Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1335

Optimizing Performance in Highly Utilized Multicores with Intelligent Prefetching

MUNEEB KHAN

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

ISSN 1651-6214
ISBN 978-91-554-9450-6
urn:nbn:se:uu:diva-272095

Dissertation presented at Uppsala University to be publicly examined in ITC/2446, Informationsteknologiskt centrum, Lägerhyddsvägen 2, Uppsala, Monday, 21 March 2016 at 13:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Per Stenström (Department of Computer Science and Engineering, Chalmers University of Technology).

Abstract

Khan, M. 2016. Optimizing Performance in Highly Utilized Multicores with Intelligent Prefetching. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1335. 54 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9450-6.

Modern processors apply sophisticated techniques, such as deep cache hierarchies and hardware prefetching, to increase performance. Such complex hardware structures have helped improve performance in general, however, their full potential is not realized as software often utilizes the memory hierarchy inefficiently. Performance can be improved further by ensuring careful interaction between software and hardware. Performance can typically improve by increasing the cache utilization and by conserving the DRAM bandwidth, i.e., retaining more useful data in the caches and lowering data requests to the DRAM. One way to achieve this is to conserve space across the cache hierarchy and increase opportunity for temporal reuse of cached data. Similarly, conserving the DRAM bandwidth is essential for performance in highly utilized multicores, as it can easily become a critical resource. When multiple cores are active and the per-core share of DRAM bandwidth shrinks, its efficient utilization plays an important role in improving the overall performance. Together the cache hierarchy and the DRAM bandwidth play a significant role in defining the overall performance in multicores.

Based on deep insight from memory behavior modeling of software, this thesis explores five software-only methods to analyze and increase performance in multicores. The underlying philosophy that drives these techniques is to increase cache utilization and conserve DRAM bandwidth by 1) focusing on making data prefetching more accurate, and 2) lowering the miss rate in the cache hierarchy either by preserving useful data longer by cache-bypassing the less useful data or via code size compaction using compiler options. *First*, we show how microarchitecture-independent memory access profiles can be used to analyze the Instruction Cache performance of software. We use this information in a compiler pass to recompile application phases (with large Instruction cache miss rate) for smaller code size in an effort to improve the application Instruction Cache behavior. *Second*, we demonstrate how a resourceefficient software prefetching method can be combined with hardware prefetching to improve performance in multicores when running software that exhibits irregular memory access patterns. *Third*, we show that hardware prefetching on high performance commodity multicores is sub-optimal and demonstrate how a resource-efficient software-only prefetching method can perform better in fully utilized multicores. *Fourth*, we present an adaptive prefetching approach that dynamically combines software and hardware prefetching in a runtime system to improve performance in highly utilized multicores. *Finally*, in the fifth work we develop a method to predict per-core prefetching configurations that deliver near-optimal overall multicore performance. These software techniques enable us to tap greater performance in multicores (up to 50%), without requiring more processing resources.

Keywords: Performance, Optimization, Prefetching, multicore, memory hierarchy

Muneeb Khan, Department of Information Technology, Computer Architecture and Computer Communication, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-554-9450-6
urn:nbn:se:uu:diva-272095 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-272095)

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim
Because it was grassy and wanted wear,
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I,
I took the one less traveled by,
And that has made all the difference

– Robert Frost

Dedicated to my very supportive parents,
my brother, my loving wife
and dearest Fateh and Rehma.

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Muneeb Khan, Andreas Sembrant and Erik Hagersten, "Low overhead Instruction Cache Modeling using Instruction Reuse Profiles". In Proc. International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2012.
I'm the primary author of this paper. Andreas Sembrant provided phase detection software and helped with the case study.

II Muneeb Khan and Erik Hagersten, "Resource Conscious Prefetching for Irregular Applications on Multicores". In Proc. International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (ICSAMOS), 2014.
I'm the primary author of this paper.

III Muneeb Khan, Andreas Sandberg and Erik Hagersten, "A Case for Resource Efficient Prefetching in Multicores". In Proc. International Conference on Parallel Processing (ICPP), 2014.
I'm the primary author of this paper. Andreas Sandberg was involved in discussions and provided some software infrastructure.

IV Muneeb Khan, Michael A. Laurenzano, Jason Mars, Erik Hagersten and David Black-Schaffer, "AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance". In Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT), 2015.
I'm the primary author of this paper. Michael Laurenzano was involved in the discussions and provided the basic **protean code** *infrastructure which I extended further to realize this work.*

V Muneeb Khan, David Black-Schaffer and Erik Hagersten, "Perf-Insight: A Simple, Scalable Approach to Optimal Data Prefetching in Multicores". Technical Report 2015-037, Department of Information Technology, Uppsala University, 2015.
I'm the primary author of this paper.

Reprints were made with permission from the publishers.
Papers have been reformatted but the text is verbatim.

Other publications not included:

- Muneeb Khan, Andreas Sandberg and Erik Hagersten, "A Case for Resource Efficient Prefetching in Multicores". (Long abstract) In Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014.
  I'm the primary author of this paper. Andreas Sandberg was involved in discussions and provided some software infrastructure. This was published as an abstract and presented as a poster.

- Muneeb Khan and Erik Hagersten, "Investigating How Simple Software Optimizations Effect Relative Throughput Scaling on Multicores", Technical Report 2012-010, Department of Information Technology, Uppsala University, 2012.
  I'm the primary author of this paper. Nikos Nikoleris was involved in the discussions.

- Muneeb Khan and Erik Hagersten, "Optimization Study for Multicores", In Proc. Swedish Workshop on Multicore Computing (MCC), 2009.
  I'm the primary author of this paper.

Contents

1. Introduction

Computers are ubiquitous in today's world. They range from handhelds and mobile devices to racked servers in data warehouses. With compute devices found everywhere, multicore processors (or, alternatively, chip multiprocessors) are the most common processing platform today. Multiple cores working together provide an efficient alternative to aggressive single-core processors in terms of processing throughput and energy efficiency.

Each core in modern processors can process data much faster than it is delivered from the main memory, which makes the DRAM a significant bottleneck especially when multiple cores are active. Modern processors make use of caches (smaller, but faster memory units located closer to the processor) to hide memory latency from DRAM and to provide higher bandwidth. To use caches to the fullest, processor architects often design elaborate cache hierarchies with a combination of larger and smaller caches, where the increasingly faster (and smaller) caches sit increasingly closer to the cores. Typically, all cores in a multicore processor share the cache hierarchy (partly) and the interface to the DRAM, the memory controller. Figure 1.1 shows the memory hierarchy in a typical multicore processor such as Intel Sandybridge and AMD Phenom II. In this case all cores have a set of private caches (L1 and L2) and share the last level cache (LLC) and the memory controller. Under such arrangement the LLC and memory bandwidth can easily become major performance bottlenecks. These *shared resources* in multicores play an important role in defining the overall multicore performance. One way to maximize performance in multicores is to optimize the use of the memory hierarchy, especially the shared memory resources. This thesis investigates software-only techniques that optimize the software application's use of the memory hierarchy in multicores, resulting in significant performance improvements.

To improve the use of shared resources and to improve performance in multicores it is important to have insight into the application's memory behavior. There are many ways to profile information about application's memory behavior, ranging from detailed simulation to low-overhead memory system modeling techniques. Low-overhead cache performance modeling techniques can provide the necessary information to enable effective software optimizations. Cache modeling techniques such as StatStack [8] can, for example, provide information about the fraction of data requests that miss at each cache level in the memory hierarchy; this is called the application's *miss-ratio*. Similarly, StatStack can also provide the fraction of all data requests from individual memory instructions in the application that miss at each cache level. The

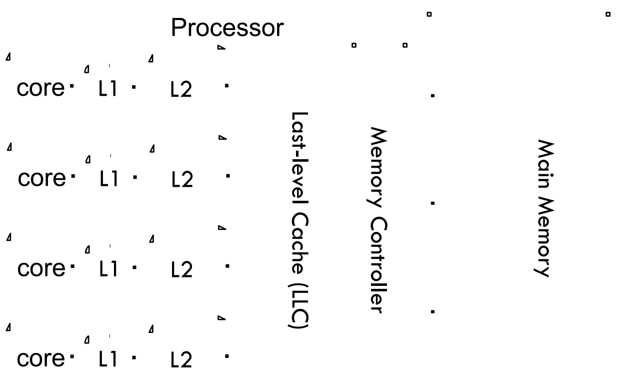


Figure 1.1. Memory hierarchy of a typical multicore processor. Each core has two private caches and shares the last level cache with other cores on the processor. The processor has a memory controller with three memory channels to the main memory.

application-wide average miss ratio behavior helps identify applications that should be targeted for software optimization of memory accesses to improve the use of shared resources. Whereas, the per-instruction information is useful to identify and target individual instructions for optimizing the application's overall memory access behavior.

For a complete picture of an application's memory behavior it is necessary to have insight into how the application's instruction stream is accessed across the memory hierarchy. Paper I facilitates this by using StatStack to model instruction cache performance. The paper presents a instruction-stream specific sampling mechanism that enables capturing instruction-reuse samples with a low-overhead. The instruction-reuse profiles enable StatStack to model instruction cache miss-ratio for any application. We combine instruction cache performance with phase information to classify application phases where the instruction stream experiences high miss ratio in the instruction cache. We use this classification to identify parts of the software codebase that are likely to affect performance (due to high instruction cache miss-ratio) and require optimizations to reduce the code footprint. Code size optimizations applied to the right parts of the codebase can improve the locality of the required application code in the instruction cache and improve performance.

