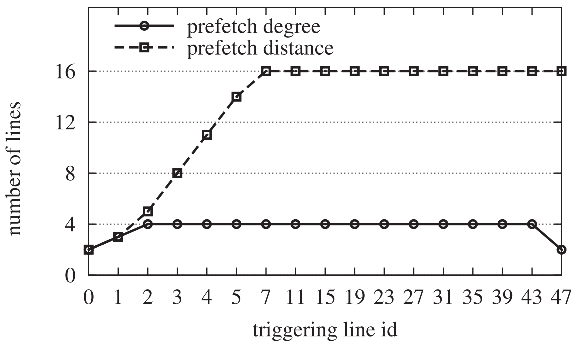


Fig. 7. Prefetch degree and prefetch distance of a single hardware prefetching stream that continuously accesses an entire page.

We also find that software prefetching instructions will *not* trigger hardware prefetchers. Instead, it will disable the hardware prefetcher if it also tries to prefetch the same cache blocks on Xeon Phi. Moreover, we verify that hardware prefetchers will not prefetch across page boundary, because pages that are contiguous in the virtual address space are not necessarily contiguous in the physical address space.

5.3.2. Single-Stream Hardware Prefetching Behavior. Figure 6 presents an overview of a single-stream hardware prefetching behavior, according to the description in Srinath et al. [2007]. After three triggering misses, the hardware prefetcher will prefetch a batch of cache lines (called *prefetch degree*). After the batch is prefetched, whenever there is an access to any of those prefetched cache lines (called *prefetched region*), it will continue to prefetch another batch (of the same *degree*) of cache lines following the end of the last prefetched region (called *prefetch start*), assuming that the distance between the prefetch start and the current access (called *prefetch distance*) is within a certain threshold (called *maximum prefetch distance*).

In this section, we study how hardware prefetching works in a *single* data stream including its prefetch degree, maximum prefetch distance, and how to trigger the hardware prefetcher to prefetch an entire page. We used a technique similar to that in Section 5.3.1 that measures the latency of a load to the target cache line and see whether it is an L2 cache hit or miss.

Figure 7 shows the behavior of a hardware streaming prefetcher that continuously accesses the next cache line in a single page. The x-axis shows the cache line number that the target *load* triggers the hardware prefetcher to start prefetching or to continue prefetching cache lines. It starts from the 0th cache line in a page and ends at the 47th line, at which time all 64 cache lines on the page have either been loaded or being prefetched. Beyond the 47th cache line, the hardware prefetcher will not be triggered, as it has reached the page boundary. The line with round dots shows the prefetch degree at the time a cache line (whose line number is shown the on x-axis) is accessed.

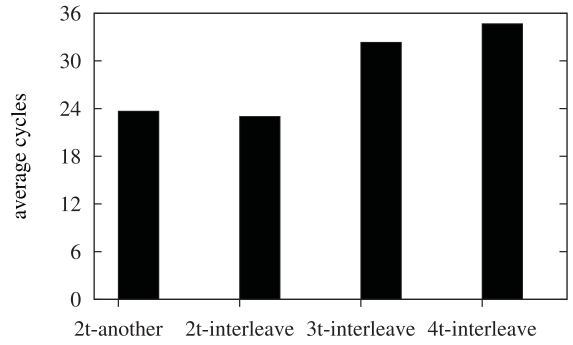


Fig. 8. A single prefetching stream distributed into different hardware SMT threads.

The miss on the 0th cache line triggers the hardware prefetcher to prefetch two lines, whereas the hit on the 1st cache line triggers three additional lines to be prefetched. After that, each cache access to the prefetched region triggers the hardware prefetcher to prefetch four additional lines (the prefetch degree on Xeon Phi is 4). The access of the 47th cache line triggers two additional lines to be prefetched (instead of four) because there are only two lines left on the page before it crosses the page boundary.