To hide memory latency (and reduce the miss ratio across the cache hierarchy) modern high performance processors are equipped with hardware prefetchers. This hardware utility detects memory access patterns of independent data streams and proactively fetches data that is expected to be used soon. Hardware prefetchers may be located at different levels in the cache hierarchy and fetch data from the next level (further from the core) in the memory hierarchy. Hardware prefetchers in commodity processors are good at detecting and prefetching for regular memory access patterns. However, irregular memory accesses, such as pointer chasing and indirect-indexing, used by many real-world applications remain unhandled. In Paper II we introduce a low-overhead framework that helps identify memory instructions that exhibit irregular access patterns and miss frequently in the cache. These memory instructions are targeted for irregular prefetching in software, while hardware prefetching takes care of prefetching for regular memory accesses. We investigate the benefits of irregular prefetching in software for single-thread performance and throughput performance when several irregular applications are co-run.

While hardware prefetchers can increase single-thread performance significantly, they also tend to increase pressure on the shared resources (LLC capacity and shared memory bandwidth). Aggressive hardware prefetchers are not accurate and fetch significant amount of useless cache lines from the DRAM. This increases useless data requests to the DRAM and wastes the shared resources. While aggressive use of shared resources is usually not an issue when running a single thread, the limited off-chip bandwidth can quickly become a bottleneck when several threads are co-run. This can limit the overall multicore performance. In Paper III we show that a less aggressive software-only prefetching method outperforms hardware prefetching in modern multicores when they are fully utilized. By accurately prefetching the required data, the software-only method conserves the shared resources. As a result of this resource-lean prefetching, shared resources are less contended when all cores are active and performance improves significantly over hardware prefetching.

The opportunity to increase performance using a less aggressive software-only prefetching mechanism, described in Paper III, gives insight into the importance of shared resources in highly utilized multicores. While the software-only prefetching method performs well in general, hardware prefetching can sometimes perform better, making the software-only option alone non-optimal. A better prefetching strategy is to choose the option that performs best in a given execution context. To address this Paper IV describes a runtime method that adapts the data prefetching strategy dynamically. The runtime system detects changes in the execution environment and reacts by monitoring the performance of several prefetching options and adapting the best strategy dynamically in real-time. The five data prefetching strategies explored in this work are a combination of hardware prefetching (on a modern Intel multicore processor) and software prefetching (from Paper III) that are applied uniformly across all cores. The ability to adapt data prefetching at runtime gives life to a robust performance optimization mechanism that improves multicore performance by more than 8% (on average) over hardware prefetching.

The runtime method in Paper IV applies the selected data prefetching option uniformly across all cores to maximize overall performance. However, applying the same policy across all cores is usually not the best way to achieve optimal performance, as some co-running applications may work better with different options. The ability to configure the prefetching strategy on a per-core basis independently exposes more opportunities for improving performance. However, this also results in a large combination of prefetch settings that must be traversed to identify and select the best. For example, 5 different prefetch settings across 4 cores results in 625 permutations (5^4) of unique prefetch configurations. Evaluating performance at runtime for all these prefetch combinations using the greedy approach described in Paper IV is not a feasible solution. We address this problem in Paper V which presents a simple scalable approach that predicts performance across all different prefetch combinations using a handful of combinations whose performance is monitored at runtime. The monitored combinations cover only 5% of the entire set of configurations (625), but are sufficient to give insight into each application's behavior under the 5 individual prefetch options. This information is combined with an iterative approach that, as a first step, estimates the memory bandwidth of the mix under any arbitrary prefetch combination, and then estimates the performance. The approach is used to predict throughput performance across all (625) prefetch combinations, which guides us to identify a prefetch configuration that provides near-optimal performance.

2. Efficient Cache Modeling

The work presented in this thesis makes extensive use of low-overhead cache modeling. *First*, to model instruction cache performance. *Second*, to identify memory-intensive applications amenable to memory access optimizations and the memory instructions that miss frequently in the caches.

StatStack is a fast cache model that estimates miss-ratio – The fraction of memory accesses that miss in a cache of a given size. As example, Figure 2.1 shows the average miss-ratios for the entire run of *mcf* benchmark and of a single load instruction in the benchmark, both modeled using StatStack. Per-instruction miss-ratio curves are used by our analysis to determine the loads that frequently miss in the cache and can benefit from data prefetching.

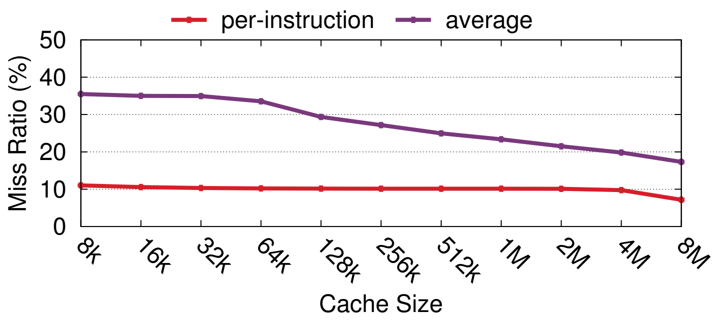


Figure 2.1. miss-ratio Modeling - average miss-ratio of *mcf* and miss-ratio of a frequently executed load - both modeled by StatStack.

2.1 Stack Distance based Modeling

Stack distance [31] forms the basis of several techniques that give insight into application data locality. Stack distance (in terms of caches) is the number of *unique* cache lines accessed between two successive references to the same cache line. Thinking in terms of a stack – a memory reference to a new cache line places the cache line at the top of the stack and pushes all other cache lines on the stack a step deeper. When a cache line X is re-referenced, it is brought back to the top of the stack from a depth d. At this point d is X's stack distance. For a given cache size S, if the stack distance d for a given cache line A is larger than S, the memory instruction that referenced cache line A will miss in the cache. If all stack distances of a cache line A are known,

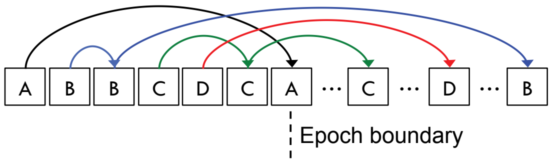


Figure 2.2. Reuse distances in memory access stream. The arcs join successive accesses to same cache line. The stack distance of A is the number of arcs crossing the epoch boundary.

the miss-ratio for any memory instruction that references A can be estimated as the fraction of A's stack distance distribution where $d > S$. The overall application miss-ratio can be estimated as the *weighted* average of all memory instructions in the application. Stack distance based cache modeling assumes a fully-associative cache.

2.2 StatStack – Sparse Data Reuse based Model

StatStack [8] is a statistical cache model that estimates application miss-ratio for data caches (that employ LRU replacement policy) of any given size. In contrast to stack distance, *reuse distance* is a simpler metric that counts all memory references between two references to the same cache line. StatStack uses sampled reuse distance profiles to estimate the stack distance distribution and model the application's cache performance. It is important to note that StatStack only models capacity misses. The input to StatStack is the reuse distance distribution of all memory references of a given application. This can be effectively estimated by sparse sampling of the application's memory reuse distances. Although the distribution is based on sparse sampling of memory references, the resulting reuse distance distribution is representative enough to accurately model miss-ratio when fed to the StatStack model. The StatStack model can be explained with the help of a simple example, consider the memory access sequence in Figure 2.2. This sequence represents a single reuse epoch of cache line A, with several interleaving memory accesses. The arcs connect subsequent accesses to the same cache lines, showing data reuse in this epoch. Here cache line A has a reuse distance of 5 since there are five memory accesses executed between two consecutive accesses to A . The stack distance for A is 3, as there are only three unique cache lines (B, C and D) accessed during A's reuse period.

The stack distance of A is equal to the number of interleaving memory accesses whose reuse distance is greater than their distance to A at the epoch's end. So, if we know all the interleaving memory accesses in A's reuse, we can determine A's stack distance. However, the reuse distance distribution input to StatStack is a representative sample of the application reuse distance and not the entire reuse trace. So to estimate the stack distance of A, StatStack applies

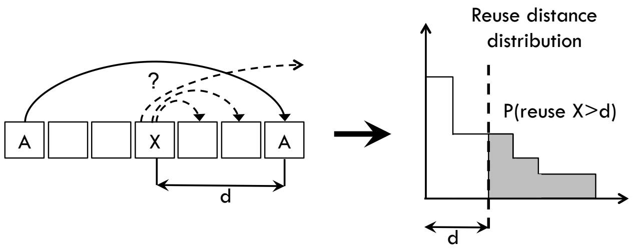


Figure 2.3. Reuse distance distribution is used to calculate the probability of (memory access) X's reuse crossing the epoch boundary.

a probabilistic approach. StatStack finds the probability for each memory access in this reuse epoch to have reuse distance greater than its distance to *A*. For example, in Figure 2.3 if the reuse of *X* is greater than *d* , then this memory access will add to the stack distance of *A*. StatStack approximates the probability of *X*'s reuse to be greater than *d* as the fraction of all memory accesses in the reuse distance distribution whose reuse distance is greater than *d*. This approximation is based on the assumption that for a given execution vicinity, the reuse distribution is uniform. This procedure is repeated to approximate the probability of reuse crossing the epoch boundary for all interleaving memory accesses. Adding these probabilities then gives us the expected stack distance of *A*.

StatStack uses this methodology to estimate the stack distance for all the memory accesses in the sampled reuse distance distribution. These estimated stack distances make up the expected stack distance distribution. This expected stack distance distribution can then be used to estimate the miss-ratio for any given cache size. For a cache size *S*, the fraction of memory references with expected stack distance greater than *S* will be the miss-ratio for that cache size. StatStack can similarly model per-instruction miss-ratios (as shown in Figure 2.1) for a given memory instruction by isolating the stack distances that correspond to the reuse distance samples associated with the memory instruction. Pan et al. have shown that reuse distance distribution can be used to model a wider range of replacement policies besides LRU, such as random replacement, pseudo-LRU and MRU [33]. Those extensions can also be used to model cache behavior. We rely on StatStack for modeling cache performance.