The line with square dots shows the prefetch distance at each cache line access, as demonstrated in Figure 6. At first, the prefetch distance continues to increase as the hardware prefetching stream continues to prefetch additional lines. When the prefetch distance goes beyond 16 lines, it stops triggering further prefetching as indicated by the missing numbers in the x-axis of Figure 7 (i.e., 6, 8, 9, 10 . . .), until the prefetch distance is back within 16 lines again.

In summary, the results show that the prefetch degree of the hardware prefetcher is 4 lines, whereas the maximum prefetch distance is 16 lines on Xeon Phi. Moreover, as the access sequence shown in Figure 7, one cache miss to the 0th line and 16 cache line hits to the cache lines with the numbers shown on the x-axis of Figure 7 would trigger the hardware prefetcher to prefetch the entire page of 64 cache lines.

We verify that the hardware prefetcher can support up to 16 such data streams as specified in the Intel data sheets. However, we also find that each data stream has to be in a different page. They cannot be on the same page—in other words, the hardware prefetcher can prefetch cache lines from up to 16 different streams, each on a different page. Each stream will independently track the three consecutive line misses on a page. Those data accesses can be interleaved with misses in other data streams on different pages. Hardware prefetchers can still track them regardless of how they are interleaved.

5.3.3. Hardware Prefetching in SMT. Each core on Xeon Phi can support up to four hardware SMT threads. When SMT is on, we would like to know whether a hardware prefetcher can be triggered by three cache line misses from different hardware threads or not. To do this, we distribute the three triggering load misses among two to four hardware threads on the same core. We synchronize those load misses to make sure they are issued in their specific order that meets the triggering conditions described in Section 5.3.1. We then issue a load to the anticipated prefetched cache line to see if the cache line has been prefetched by the hardware prefetcher or not.

Figure 8 shows the results when those three triggering load misses are distributed among two, three, and four hardware threads, marked as "2t-interleave," "3t-interleave," and "4t-interleave." In the "4t-interleave" case, each of the first three hardware threads issues a triggering load miss, and the fourth hardware thread issues the test load operation. The "2t-another" is the case in which all three triggering load misses are in the same hardware thread, and the test load operation is issued from another hardware thread. From the latency of the test load (shown in the *y*-axis), we can see that the hardware prefetcher can be triggered by three load misses regardless of which hardware threads they come from. The slight increase in the hit latency of the test load (around 12 cycles) when three or four hardware threads are used is due to the nondeterministic interleaving of hardware threads between the two hardware pipelines on the same core that causes the triggering loads to be issued at a later time. It is certainly not as large as 320 cycles for a typical L2 cache miss.

The insight here is that hardware threads (on the same core) accessing the same page could hinder the hardware prefetcher because the nondeterministic interleaving among hardware threads can make data access patterns very random for the hardware prefetcher to detect them. It could be more effective if hardware threads could access different pages so that the hardware prefetcher can detect different data streams on different pages. This may require the intervention of a compiler or a programmer.

5.3.4. Hardware Prefetching in Huge (2MB) Pages. So far, we have presented hardware prefetching results only for the 4KB page size. We also have taken the same measurements using the 2MB page size. We find that the hardware prefetcher treats 2MB pages as if they were 4KB pages. For example, it still cannot prefetch across a 4KB boundary even if it has 2MB in each page. All results that we measured are the same as those in 4KB page size. Notice that software prefetching is ignorant of the page sizes. Hence, it has no effect on software prefetching.

5.4. Off-Chip Memory Bandwidth

As off-chip bandwidth is shared among all cores, we have to consider potential contention among cores and use traditional LETE-based approaches. We have to use the global timer provided on Xeon Phi instead of the local timer on each core. The microbenchmarks are written in C code instead of assembly code. To minimize the branch overhead, we unroll the loop to access two pages in each iteration. We calculate the *effective bandwidth* by dividing the total amount of data accessed from off-chip memory by the measured elapsed time. The peak bandwidth provided in Intel data sheets is 153.6GB/s. It is not possible to accurately measure the peak bandwidth, because the access time will increase nonlinearly when it approaches the peak bandwidth.