2.3 Data Reuse Sampling

Data reuse samples are used as input to StatStack to model cache behavior for different cache sizes. To enable efficient cache modeling we need to sample data reuse with a low-overhead. Data reuse sampling captures reuse distance profiles across the entire execution of an application. The sampler (developed by Berg and Hagersten) attaches to a process (like a debugger process) and uses performance counter overflow traps to halt the execution of the application process after a number of memory references have executed. Memory references are sampled randomly to avoid sampling bias. After deciding to sample a memory reference, the sampler records the address of the cache line accessed and sets page protection for the entire page, where the cache line resides. The sampler then records the number of memory references executed to date and then continues executing the process. When a memory reference tries to access the cache line being "watched", a page access fault occurs halting the execution of the process. The sampler determines if the page access fault was due to accessing the cache line being sampled. If so, the reuse distance for this cache line is recorded as the difference of memory references executed to date and the number of memory references recorded previously, when setting the watchpoint. The sampler removes the page protection (if no other cache line in the page is being watched) and continues the process. This watchpoint mechanism has the drawback of considerable amount of false-positives. This happens when a cache line other than the one being watched is accessed from the same page, after the watchpoint has been setup for that page. The sampler simply ignores such cases of false-positives and continues the execution of the stopped process. Sembrant et al. show that data reuse samples can be captured over the entire execution of an application with an overhead of less than 20% over native execution using a phase-guided approach.

3. Fast Instruction Cache Modeling

Performance improvement techniques mostly focus on optimizing the data access behavior and largely ignore the role of the instruction cache. Since most applications miss more frequently in the data cache compared to the instruction cache. However, simply comparing the frequency of instruction cache misses and data cache misses can be misleading as applications usually execute several instructions for every data reference. The instruction stream is the source of continuous application execution and latency stemming from stalls in the instruction stream directly impacts the application performance. The more an application misses in the instruction cache the slower it runs. Paper I describes how statistical cache modeling can be extended to model instruction cache performance with low-overhead. We further demonstrate how this information can be used to improve instruction cache performance.

Chapter 2 described how StatStack uses sparse data reuse samples to model the data cache performance. Similarly, to model instruction cache performance, StatStack requires instruction reuse profiles. There are several challenges when it comes to enabling low-overhead sampling of instruction reuse. First, page-protection based sampling (in the instruction stream) is very slow since the most active part of the application code resides in a handful of pages. The control usually remains in the same page after setting watchpoints (Section 2.3) and the resulting false positives slowdown the execution speed to that of single-stepping i.e. stopping execution due to false-positives after each instruction. Second, instructions in x86 processors are not cache line aligned and sometimes an instruction in one cache line can spill into the next. This raises the issue of which cache line(s) to sample. Third, instructions in the x86 instruction set architecture are not the same size.

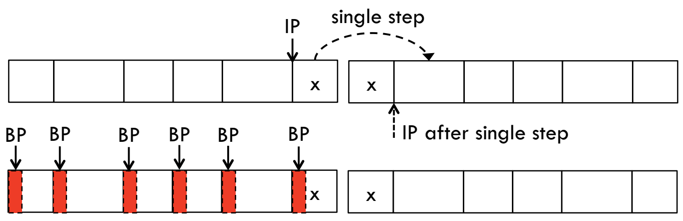
3.1 Low-overhead Instruction Reuse Profiling

To avoid the slowdown from frequent false-positive page traps we instead use breakpoints on instructions inside the cache line being sampled. Watchpoints can be effectively localized to a single cache line if breakpoints are set on every single instruction in the cache line being sampled. This way, one of the breakpoints would trigger when an instruction in the cache line is accessed again. As a result the application process will halt and the cache line reuse can be measured. We note that such an arrangement is only needed when measuring the temporal reuse of instruction cache lines. The more common (and probable) case in instruction streams is spatial reuse. To sample instruction reuse, the sampler halts the application execution after a random number of instructions. The reuse for the cache line pointed by the instruction pointer (ip) is sampled. The sampler single-steps the application process to execute exactly one more instruction in the instruction stream. If the instruction pointer now points to an instruction in the same cache line, a reuse distance of 0 is recorded for this sample. This is simply a case of spatial locality. However, if the execution lands in a new cache line, then the sampler decides to place watchpoint on the sampled cache line, and record its reuse distance when execution returns there. The reuse distance in this case will be greater than zero.

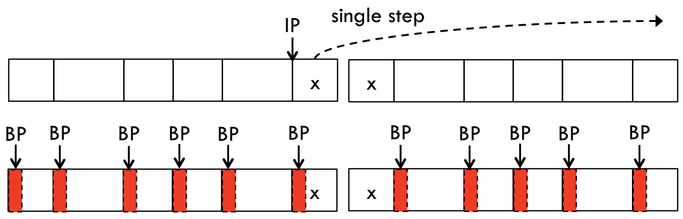
To monitor cache line reuse (when reuse is greater than zero), the sampler sets a breakpoint on the first byte of the instructions residing in that cache line. This is done by overwriting the first byte of the instructions with the breakpoint instruction. The original first byte of the instructions is saved in a data structure to be restored later. The sampler then records the number of instructions retired to date, and continues the execution of the application process. When execution returns to the sampled cache line, a breakpoint will trigger and halt the process. The sampler then measures the reuse distance of the sampled cache line as the difference of instructions retired to date and the number recorded previously. The sampler then restores the original first byte of the instructions in the cache line before continuing the program execution. The sampling technique described here is generic enough to be applicable to any architecture, even though our implementation is for x86.

Instructions in the x86 ISA are of variable length and not necessarily aligned to the cache line boundary. Consequently some instructions may span two cache lines. For example, in the benchmark *gcc* about 5% of the instructions spanned two cache lines. An instruction crossing a cache line boundary is called a Multi-Block Reference (MBR). If the sampled instruction happens to be a MBR, the sampler monitors both the cache lines containing parts of the sampled instruction. An MBR instruction is also the last instruction in a cache line. The following two cases can occur when single-stepping an MBR instruction: Figure 3.1a (if MBR is not a jump) – The execution arrives at the first instruction in the next cache line. The sampler sets breakpoints on the cache line containing the first byte of the MBR instruction. The sample for this cache line will be greater than 0. For the next cache line, 0 reuse distance is recorded, since the next instruction executed is in the same cache line where the later part of the MBR instruction is located. Figure 3.1b (if MBR is a jump) – The execution jumps to a cache line other than the very next. The sampler sets breakpoints on both the contiguous cache lines that the MBR instruction spans. Reuse distance is recorded for each cache line separately when their respective breakpoints trigger.

Instruction cache lines contain several instructions and sometimes also several basic blocks. The sampler may therefore stop at an instruction in the



(a) MBR instruction is not a branch



(b) MBR instruction is a branch

Figure 3.1. To measure instruction reuse for a Multi-Block Reference (MBR) (An instruction crossing a cache line boundary), the sampler single-steps the execution. If the execution remains on the second cache line (3.1a), the first cache line is watched, and 0 reuse is registered for the second cache line. In case of a branch (3.1b), both cache lines are watched.

middle of the cache line. If that happens, it is hard to know the address of the first instruction (or any preceding instructions) in that cache line in order to set a breakpoint. One solution is to use static analysis on the binary, however, we are interested in an approach to sample a dynamic instruction stream. This is especially a challenge in environments using JIT (just-in-time) compilation. We adopt a fast and effective learning mechanism to handle this problem. We decode instructions for a part of the instruction stream, starting from the instruction being sampled up to that cache line's boundary, and save their addresses in a hash map (knowledgebase). The sampler repeats this process every time it stops to take a sample. This way the sampler builds a knowledgebase over time about instruction addresses in sampled cache lines. When the sampler decides to monitor a cache line, it uses the address of the sampled instruction to hash into the knowledgebase to retrieve addresses of known instructions in that cache line. The sampler places breakpoints on the retrieved addresses to monitor the cache line. The sampler also adds information about previously unseen instructions belonging to a known cache line as it samples the same cache lines over and over again. This improves the accuracy of the sampler when trying to place breakpoints in known cache lines. The learning mechanism can however be a source of errors when computing reuse distances larger than zero. It is entirely possible that the sampler is never able to learn all instructions in the cache lines it is trying to set breakpoints for.

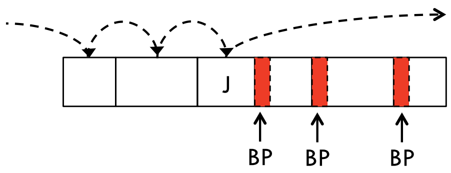


Figure 3.2. Instruction cache lines can sometimes contain several basic blocks. If the execution jumps into the middle of a cache line, breakpoints can only be inserted into the succeeding instructions. As a result, only parts of the cache line will be watched. If execution continues on the non-watched part, the measured reuse distance will be exaggerated.

As a result, breakpoints in later parts of watched cache lines may not trigger due to execution being deferred by a jump (Figure 3.2). This jump may be separating the unknown part of the cache line from the known part. This can happen at basic block boundaries, where instructions from two different basic blocks reside in the same cache line. This can result in reuse distances of such samples being exaggerated. However, in practice this did not affect the overall accuracy.

3.2 Sampling Overhead

Figure 3.3 compares the overhead of a breakpoint-based sampler (described in Section 3.1) to a instruction-reuse sampler implemented using page-based sampling (described in Section 2.3). Our breakpoint-based sampler is 10× faster than the page-based approach.

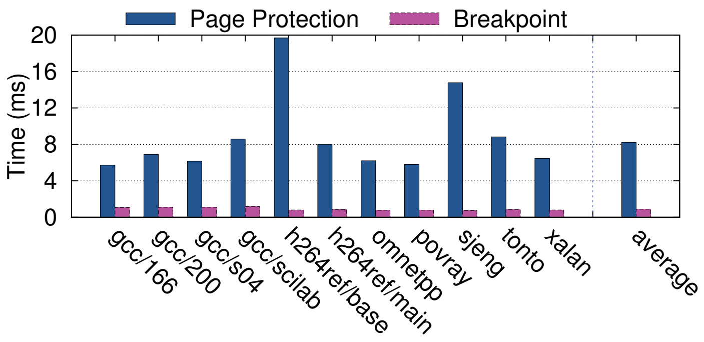


Figure 3.3. Overhead per sample for breakpoint and page-protection sampler. Page-protection has high overhead because of false-positives. Breakpoint based sampler is 10× faster than page-protection.

3.3 Optimizing Instruction Cache Performance

Applications' execution varies over time due to phases. Phase information can be used to estimate per-phase miss ratios and explore program regions with high instruction cache miss ratio. Such detailed information about individual program phases is not deducible by simply looking at the average miss ratio of the entire execution. We used ScarPhase [40], a low-overhead phase detection library to detect program phases. We combine phase information (from ScarPhase) with StatStack to model per-phase instruction cache miss ratio for an application. This helps us point out program phases that suffer from a high instruction cache miss ratio. We can use this information to optimize such program phases, for example, for reduced instruction cache (code) footprint.

By classifying instruction cache miss-ratios by phase for *gcc*, we found out that one recurring phase (that executes for about 10% of the time) has a miss ratio of about 1.4% for a 32 kB cache size. Using debug information we identified the part of the application source that is responsible for this phase and recompiled it with the *Os* (code size) compiler optimization. The *Os* option directs the compiler to optimize for smaller code size instead of performance. As a result, the active code footprint was reduced from 73 kB to around 64 kB for the entire execution and the instruction cache miss ratio for this phase reduced by almost a factor two, from 1.4% to 0.77%.

3.4 Summary

To fully understand an application's performance issues arising from memory bottlenecks it is important to understand the behavior of the instruction stream (in addition to the data stream). Paper I extends the low-overhead statistical cache model StatStack [8] to model the performance of instruction caches of varying sizes. Most importantly, the paper describes an efficient instruction-stream specific reuse profiling method. Instead of relying on watchpoints at the page-level granularity (Section 2.3), we developed a breakpoint-based approach that localizes the watchpoint to only the cache lines being monitored for temporal reuse. The method lowers the sampling overhead (compared to the page-based approach) by a factor of ten, on average slowing down applications by only 25% over native execution. The paper evaluates the accuracy of the models miss ratio estimation against reference miss ratio obtained from functional cache simulation. Finally, the paper shows how the model can guide compiler optimizations. With phase guided profiling, we modeled instruction cache performance over time to identify phases with large instruction cache footprint. We then demonstrate that this information can be used to reduce code footprint and optimize for instruction cache behavior, lowering the miss ratio significantly.

4. Data Prefetching

Processor throughput can be improved by applying techniques that effectively hide the memory latency of memory accesses that frequently miss in the cache. Data prefetching is one such technique that hides memory latency (and lowers miss rate) by identifying the behavior of individual memory access streams and proactively fetching data for them. Proactively fetching useful data can effectively hide the latency of performance critical loads, i.e. loads that suffer long latency cache misses, which can increase performance significantly. However, data prefetches must be issued well ahead in time of the actual access to effectively hide the memory latency. Workloads that largely traverse the memory in a uniform way, with regular strided accesses, benefit most from data prefetching as it is easy to predict the next access correctly and prefetch the required data in a timely manner. In modern processors prefetching is usually done at the cache line granularity and can be performed by the hardware, the compiler or the programmer. The prefetched data is placed in the caches.

4.1 Hardware Prefetching

Modern high performance processors largely employ built-in hardware prefetchers to proactively fetch useful data to the cache from the memory hierarchy. Hardware prefetchers monitor the processor's accesses to detect regular access patterns or strides, and use this information to automatically generate memory addresses that will be accessed in future and should be prefetched. Some hardware prefetchers also always prefetch the adjacent cache line after a demand access. This is called next-line prefetching. Modern multicore processors have elaborate cache hierarchies with several levels of caches. Usually, independent hardware prefetchers work across the different cache levels. When a hardware prefetcher predicts the cache line that should be prefetched, it requests that cache line from the next level in the memory hierarchy. The hardware prefetchers at LLC issue prefetch requests directly to the DRAM. As a result, prefetchers' activity at the LLC can increase the off-chip traffic. High performance processors typically employ aggressive prefetchers which can benefit performance significantly, but also increase useless prefetching due to their lower accuracy [34].

4.2 Performance Issues In Multicores

Aggressive prefetchers in high performance processors can significantly increase data requests to DRAM, sometimes resulting in a twofold increase in the off-chip data traffic. Such aggressive prefetching wastes shared resources such as the LLC space and off-chip bandwidth. This can severely impact the performance of threads co-running on neighboring cores when several cores become active and the LLC space and off-chip bandwidth are constrained. Such affects limit the overall throughput performance in multicores [7, 22, 34, 42]. Throughput performance in multicores can be increased significantly by appropriately regulating the aggressiveness of hardware prefetchers in execution environments where the shared off-chip bandwidth (and LLC space) is constrained [7, 16, 18, 22, 34, 42].

4.3 Software Prefetching

Many modern processors support data prefetching in software. Most ISAs provide prefetch instructions that can be used by the compiler or the programmer at appropriate places in the code. Prefetch instructions are usually non-binding, i.e., the prefetched data is placed in the cache hierarchy instead of a register. Some processors provide instructions that can be used to prefetch data to different levels in the cache hierarchy, such as the *prefetcht0*, *prefetcht1* and *prefetcht2* instructions supported by Intel processors. The non-temporal prefetch instruction (*prefetchnta* in Intel) hints the processor that the prefetched data should not be cached for long as it will not be re-used soon. Several studies [27, 28, 43, 44] have proposed using application execution profiles to hint compilers on where to insert software prefetches in the code.

4.4 Irregular Memory Accesses

Data prefetching is easily performed for memory accesses that traverse the memory space in a uniform, regular and predictable way. However, some memory accesses traverse the memory address space in a non-uniform way which is hard to predict. For example, irregular memory accesses arise due to traversal of linked data structures, such as trees and graphs. Hardware prefetchers in commodity processors do not handle such random accesses well and the memory latency of such accesses mostly remains fully exposed. Chapter 5 discusses efficient software prefetching for irregular accesses in detail.

5. Optimizing Performance for Irregular Applications

Modern processors typically employ stream (or stride) prefetchers that proactively fetch data from DRAM before it is requested by the application. This prefetching hides memory latency for applications with regular memory access patterns. However, hardware prefetchers in commodity processors do not handle irregular memory accesses well, that arise in many real-world applications, for example, due to pointer-chasing and indirect indexing. While it is possible to lower the latency of irregular accesses by prefetching them in software [1, 21, 26, 35, 37], doing so requires a method that conserves shared resources (LLC capacity and off-chip bandwidth) in multicores.

Hardware prefetchers in commodity processors can be very aggressive, sometimes prefetching more than twice the data required for an application [16, 18]. As a result, LLC space is wasted and off-chip bandwidth consumption grows. This hurts the performance of threads co-running on neighboring cores. Irregular prefetching mechanisms that greedily prefetch all identified pointers (such as [5, 26]) can further pressure shared resources and degrade overall throughput. Therefore, to efficiently handle irregular prefetches in multicores we have to be careful to not over-prefetch and stress the shared (LLC and DRAM bandwidth) resources. Paper II describes such a method.

5.1 Identifying Delinquent Loads

Load instructions that miss frequently in the data cache are called delinquent loads. Software prefetching must only target such instructions to improve application performance with minimal overhead. To identify delinquent loads we use StatStack to model per-instruction miss ratios (Figure 2.1). We only insert prefetches for load instructions that can benefit from prefetching in software. We use a cost-benefit analysis to measure the trade-off between the cost of the software prefetch and the performance benefit it can deliver. This analysis takes into account the memory latency, the latency of the target load instruction and the miss-ratio of the target load instruction (Figure 2.1). This helps us determine quantitatively if inserting software prefetch for a load will improve performance or not. For example, assume that a load hits in the L1 cache 90% of the time and 10% of the time in L2. A software prefetch for this instruction will be effective only 1 in 10 times to remove the L1 misses.

Assume that it takes 1 cycle to execute a software prefetch instruction and the latency to the L2 cache is 5 cycles. Then we will end up executing 10 prefetch instructions costing us 10 cycles and in return saving only 5 cycles (saving an L1 miss that hits in L2). Such software prefetching becomes an overhead and hurts performance instead of improving it. We employ such a cost-benefit analysis to avoid inserting software prefetches that may hurt performance.

A load is considered favorable for prefetching only if it is estimated to save more cycles than the software prefetching instructions will cost. The delinquent load identification pass performs the cost-benefit analysis and selects only those delinquent loads for which software prefetching benefits performance. Only these loads are further analyzed for irregular prefetching.

5.2 Identifying Irregular Memory Accesses

Since hardware prefetching can handle regular memory accesses well, we only insert software prefetches for irregular memory accesses. We identify and prefetch for irregular memory accesses by using per-instruction stride information and short instruction trace samples. This information is gathered during the sampling pass along with the data reuse information. Figure 5.1 illustrates the combined *data reuse*, *stride* and *trace* sampling for a memory reference. The additional functionality of recording strides and micro-traces can be added to the existing reuse sampler (Section 2.3) keeping the average overhead below 40% over native execution. We use the per-instruction stride information to separate regular-strided loads and irregular loads. Loads that exhibit a dominant stride (i.e. more than 70% of it's stride samples are similar) are categorized as regular-strided loads; the rest are categorized as irregular. Of all the delinquent loads identified (Section 5.1), only those categorized as *irregular* are considered for the prefetching analysis. Next, the algorithm extracts the sequence of instructions required to compute the address to prefetch, called the *Precomputation Sequence* [35], for each of these irregular delinquent loads, and schedules it at the assembler level.