Due to space constraints, we only present effective off-chip bandwidth results using *software prefetching*, which can achieve the highest effective bandwidth, as prefetching is nonblocking on Xeon Phi.[7] As Figure 9 shows, when there is one hardware thread per core and each thread is issuing software prefetching (in the *shared* cache coherence state) continuously, the combined bandwidth used by software prefetching increases with the number of cores up to 48 cores. Using more hardware threads per core results in a slight decrease in effective bandwidth, because the scheduling of hardware threads and their contention for MSHRs will increase the total elapsed time. The highest bandwidth achieved with aggregated software prefetching is about 91GB/s, or 59% of the peak bandwidth, which is in line with the Intel data sheets [XeonPhiManual 2014].

5.5. Summary of Results from Xeon Phi

We summarize all of the data we have measured on Xeon Phi in Table V. All undocumented data are in an italic font.

[7]Other effective off-chip bandwidth results using exclusive software prefetching, streaming, and random loads and stores can also be obtained using our open source microbenchmarks.

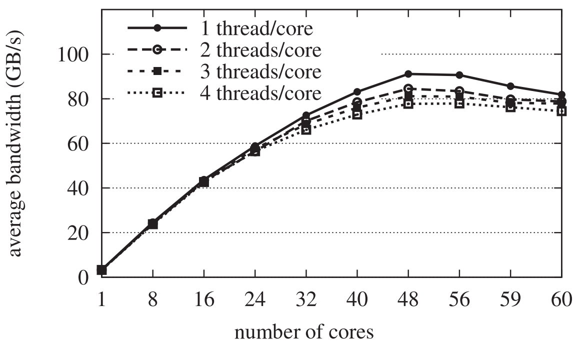


Fig. 9. Effective off-chip bandwidth for shared prefetching with increasing number of cores and hardware SMT threads.

Table V. Summary of Measured Memory Microarchitectural Details on Xeon Phi

Cache/memory hierarchy latency (cycles)	
L1 cache hit	1–2
L2 cache hit	*22–23*
memory access	318–346
on-chip ring interconnect	*average 243, stdev 32*
level-1 TLB miss	*29 on 4KB page, 10 on 2MB page*
level-2 TLB miss	*91 on 4KB page, 86 on 2MB page*
page fault	*2300 in memory, 3M on disk*
Off-chip bandwidth	
91 GB/s, 59% of peak bandwidth	
SW prefetching parameters	
SW prefetch can work	level-1/level-2 TLB miss, across page boundary
SW prefetch cannot work	page fault
number of L1 MSHRs	*8, shared by HW threads*
number of L2 MSHRs	*56–60, shared by HW threads*
exclusive prefetch effect	work better for stores
HW prefetching parameters	
HW prefetch can work	regular load/store
HW prefetch cannot work	*SW prefetch*, across page boundary
detect / prefetch granularity	*cache line granularity*
line misses to trigger	*3 ascending or descending misses; corner cases need 1 or 2 misses*
prefetch degree	4 cache lines
max prefetch distance	*16 cache lines*
single page behavior	*1 miss and 16 hits will trigger the hardware prefetcher to prefetch an entire page with 64 lines*
number of streams	*1 per page*, in total, 16 streams
streams in SMT	*does not distinguish HW threads*
HW prefetch on 2MB page	*treat it as a 4KB page*

Note: Undocumented data are in an *italic* font.

Table VI. Summary of Measured Memory Microarchitectural Details on Sandy Bridge

Cache/memory hierarchy latency (cycles)	
private L1 cache hit	4
private L2 cache hit	12
shared L3 cache hit	26–31
L2/L1 cache in other cores	around 60
memory access	around 200
Off-chip bandwidth	
38 GB/s, 74% of peak bandwidth	
SW prefetching parameters	
number of L1 MSHRs	10, shared by HW threads
number of L2 MSHRs	*10, shared by HW threads*
HW prefetching parameters	
L1 streamer prefetcher	
trigger condition	3 loads to same cache line
prefetch target	next cache line
number of streams	*8*
L1 IP-based stride prefetcher	
trigger condition	same load instruction with stride access
prefetch target	current address plus stride
number of streams	*8*
L2 spatial prefetcher	
trigger condition	1 miss in either line of the pair of two cache lines
prefetch target	the other cache line in the pair
L2 streamer prefetcher, similar to Xeon Phi	
trigger condition	*5 ascending/descending line misses from load/*
	store/prefetch, or 1 miss from L1 HW prefetcher
prefetch degree	2 cache lines
max prefetch distance	20 cache lines
number of streams	32, 1 forward and 1 backward per page

Note: Undocumented data are in an *italic* font.