5.3 Prefetching Irregular Memory Accesses

This work is about prefetching efficiently for irregular memory accesses – specifically pointer chasing (such as linked data structures) and indirect indexing (such as a[b[i]]). This section describes how sampled runtime information is used to determine the sequence of instructions required to prefetch data for an irregular load. This sequence of instructions is the *Precomputation Sequence* (PS). For each irregular load, the algorithm scans all sampled micro-traces associated with it, and detects unique sub-loops (that correspond to unique paths in a loop) and the probability with which each such path is

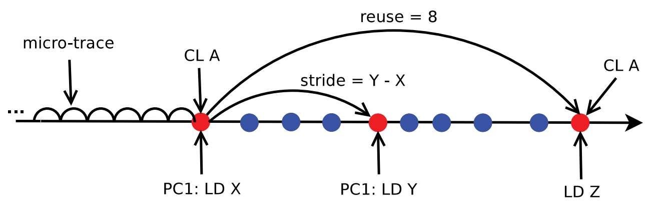


Figure 5.1. Data Reuse, Stride and Trace sampling - *Stride* is computed as the difference of data addresses accessed by successive executions of sampled load *PC1*. *Reuse distance* is the number of memory references between two accesses to sampled cache line (CL A). *micro-trace* is a small history of instruction addresses executed just before the sampled load.

executed. The sub-loop with the highest probability corresponds to the most frequently executed path in a loop, and is selected for PS extraction.

Linked Data Structures (LDS): The selected sub-loop is scanned (starting from the irregular delinquent load) to detect loads that have data dependencies. All such loads are added to a queue until a dataflow cycle is detected. At this point we should have all the instructions required to compute the address the irregular delinquent load will access in the next iteration. This sequence of instructions isolated from the selected path is the PS for the irregular load. If the irregular load accesses a deep nested object the extracted PS is longer. During our experiments we found that prefetching nested objects results in negligible performance improvements. This is because in out-of-order execution the miss latency of accessing a nested object is overlapped with the miss latency of accessing the next pointer, when the loops are small. Ebrahimi et al. make a similar observation in their work [6].

Indirect Indexing: In case of indirect-indexing the instruction producing the redirection-index (*b[i]* in *a[b[i]]*) is a regular-strided load. When scanning the selected sub-loop for loads with dataflow edges, the analysis tracks the instruction that produces the register containing the redirection index (*b[i]*) used to offset into array *a*. Once the instruction that produces the redirection index is found, we determine the stride of this instruction. At this point we should have all the instructions required to compute the address the indirect-indexing load will access in the next iteration. This sequence of instructions isolated from the selected path is the PS for the indirect indexing delinquent load.

The extracted PS can be used to schedule instructions that prefetch data for the next iteration for irregular delinquent loads. For detailed discussion on how the PS is scheduled at the assembler level to perform software prefetching see Paper II. The instructions in the PS contribute to execution overhead and are taken into account for the cost-benefit analysis described in Section 5.1.

5.4 Performance

To evaluate how this method performs in multicores we experimented with mixes of several applications co-run in parallel to completion. Several applications running in parallel on different cores stress the shared LLC and off-chip bandwidth. In such an execution environment, a resource conscious prefetching strategy will benefit performance most. Figure 5.2 compares the speedup from i) performance achieved with hardware prefetching alone (HWPF), and ii) performance achieved by combining hardware prefetching and irregular prefetching (HW+Ir-Pref) across 182 mixes of 4 applications that exhibit irregular memory access patterns. The average for irregular software prefetching alone (Ir-Pref) is also shown. The performance is normalized to the baseline of *No-Prefetching*. On average, irregular prefetching alone performs 10% better than the baseline. When hardware prefetching and irregular prefetching work together, performance over hardware prefetching alone improves by 8% on average.

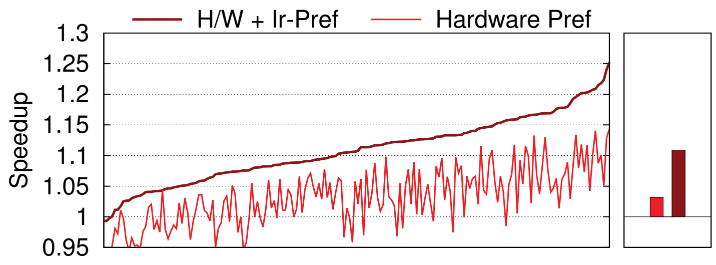


Figure 5.2. Performance across 182 mixed 4-core workloads on AMD Phenom II (averages on right). Mixes are sorted in increasing order of speedup for *HW+Ir-Pref.*

5.5 Binary Rewriting

There are several ways to schedule (insert) the instructions in the application to enable software prefetching. Dynamic binary rewriting and Just-In-Time compilation are two methods that provide the flexibility to manipulate the instruction stream efficiently at runtime. All software prefetching work in this thesis relies heavily on the ability to introduce software prefetching dynamically in the instruction stream. This is especially important when considering real-world scenarios where processor loads can vary significantly over time and enabling/disabling software prefetching may be necessary to manage performance at runtime. Paper IV discusses a runtime framework that has the capability of maintaining multiple versions of an application and switching between those versions at runtime.

5.6 Summary

Many real-world applications exhibit irregular memory access patterns during execution. Such memory accesses usually arise from the use of linked data structures and indirect-indexing. Hardware prefetchers in commodity processors can not handle irregular accesses well and the memory latency usually remains fully exposed for such memory accesses. To capitalize on this performance opportunity Paper II proposes a low-overhead framework that can do useful and shared-resource friendly prefetching of irregular memory access patterns in software. We use runtime profiles to i) identify irregular delinquent loads that can benefit from prefetching in software, and 2) identify and schedule instructions that prefetch useful data for such loads. The proposed method ensures shared resource (LLC space and off-chip bandwidth) conservation in multicores. As a result the method benefits throughput performance (on average 9%) in highly utilized multicores without wasting the shared resources.

6. Resource Efficient Data Prefetching

Hardware prefetchers in commodity processors are very aggressive and increase the useless off-chip traffic significantly [16, 18, 34, 42]. This wastes precious LLC space and off-chip bandwidth. While increased use of shared resources is usually not an issue when running a single thread, the limited off-chip bandwidth can quickly become a bottleneck when several threads co-run [11, 16]. In such an execution environment, where shared resources are constrained, useless prefetching (on any thread) can be harmful and impact the performance of co-running threads [34, 42]. To get better performance on highly loaded multicore processors, it is essential to regulate data prefetching so that it is timely as well as accurate, i.e., does not waste shared resources. In Paper III we present a framework that uses StatStack to accurately identify delinquent memory instructions and automatically insert software prefetches for them. Our prefetching scheme has good accuracy and lowers LLC pollution and off-chip bandwidth consumption. The full advantage of the scheme is realized when several cores are used and demand for shared resources grows.

6.1 Identifying Regular Delinquent Loads

We have already described in Section 5.1 how StatStack is used to identify delinquent loads. The work described in Paper III uses StatStack in the same way along with the cost-benefit analysis (Section 5.1) to single out the loads that should be targeted for software prefetching. One difference (w.r.t. irregular prefetching, Section 5.3) is that the number of prefetch instructions scheduled for prefetching of regular accesses is just one, i.e., the software prefetch instruction. So the cost-benefit analysis here assumes the overhead of executing only one additional instruction, unlike irregular prefetching which requires several. Our method uses per-instruction stride samples (see Section 5.2) to identify delinquent loads with regular strides. The stride analysis groups all strides of similar size that are likely to fall in the same cache line. After grouping similar strides, the analysis categorizes a load as having regular stride if more than 70% of its stride samples are similar. The analysis then selects the most frequent stride to compute a suitable prefetch distance.

6.2 How Far to Prefetch

To effectively hide memory latency, prefetches should be issued a suitable number of iterations earlier than the demand load. This is called the *prefetch* *distance* and it is a function of the memory latency and the number of cycles it takes to execute a single iteration of the loop where the delinquent load is located [32]. The relationship between these two latencies defines the suitable prefetch distance that should be used to ensure timely prefetching. We use performance counters to determine the average memory latency. To determine the latency of a single iteration of the loop we use *recurrence* (total memory instructions inside a loop)[1] and the average latency of a single memory access as measured using performance counters. Here, we make the assumption that the latency of the loop's iteration mainly comes from memory instructions. This assumption works well in practice. The prefetch distance computed is converted to an offset (number of bytes) for the software prefetch instruction inserted.

6.3 Cache Bypassing

Modern processors support a special data prefetching mechanism called non-temporal prefetch. The non-temporal prefetch instruction can be used to prefetch data into the L1 cache without evicting other data from the cache hierarchy. When this cache line is evicted, it goes directly to the DRAM instead of working its way up the cache hierarchy. This behavior is extremely useful when it is known that some data will not be reused from L2/LLC and can help retain temporally useful data in the caches longer. To improve on software prefetching, we include an analysis (originally proposed by Sandberg et al. [38]) to discover opportunities for cache bypassing. Briefly, the technique uses StatStack to determine if the data accessed by a delinquent load is reused from the cache hierarchy by subsequent memory instruction(s). This is determined using the per-instruction miss ratio curves (as shown in Figure 2.1) of the subsequent memory instruction(s). If these memory instructions do not reuse data from the caches their miss ratio will be constant between the L1 and LLC sizes. This information is enough to determine that the target delinquent load is a non-temporal access and its data can be cache-bypassed.

6.4 Prefetch Coverage & Insertion

To evaluate how effectively cache misses are covered by our method we looked at 12 benchmarks (11 from Spec CPU 2006 [9] and a genetic algorithm application *cigar* [23]), whose dataset does not fit in the LLC. On average the L1 miss coverage is 58%, similar to the coverage of a prefetcher that targets strided loads. The analysis identifies the loads targeted for software prefetch (Section 6.1), the type of prefetch, i.e. normal or non-temporal (Section 6.3),

[1]interleaving memory instructions between *PC1: LD X* and *PC1: LD Y* in Figure 5.1.

and the ideal prefetch distance (Section 6.2). x86 architectures support the *base+offset* addressing mode and inserting a single prefetch instruction of the format "*prefetch offset*(*base*)" suffices. For a load at address *A* using base register *base* to access memory, the prefetch is inserted right after it in the source as follows

```
A: load (base), dst
   prefetch[nta] prefetch-distance(base)
```

Here the offset is the prefetch distance and the base register is taken directly from the target load. Such optimizations can be applied directly at the binary level via dynamic binary rewriting (as discussed in Section 5.5). This method is aimed towards compiler and source independence, such that optimizations may be applied even when the source is not available.

6.5 Performance Scaling

Hardware prefetching increases useless off-chip traffic significantly, thereby frequently polluting the shared LLC, and consuming more off-chip bandwidth. This directly impacts the performance of threads co-running on neighboring cores. To assess the positive impact of our software prefetching method's resource conservation in multicores, we ran 180 randomly generated workload mixes on both processors. Each mix contains four randomly selected workloads (from the 12 benchmarks). The workloads in each mix are run in parallel on four cores to completion. Figure 6.1 compares the throughput performance (weighted speedup) when using our software prefetching method and hardware prefetching alone on a 4-core AMD Phenom II multicore processor. The baseline is the original mix with hardware prefetching turned off. Figure 6.1 shows the distribution function of throughput performance across the 180 mixes[2]. At best, software prefetching increases overall performance by 28% whereas hardware prefetching improves performance by 17%. On average, software prefetching improves performance by 10% over hardware prefetching. This is because our software-only prefetching method is accurate and conserves the shared LLC and off-chip bandwidth.

6.6 Summary

High performance processors widely employ hardware prefetching to hide memory latency. While hardware prefetchers can improve single-thread performance significantly, aggressive prefetchers waste shared resources, the off-chip bandwidth and LLC capacity [34]. This impacts the overall processor

[2]Both, hardware prefetching and software prefetching approach, have their points sorted according to their speedup.

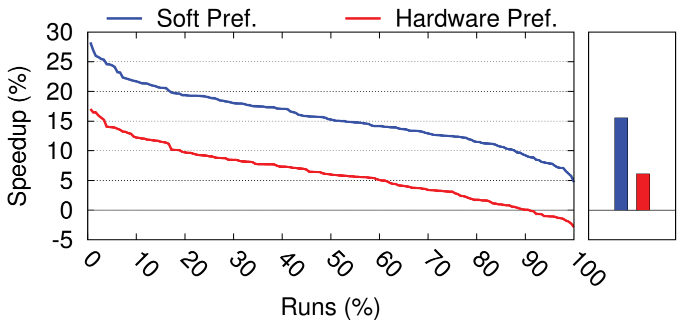


Figure 6.1. Distribution function of performance (weighted speedup) across 180 mixed workloads on AMD Phenom II (averages on right).

performance [16, 18]. Paper III investigates a resource-efficient prefetching method that helps improve throughput performance in multicores when shared resources are constrained. The proposed method i) accurately prefetches the required data, ii) avoids (useless) speculative prefetching, and iii) employs cache bypassing to retain useful data in the cache hierarchy. In contrast to hardware prefetchers (in commodity multicores), this resource-efficient prefetching method avoids useless off-chip traffic, and as a result avoids LLC pollution and lowers off-chip bandwidth demand. This improves throughput performance in multicores when several applications co-run and share resources. Across 180 4-application mixes that fully utilize the multicore, the resource-efficient software prefetching method achieves higher throughput (15% on average) compared to hardware prefetching (5% on average). This work highlights the importance of shared-resource friendly prefetching for optimizing performance in multicores.

7. Adaptive Resource Efficient Prefetching

In Chapter 6 we described how aggressive hardware prefetchers hurt overall performance in commodity multicores. We proposed a simple solution to the problem, that is to use static software prefetching instead of hardware prefetching when multicores are fully utilized. The accurate software prefetching method proposed ensures better average performance compared to hardware prefetching in fully loaded multicores. However, any static prefetching approach is sub-optimal given varying workload execution environments and processor loads, and can not be expected to maximize performance in all situations. So it is extremely important to identify and utilize the right data prefetching options for a given execution context [7, 22]. Paper IV describes such a framework, that identifies and applies the correct prefetching approach at runtime to maximize performance.

Experiments with multicores show that the optimal data prefetching strategy varies with changing execution environment. In Figure 7.1, the color coded best prefetching curve (called *static-max*) shows the overall speedup of the best performing option from 5 different prefetching options across 160 mixed workloads (of 4 co-running applications). The (five) different prefetching options are shown with different colors and are a combination of hardware and/or software prefetching applied uniformly across all cores. It is evident from the curve that there is no single prefetching option that serves as silver bullet and the winning prefetch strategy is context dependent. To maximize performance we need the ability to i) dynamically combine any available hardware and software prefetching options, and ii) discover the best prefetching option at runtime and adapt it. This functionality can be achieved in a runtime framework that facilitates binary rewriting (Section 5.5) or multi-versioning of application code to turn on/off software prefetching on the fly. The runtime framework, at the same time, is also responsible for controlling hardware prefetching on all cores. Such functionality gives us the ability to control data prefetching and combine it in several different ways in a live system.

7.1 Combining Hardware and Software Prefetching

Choosing the best prefetching option requires exploring performance across various hardware and software prefetching configurations. Hardware prefetchers on modern processors can be configured by programming Model-Specific Registers (MSR). To configure software prefetching the runtime should be

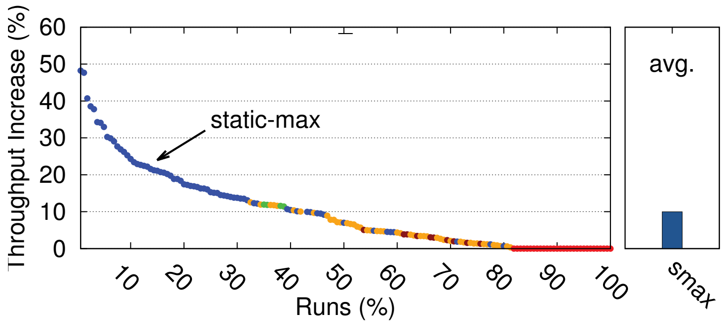


Figure 7.1. The *static-max* curve shows that applying the right prefetch settings can improve performance up to 50% and 10% on average (*smax*) over hardware prefetching. Each prefetch setting is shown with a different color. There is no single prefetch option that always performs best implying that the best prefetch option depends on the execution environment.

able to insert (and remove) software prefetches in the code on the fly. To dynamically adjust prefetch configurations and sample performance for those settings, the runtime must switch very quickly (to avoid performance overheads) between the different settings, which includes configuring hardware and software prefetching at runtime.

Inserting (and removing) software prefetches in the instruction stream on the fly (to enable switching of software prefetch configurations) is a major challenge. This can be achieved using JIT compilation, however, frequent JITing of code impacts performance and is not a feasible option. To that end we have extended the *Protean code* framework, originally developed by Laurenzano et al. [19], to maintain and use multiple versions of application. Using *Protean code* we create two versions of the application binary at execution start up: the original binary and another one with software prefetches inserted. The runtime can then switch between these two versions at runtime to turn off/on software prefetching. The runtime can combine this capability with various hardware prefetcher configurations to create several prefetching options. This work combines hardware and software prefetching options to configure prefetch settings in a total of 5 different ways – i) *Hardware Prefetching* only, ii) *Hardware Prefetching* + *Software Prefetching*, iii) *L1 Hardware prefetching* + *Software Prefetching*[1], iv) *Software prefetching* only (all hardware prefetchers disabled) and v) *No Prefetching*. The selected option is applied uniformly across all cores.

[1]Hardware prefetchers at L1 and L2/LLC can be configured separately on Intel Sandybridge processors. In this case hardware prefetching is enabled at L1 level only.

7.2 Choosing Best Option at Runtime

The runtime framework operates in two modes: 1) the exploration mode – when the runtime quickly samples performance across all prefetching options (listed in Section 7.1) to identify the best option, and 2) the performance monitoring mode – when performance is monitored for the prefetch configuration applied to sense phase changes. We use performance as a proxy to detect (phase) changes in execution environment. In the monitoring mode, when the runtime senses a phase change it enters the exploration mode to determine the best policy for the new phase. After sampling the performance (for 125 ms) for each prefetch option (in exploration mode), the runtime applies the option with the maximum performance. The runtime then re-enters the monitoring phase to sense changes in the execution environment. Paper IV discusses phase-based exploration in detail.

7.3 Performance

Figure 7.2 shows the performance (weighted speedup over hardware prefetching) of the phase-adaptive runtime prefetching method - *adaptive resource-efficient prefetching* (AREP) across 160 mixes of 4 applications. The mixes are sorted in descending order of static-max performance. The static-max shows that there is an opportunity to improve the throughput by 10% on average by statically choosing the best prefetching choice for the entire execution of the mix. Note that the static-max incurs no exploration. As described (in Sections 7.1 and 7.2) AREP monitors the performance of five different prefetch options at phase boundaries and adapts the best performing policy until the next phase change. AREP improves the performance by 8.1% on average and up to 49% in the best case.

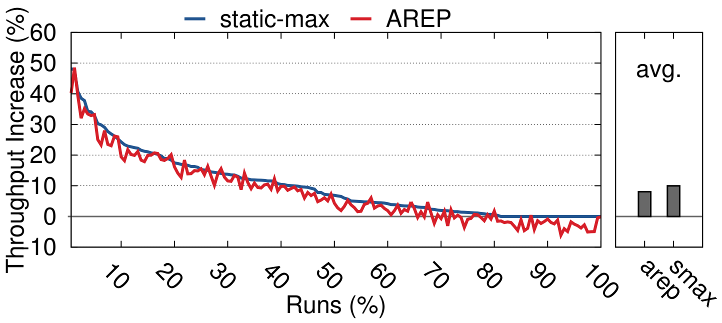


Figure 7.2. Performance of AREP compared to hardware prefetching (0-axis) and static-max. AREP, on average, is within 2% of the static-max performance. The mixes are sorted in descending order of static-max performance.