6. PORTABILITY OF THE SETE-BASED MICROBENCHMARKING METHODOLOGY

Our proposed microbenchmarks can be ported to other Intel or non-Intel processors with in-order or out-of-order cores because similar high-precision, low-overhead *rdtsc* and *rdtscp* instructions are available on most modern microprocessors. Furthermore, our proposed design guidelines to avoid interference caused by hardware limitations (e.g., cache pollution, TLB misses), OS operations (e.g., thread scheduling, paging), and compiler optimizations are also applicable to most modern microprocessors. To demonstrate the portability of our SETE-based microbenchmarking methodology, we try it on an eight-core Intel Sandy Bridge processor (Xeon® E5-2650) [SandyBridge 2014] that has out-of-order cores.

In contrast to Xeon Phi, many of the memory microarchitectural details on Sandy Bridge have already been released in Intel's data sheets [Intel64IA32Manual 2014]. Table VI summarizes the major memory microarchitectural parameters that we measured on Sandy Bridge. We validate that all of the available parameters in the data sheets [Intel64IA32Manual 2014] are the same as those we measured.

Due to space constraints, we mainly present prefetching-related parameters on Sandy Bridge in this section, which can only be measured using our SETE-based microbenchmarks. The methodology is similar to that on Xeon Phi. As well, we need to use *rdtsc* and *rdtscp* instructions patched with data-dependent instructions to measure the time on Sandy Bridge as explained in Section 3.1.

We first measure the software prefetching related parameters. On Sandy Bridge, only prefetching instructions that prefetch data in *shared* coherence state are supported. They can prefetch data to an L2 cache or to an L1 cache. Prefetching to an L3 cache is not supported. First, we verify that the triggering conditions of software prefetching on Sandy Bridge are the same as those on Xeon Phi. Second, we verify that the number of L1 MSHRs is 10, as specified in the data sheets [Intel64IA32Manual 2014]. Moreover, we find that the number of L2 MSHRs is also 10, which is unavailable in the data sheets. This important parameter indicates that the very effective two-stage coordinated software prefetching strategy on Xeon Phi will not work on Sandy Bridge.

We then measure the hardware prefetching related parameters. On Sandy Bridge, there are two hardware prefetchers on each L1 cache: the data cache unit (DCU) streamer prefetcher and the instruction pointer (IP)-based stride prefetcher. There are also two hardware prefetchers on each L2 cache: the spatial prefetcher (also known as the adjacent cache line prefetcher) and the streamer prefetcher. There is no hardware prefetcher on the L3 cache. All hardware prefecthers cannot prefetch across page boundaries.