7.4 Summary

High performance processors employ hardware prefetching to hide memory latency. While hardware prefetchers can improve single-thread performance considerably, aggressive prefetchers waste shared resources, such as offchip bandwidth and last level cache capacity, which can impact overall processor performance [11, 16, 34]. Chapter 6 discusses that a resource-efficient software-only prefetching method can avoid useless prefetches and improve throughput performance significantly in fully utilized multicores. However, static software prefetching approach can be sub-optimal for some mixes, as seen in Figure 7.1. Moreover, hardware prefetching is not always harmful and performance can be improved further by selectively combining hardware and software prefetching at runtime. Paper IV describes a runtime framework (called AREP) that can configure 5 different data prefetching options across the multicore and select the one that performs best at runtime. The 5 prefetching options are a combination of hardware and software prefetching. The results show that AREP can improve throughput over hardware prefetching by up to 49% at best and 8.1% on average. Paper V looks at further improving overall performance by predicting the optimal prefetch combination where the prefetch options are adjusted independently across all cores.

8. Predicting the Impact of Data Prefetching on Performance

Chapter 7 showed that relying on a single data prefetching method is sub-optimal. Moreover it describes a framework (AREP) that dynamically explores several available data prefetching options at runtime and applies the single best prefetching option (uniformly across all cores) that improves throughput performance for the multicore. However, applying a single prefetch option is usually not the best setting for all co-running threads. Each application has different sensitivities to different prefetching options, and to how much the shared memory bandwidth [22] and LLC are stressed by co-running applications [30, 36, 45]. So, requiring the same prefetch strategy for each co-running application in a mix sacrifices a significant amount of performance. Prefetching needs to be configured at the per-application level to achieve optimal performance. AREP could potentially support such exploration. However, the large exploration space that needs to be traversed at runtime is a significant challenge. For example, in case of a 4-core processor and 5 prefetch options, a total of 625 (5^4) prefetch combinations need to be explored. The number of prefetch combinations grow exponentially with each added core. The approach used by AREP is to sample the performance of each prefetch option for a short duration. However, this approach is not feasible for exploring too many options at runtime.

Figure 8.1 shows the performance for a mix of 4 applications across the 625 prefetch combinations. The performance of the five options explored by AREP are clearly marked and show that there is significant room for improving performance further by configuring prefetching on a per-core basis. In this mix, AREP increases performance by 5%, whereas, the optimal combination increases performance by 18%. Paper V introduces an efficient and scalable method *Perf-Insight* that uses the application bandwidth and performance behavior (for the different prefetch options) in a mix to predict the optimal prefetch combination. The method looks at how each of the co-running applications respond to choices in its own prefetch strategy and the off-chip bandwidth pressure of other applications in the mix. The method uses this information to estimate each application's performance within a mix for a given prefetch option and behavior of co-running applications. This is used as a basis to predict the prefetching combination that maximizes performance.

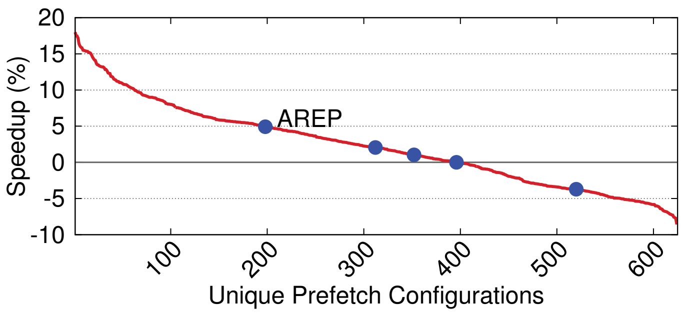


Figure 8.1. Distribution of performance gain over *No-Prefetching* across all 625 different possible prefetch combinations for a 4 application mix. The combinations are sorted by their overall performance. The 5 blue points highlight the settings sampled by AREP, where all applications have the same prefetch setting. In this mix the maximum speedup of AREP [18] is 5%, while the maximum speedup can be up to 18% with the per-application prefetch settings configured independently.

8.1 Application Bandwidth & Performance Behavior

To understand how prefetching choices affect an application's memory bandwidth consumption and performance (in a mix) we need to look into each application's sensitivity to prefetching (how much its individual performance increases as data is prefetched faster) and how effective each prefetching option is for that application. These sensitivities are particularly important in application mixes because off-chip bandwidth and LLC are shared, and often scarce, resources [13, 16, 18]. To quantify these sensitivities, we first investigate the relationship of individual applications' off-chip bandwidth consumption to that of the other applications executing in the mix, for each prefetch setting. This gives us insight into how off-chip bandwidth is shared within the mix, taking into account the aggressiveness of each application. Figure 8.2 shows the off-chip bandwidth share for one application as a function of different prefetch options (shown using different colors) and as a function of the off-chip bandwidth used by other co-running applications (x-axis). To connect this to performance, we then investigate how the individual applications' performance changes as a function of their own off-chip bandwidth consumption, again, as for each prefetch option. Figure 8.3 shows this relationship. The data in both Figures 8.2 and 8.3 show good uniformity. Using this data we have developed a model for application performance sensitivity in context of the application's off-chip bandwidth share and prefetch option applied. The models can then be used to estimate each application's off-chip bandwidth and performance (for a given prefetch setting) in a mix. However, to make this approach practical it is important to effectively estimate the applications' off-chip bandwidth and performance models using fewer data points.

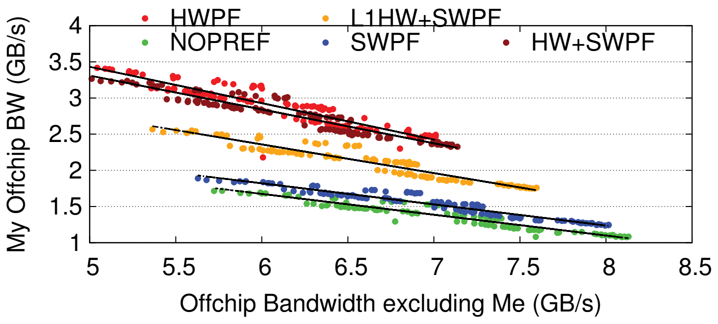


Figure 8.2. Off-chip bandwidth for *GemsFDTD* for the five different prefetch options as a function of the bandwidth of the other applications in the mix, across all 625 possible combinations in the application mix. For each prefetch option there is a clear linear relationship between *GemsFDTD's* bandwidth and the bandwidth demand (pressure) of the other applications in the mix (excluding GemsFDTD). The black lines show the linear BW-BW Model based on the reference data from all 625 possible prefetch combinations for this mix.

8.2 Estimating Application Behavior Efficiently

The data shown in Figures 8.2 and 8.3 indicate that we can estimate the application's off-chip bandwidth and performance behavior in a mix using a simple linear model for each prefetching choice. Figure 8.2 shows the linear fit called BW-BW Model for each prefetching choice, whereas the BW-Perf Model is illustrated in Figure 8.3. Further, with such models we can iteratively solve for how any combination of prefetching choices will perform in a mix. However, the data presented so far is based on exhaustive evaluations of all 625 possible combinations, and is neither practical to gather in itself (taking nearly 21 hours with our 2 minute-per-mix experiments) nor remotely feasible for larger core counts. To lower the number of data points to model the off-chip bandwidth and performance models we note that the linear BW-BW Models and BW-Perf Models can be approximated by points at either extreme: the best (most) off-chip bandwidth achieved for each prefetch setting and the worst (least). If we can efficiently find these points (or close approximations to them) then we can reduce the required data to two points per application per prefetch setting.

To identify the best and worst bandwidth points for each prefetch option we make the following assumptions: The best conditions for an application (most off-chip bandwidth) will occur when the other applications in the mix use as little off-chip bandwidth as possible, which means setting them to *No-Prefetching*, and the worst (least off-chip bandwidth) will occur when the other applications in the mix use as much off-chip bandwidth as possible, which means setting them to *Hardware Prefetching*.

The accuracy of this best case/worst case approximation is shown for one application in a 4 application mix in Figures 8.4a and 8.4b. The results from the best case/worst case points are shown for each prefetch setting with col-

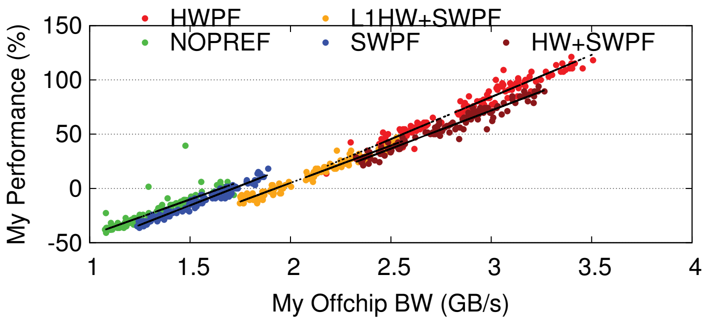


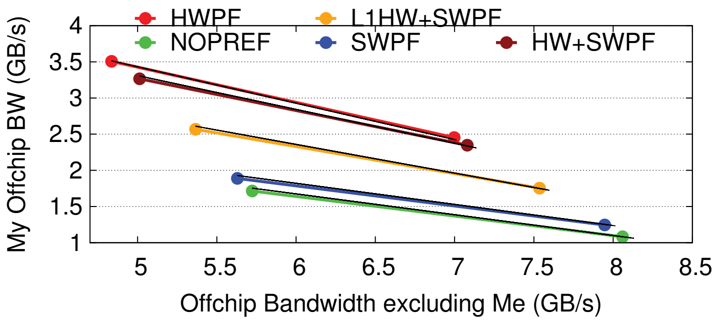
Figure 8.3. Performance (speedup over *No-Prefetching*) of *GemsFDTD* for the five different prefetch options as a function of its bandwidth, across all 625 possible combinations in the application mix. For each prefetch option there is a clear linear relationship between *GemsFDTD's* performance and its bandwidth. The black lines show the linear BW-Perf Model, based on the reference data from all 625 possible prefetch combinations for this mix.

ored dots (showing the extreme points) and the linear fit with colored lines. The linear fit to all 625 combinations is shown with black lines for reference. The fit to the best case/worst case points is very close to the fit using all combinations, while only requiring gathering data for 10 combinations (for this application). Across all 4 applications in the mix, only 34 combinations are required to approximate the BW-BW Model and the BW-Perf Model. This can be effectively used to model the sensitivity of an application's performance to its share in the off-chip bandwidth. The number of combinations required to estimate the BW-BW Model and BW-Perf Model grow linearly in the number of cores, whereas the total prefetch combinations grow exponentially.