(1) *L1 streamer prefetcher.* The L1 streamer prefetcher on Sandy Bridge is quite different from the streamer prefetcher on Xeon Phi. We verify that it is triggered by three accesses (only loads) to the same cache line, and it only prefetches the next cache line when triggered. Moreover, we find that it can support at most eight such streams, which is unavailable in the data sheets.
(2) *L1 IP-based stride prefetcher.* The L1 IP-based stride prefetcher keeps track of an individual load instruction to detect regular stride (in bytes). If a regular stride is detected, a prefetch is sent to the address calculated by adding the stride to the current address. The stride can be up to 2KB and can be both forward and backward. We verify all patterns of the L1 IP-based stride prefetchers. Moreover, we find that it can support up to eight such stride prefetching streams at most—that is, it can keep track of eight such load instructions, which is unavailable in the data sheets.
(3) *L2 spatial prefetcher.* The L2 spatial prefetcher is also known as the adjacent cache line prefetcher. It combines two adjacent cache lines (these two lines form a 128-byte aligned chunk) together into a pair. Whenever one cache line in the pair triggers a cache miss, it prefetches the other line in the pair. We have verified this behavior.
(4) *L2 streamer prefetcher.* The L2 streamer prefetcher on Sandy Bridge is similar to the streamer prefetcher on Xeon Phi. However, specific parameters of the prefetcher are different. We first verify the following behavior: (1) the prefetch degree is 2 cache lines, and maximum prefetch distance is 20 lines; (2) it can support 32 streams at most, and each page can support 1 forward and 1 backward stream; and (3) it can be triggered by all misses from loads, stores, software prefetching, and L1 hardware prefetchers. Moreover, we find that to trigger the L2 streamer prefetcher, it needs five misses from loads, stores, or software prefetching. This avoids aggressive prefetching for short streams, and the incurred overhead is negligible due to out-of-order execution and nonblocking caches on Sandy Bridge. However, if the misses come from the L1 hardware prefetchers, only one miss is needed because it already knows that it is not a short stream. These triggering conditions are also unavailable in the data sheets.

These measured hardware prefetcher parameters provide a few important insights. First, the L1 hardware prefetchers are quite conservative—that is, they can only be triggered by loads and can support at most eight streams. If an application has more than eight streams, we need L1 software prefetching to help prefetching more data. Second, the L2 hardware prefetchers (combined with out-of-order execution and nonblocking caches) are very effective most of the time. However, they also have some limitations: (1) they cannot prefetch across a page boundary, and (2) they cannot prefetch beyond 20 cache lines. Thus, we can use software prefetching to trigger the L2 hardware prefetchers if the needed prefetch distance is beyond 20 lines.

7. CASE STUDY: MULTISTAGE COORDINATED DATA PREFETCHING

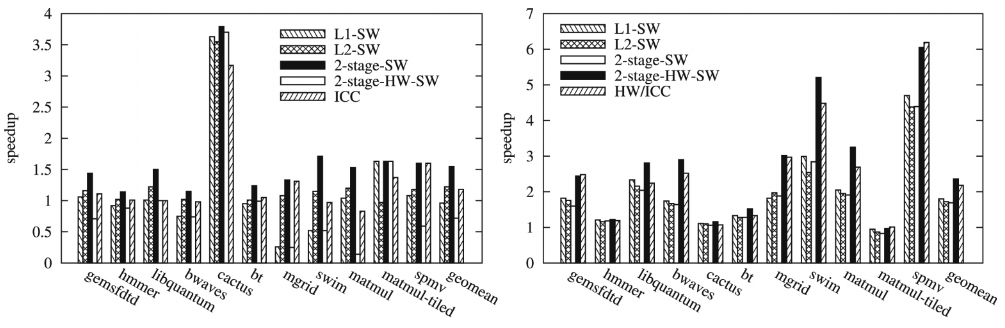
In Section 5 and Section 6, we presented many useful insights into hardware and software prefetching based on our measured data. In this section, we do a case study of multistage coordinated data prefetching on both Xeon Phi and Sandy Bridge to demonstrate the usefulness of our measured data, which we published in [Mehta et al. 2014]. Due to space constraints, we briefly explain our proposed strategies and quote the results from [Mehta et al. 2014].

Multistage coordinated software prefetching on Xeon Phi: L1 software prefetching + L2 software prefetching. On Xeon Phi, hardware prefetchers will be disabled if software prefetching is accessing the same cache blocks as described in Section 5.3.1. Moreover, the number of MSHRs at the L1 cache is only 8, which limits its number of outstanding L1 software prefetches. However, the number of MSHRs at the L2 cache is around 56 to 60, which allows more L2 software prefetches. Therefore, we propose two-stage coordinated software prefetching for Xeon Phi. Data are first brought from memory to the L2 cache with a much larger prefetch distance that requires more outstanding L2 software prefetches (prefetch distance is calculated by dividing the prefetch latency by loop iteration time, as proposed in Mowry et al. [1992]). Data are then brought from the L2 cache to the L1 cache with a much smaller prefetch distance—that is, with fewer outstanding L1 software prefetches. As a result, the resource contention on MSHRs is greatly reduced, and the performance is significantly improved.

Multistage coordinated hardware and software prefetching on Sandy Bridge: L1 software prefetching + L2 hardware prefetching. In contrast, on Sandy Bridge, there are only 10 MSHRs at both the L1 and L2 cache as described in Section 6, and two-stage coordinated software prefetching will not work well. Hence, we have to rely more on hardware prefetchers. According to our measurements in Section 6, L1 hardware prefetchers on Sandy Bridge are quite conservative: they can only prefetch for loads and can support at most eight streams. On the other hand, the L2 hardware prefetchers are quite effective. Based on such observations, one strategy is to use only L2 hardware prefetchers and disable L1 hardware prefetchers. To achieve a better performance, we can use a two-stage coordinated hardware and software prefetching for Sandy Bridge (as opposed to only *software* prefetching on Xeon Phi). Data are first brought from memory to the L2 cache using the effective L2 *hardware* prefetchers. Then, to hide the L1 miss latency that cannot be hidden by out-of-order execution and nonblocking caches [Lee et al. 2012], data are further brought from the L2 cache to the L1 cache using L1 *software* prefetching. Moreover, to overcome the limitation of L2 hardware prefetchers (i.e., it cannot prefetch beyond 20 cache lines and cannot prefetch across a page boundary), we use L1 software prefetching to train L2 hardware prefetchers with a much larger prefetch distance when necessary. As presented in [Mehta et al. 2014], we make a prefetch distance as large as possible. To avoid pollution in the L1 cache, we calculate the L1 software prefetch distance as “1/4 of L1 cache size” divided by “data access size per loop iteration.” As a result, the L2 hardware prefetchers can be trained to prefetch beyond 20 cache lines and page boundaries with a better performance.

7.1. Experimental Environment and Results

We have implemented our multistage coordinated data prefetching algorithm based on ROSE [Quinlan et al. 2001], an open source source-to-source compiler. After generating



(a) prefetching speedup on Xeon Phi (b) prefetching speedup on Sandy Bridge

Fig. 10. Performance speedup of different prefetching strategies for single-thread execution on Xeon Phi (a) and Sandy Bridge (b).

the transformed source code that contains prefetching instructions, we use the Intel ICC compiler (v13) [Krishnaiyer et al. 2013; ICC 2014] with the O3 option to generate the executable. In our experiments, we choose a diverse set of memory-intensive benchmarks from the SEPC CPU2006 [SPECCPU2006 2014] and OMP2012 [SPECOMP2012 2014] benchmark suites with the reference inputs that have high cache miss rates. We also choose two frequently used kernels: dense matrix-matrix multiplication (matmul) and sparse matrix-vector multiplication (spmv).

Figure 10 compares the performance of different prefetching strategies for the chosen benchmarks running with a single thread on Xeon Phi and Sandy Bridge, respectively. In the baseline configuration, we turn off all hardware and software prefetching. Note that on Xeon Phi, the hardware prefetcher cannot be turned off. Except for our proposed two-stage coordinated software prefetching (2-stage-SW) and two-stage coordinated hardware and software prefetching (2-stage-HW-SW), we also evaluate the prefetching performance speedup for (1) L1-SW, using L1 software prefetching only; (2) L2-SW, using L2 software prefetching only; (3) ICC [Krishnaiyer et al. 2013; ICC 2014], using ICC with the -opt-prefetch option enabled; and (4) HW/ICC, with all hardware prefetchers enabled on Sandy Bridge, together with ICC software prefetching.

Figure 10(a) shows the prefetching performance speedup on Xeon Phi. First, our two-stage coordinated software prefetching performs the best. Compared to the hardware prefetcher, it achieves 1.55X speedup on average, whereas compared to ICC, it achieves 1.3X speedup on average because ICC does not coordinate the prefetching between multilevel caches in an effective way. Second, two-stage hardware and software prefetching performs worst because hardware prefetcher and software prefetching cannot coordinate effectively on Xeon Phi. Third, L1 software prefetching alone usually cannot improve the performance due to severe contention on the eight MSHRs. Fourth, L2 software prefetching alone performs better than the L2 hardware prefetcher alone. Finally, for *cactus*, all software prefetching strategies achieve significant speedup compared to the hardware prefetching because it has around 80 streams, whereas the hardware prefetcher only supports 16 streams.

Figure 10(b) shows the prefetching performance speedup on Sandy Bridge. The results are quite different from those on Xeon Phi. Actually, the best prefetching strategy on Xeon Phi becomes the worst on Sandy Bridge and vice versa. First, two-stage coordinated hardware and software prefetching performs the best. Compared to the baseline, it achieves 2.36X speedup on average, whereas compared to HW/ICC, it achieves 1.08X speedup on average due to the coordination. The speedup compared to HW/ICC mainly comes from three categories of benchmarks: (1) benchmarks with significant L1 miss latency that cannot be tolerated by out-of-order execution and nonblocking caches, such as swim; (2) benchmarks that needs large prefetch distance that cannot be prefetched by the L2 hardware prefetcher, such as matmul and bwaves; and (3) benchmarks with both property (1) and property (2), such as libquantum. For other benchmarks, two-stage coordinated hardware and software prefetching achieve comparable performance with hardware prefetchers (HW/ICC) since the software prefetching instruction overhead incurred is negligible. Second, two-stage software prefetching performs worst. Since both the L1 and L2 cache have only 10 MSHRs, two-stage software prefetching does not have any advantage over using L1 or L2 software prefetching alone. Furthermore, two-stage software prefetching incurs more instruction overhead than L1 software prefetching alone. Third, hardware prefetching performs better than pure software prefetching because it does not incur additional instruction overhead and is not limited by MHSR resources. Finally, for *cactus*, all prefetching strategies achieve modest speedup because out-of-order execution and nonblocking caches on Sandy Bridge can tolerate a large portion of the stalls caused by cache misses.

In summary, our multistage coordinated data prefetching case study has demonstrated that using our measured data, we can achieve significant performance improvement.

8. CONCLUSION

In this article, we proposed a microbenchmarking methodology that allows software developers to measure various memory microarchitectural features based on SETEs. We presented a comprehensive analysis of potential interfering factors from hardware resource constraints, operating system effects, and compiler optimizations, which can affect the intended behavior of such microbenchmarks. We further proposed mechanisms to control and mitigate those potential interferences. Using the proposed methodology, we can measure not only those microarchitectural features that can use traditional LETEs, such as cache/memory latency and off-chip bandwidth parameters, but also those that require SETEs, such as software and hardware prefetching related details, which have not been explored before.

Moreover, we demonstrated the effectiveness of our approach by developing a set of open source microbenchmarks for multi- and many-core processors that have either in-order or out-of-order cores. We measured many memory microarchitectural details on Xeon Phi (many of them are unavailable in published data sheets), which are summarized in Table V. We further demonstrated the portability of our approach on the Intel Sandy Bridge multicore processor and measured some undocumented but important memory microarchitectural parameters, which are summarized in Table VI. We also provided some useful insights for effective software and hardware prefetching based on the measured data. A case study that applies those insights on Xeon Phi and Sandy Bridge shows that multistage coordinated data prefetching strategies can significantly improve the application performance.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive feedback that helped us improve the quality of our work. This work was done as a partial fulfillment of the dissertation work by James Greensky at the University of Minnesota at Twin Cities.

REFERENCES

Vlastimil Babka and Petr Tuma. 2009. Investigating cache parameters of x86 family processors. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*. 77–96.

George Chrysos. 2012. Knights Corner, Intel's first many integrated core (MIC) architecture product. In *Proceedings of Hot Chips: A Symposium on High Performance Chips (HotChips'12)*.

John Demme and Simha Sethumadhavan. 2011. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. 353–364.

Paul J. Drongowski. 2014. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. Retrieved November 16, 2014, from http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/AMD_IBS_ paper_EN.pdf.

Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 316–326.

Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2006. A performance counter architecture for computing accurate CPI components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. 175–184.

Stijn Eyerman, Kenneth Hoste, and Lieven Eeckhout. 2011. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'11)*. 216–226.

Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. 2013. *Benchmarking Intel Xeon Phi to Guide Kernel Design*. Technical Report PDS-2013-005. Delft University of Technology, The Netherlands.

Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 37–48.

Brian A. Fields, Rastislav Bodík, Mark D. Hill, and Chris J. Newburn. 2003. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*. 228–239.

Brian A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. 2004. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Code Optimization* 1, 3, 272–304.

ICC. 2014. Intel ICC Compiler. Retrieved November 16, 2014, from https://software.intel.com/en-us/intel-compilers.

Intel64IA32Manual. 2014. Intel 64 and IA-32 Architectures Optimization Reference Manual. Retrieved November 16, 2014, from http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on POWER7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 137–146.

Guido Juckeland, Michael Kluge, Wolfgang E. Nagel, and Stefan Puger. 2004. Performance analysis with BenchIT: Portable, flexible, easy to use. In *Proceedings of the 1st International Conference on the Quantitative Evaluation of Systems*. 320–321.

Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2011. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 393–404.

Rakesh Krishnaiyer, Emre Kultursay, Pankaj Chawla, Serguei Preis, Anatoly Zvezdin, and Hideki Saito. 2013. Compiler-based data prefetching and streaming non-temporal store generation for the Intel Xeon Phi coprocessor. In *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*. 1575–1586.

Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization* 2, 1–2, 29.

David Levinthal. 2014. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performa nce_analysis_guide.pdf.

LMbench. 2014. LMbench: Tools for Performance Analysis. Retrieved November 16, 2014, from http://www.bitmover.com/lmbench/.

John D. McCalpin. 2014. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Retrieved November 16, 2014, from http://www.cs.virginia.edu/stream/.

Sanyam Mehta, Zhenman Fang, Antonia Zhai, and Pen-Chung Yew. 2014. Multi-stage coordinated prefetching for present-day processors. In *Proceedings of the 28th International Conference on Supercomputing (ICS'14)*.

Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. 2009. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 261–270.

Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*. 62–73.

Gabriele Paoloni. 2014. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. Retrieved November 16, 2014, from http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

Lu Peng, Jih-Kwon Peir, Tribuvan K. Prakash, Carl Staelin, Yen-Kuang Chen, and David M. Koppelman. 2008. Memory hierarchy performance measurement of commercial dual-core desktop processors. *Journal of Systems Architecture: The EUROMICRO Journal* 54, 8, 816–828.

Daniel J. Quinlan, Markus Schordan, Bobby Philip, and Markus Kowarschik. 2001. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing (LCPC'01)*. 383–394.

RDTSC. 2014. Using the RDTSC Instruction for Performance Monitoring. Retrieved November 16, 2014, from http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

SandyBridge. 2014. Intel Sandy Bridge Processor. Retrieved November 16, 2014, from http://ark.intel.com/products/64590.

Seung Woo Son, Mahmut Kandemir, Mustafa Karakoy, and Dhruva Chakrabarti. 2009. A compiler-directed data prefetching scheme for chip multiprocessors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. 209–218.

SPECCPU2006. 2014. The SPEC CPU 2006 Benchmark Suite. Retrieved November 16, 2014, from http://www.spec.org/cpu2006/.

SPECOMP2012. 2014. The SPEC OMP 2012 Benchmark Suite. Retrieved November 16, 2014, from http://www.spec.org/omp/.

Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. 63–74.

James Tuck, Luis Ceze, and Josep Torrellas. 2006. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 409–422.

Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'08)*. 31:1–31:11.

Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*. 235–246.

XeonPhiManual. 2014. Intel Xeon Phi Coprocessor System Software Developers Guide. Retrieved November 16, 2014, from http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf.

Received June 2014; revised October 2014; accepted October 2014