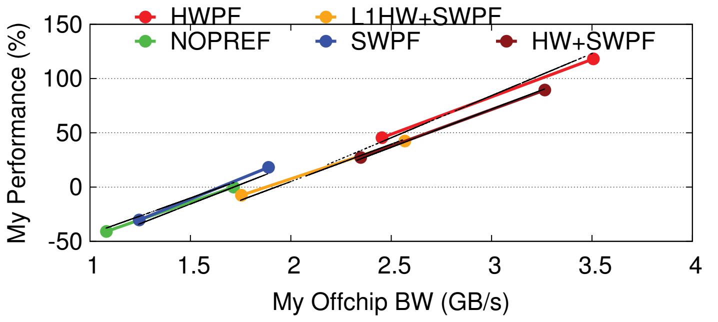
8.3 Performance Prediction

The BW-BW Model and BW-Perf Model allow us to understand the interaction between the off-chip bandwidth demands and allocations in a mix and the resulting performance for each application, as a function of the chosen prefetch settings. To determine the performance for an arbitrary selection of prefetching choices in a mix, we run an iterative solver that uses the BW-BW Model to find a steady-state solution to how much bandwidth each application receives. For each application we begin by using the four best case settings as the starting point for our iterations. Once we have arrived at a steady-state for the applications' bandwidths, we can estimate performance from the BW-Perf Model for each application.

To find the best performing prefetch combination, we repeat the above for each setting for each application (625 combinations in total), which requires 34 best-case/worst-case data points. Figure 8.5 compares the predicted perfor-



(a) Application Off-chip Bandwidth Model - *GemsFDTD*



(b) Application Performance Model - *GemsFDTD*

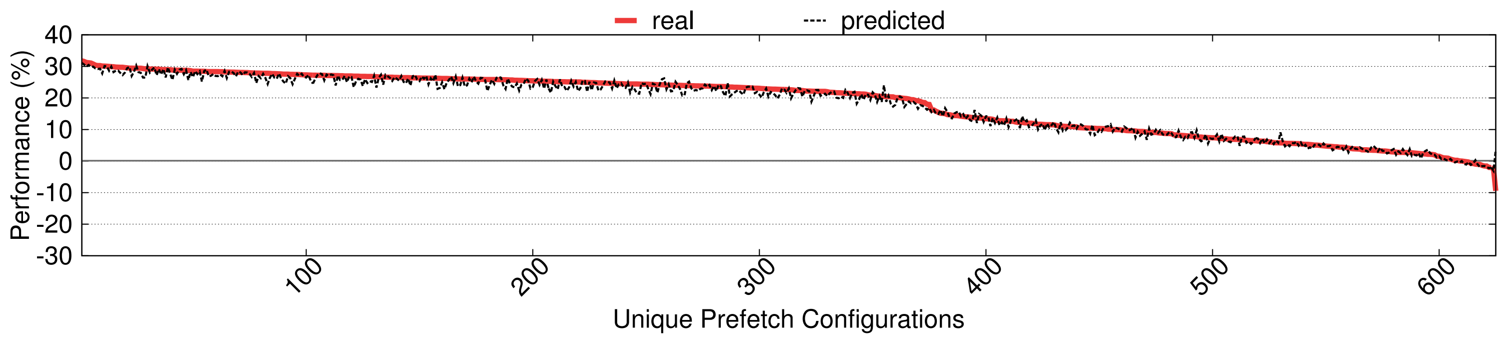
Figure 8.4. Comparison of our Off-chip Bandwidth Model and Performance Model for one application in a 4 application mix. Colored points and lines show the result of modeling behavior using only the best case and worst case points, while black lines show the results of using the all 625 combinations (also shown in Figures 8.2 and 8.3). The model created using the extreme points accurately depicts the overall off-chip bandwidth and performance trends for the applications for each prefetch choice, as can be seen by the very good match between the black and colored lines.

mance to reference data for a single mix. Prefetching combinations are sorted in decreasing order of real performance. Here the clear shift in performance after the best 380 configurations is correctly captured by our model. The average errors for predicted performance is 1% in this case. To select the optimal prefetching configuration we simply select the combination with the highest predicted performance. Estimation errors are below 3.5% for most mixes used in experiments, which shows that using best case/worst case input data and our iterative solver can predict performance accurately. Perf-Insight helps deliver near-optimal performance. Across 14 mixes, the performance increase over No-Prefetching are: 9% for hardware prefetching, 22% for the oracle, 15% for AREP and 21% for Perf-Insight. Perf-Insight's performance on average is merely 1% shy of the optimal (oracle).

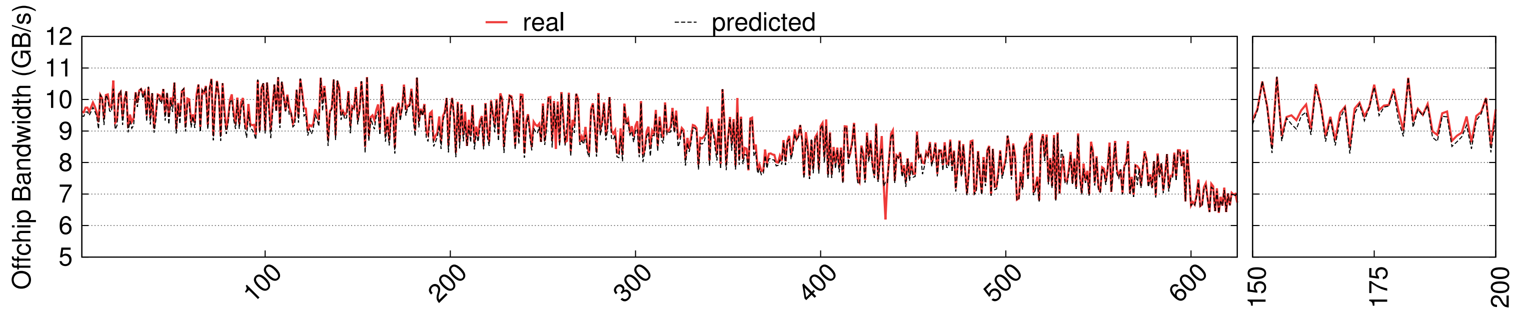
8.4 Summary

Hardware and software data prefetching options can be effectively combined to improve multicore performance using the runtime method (AREP) described in Paper IV. AREP regulates data prefetcher aggressiveness (uniformly across all cores) at runtime to improve overall performance. However, this approach leaves considerable performance opportunity un-explored as data prefetching is not configured at the per-core level. The reason AREP avoids per-core tuning of prefetch settings is that the prefetch combinations grow exponentially with increasing number of cores, and it is not possible to monitor them all at runtime.

We developed an efficient way to solve this problem with an approach that predicts the performance of all prefetch combinations based on application behavior information used as input. Paper V introduces Perf-Insight, a new, simple model for understanding the interaction between applications' individual performance and off-chip bandwidth, the off-chip bandwidth demand of the other applications in the mix, and the per-application prefetching choice. To make this approach practical, we developed a technique to efficiently calibrate the models using the best-case/worst case settings, which allows us to both calibrate applications while running in the mix and also scale the calibration procedure linearly with the number of applications. Finally, we use this approach to predict throughput performance (of mixes) across all combinations of available prefetch options, which guides us to select the best prefetch option on a per-core basis to achieve near-optimal performance. While we demonstrated Perf-Insight with a collection of 5 hardware and software prefetching techniques and chose the combination with the best performance, the Perf-Insight approach works with any available prefetching choices that can be controlled at runtime. With Perf-Insight we are able to robustly choose near-optimal prefetch combinations for a range of workload mixes, and demonstrate significant real-world performance benefits across a wide variety of workload mixes on commodity hardware.



(a) Performance Prediction



(b) Off-chip Bandwidth Prediction

Figure 8.5. Comparison of Performance prediction (top) and Off-chip Bandwidth prediction (bottom) across the all combinations of prefetch settings for a mix of 4 applications. Off-chip Bandwidth prediction error is less than 1% on average for this mix. An enlargement of several of the mixes is shown to demonstrate how accurate the bandwidth prediction is.

9. Related Work

This thesis discusses software techniques that mainly utilize data prefetching to improve performance in multicores. The list of prefetching work is too detailed to cover exhaustively, so we discuss the most relevant prior work that relates to the prefetching techniques described in this thesis.

Several works have used software prefetching to improve single-thread performance [28, 29, 35, 43, 44, 46]. Mowry et al. [32] and Santhanam et al. [39] describe a compiler based software prefetching algorithm based on static analysis. Implementation of similar software prefetching algorithms are available in production compilers such as GCC and Open64. However, software prefetching guided by static compiler analysis is shown to degrade single-thread performance [29]. Profile-guided optimization (PGO) techniques, such as [27, 43, 44], have used stride profiling to discover loads with frequently recurring strides, to insert software prefetches for them. Lee et al. [20] investigated combining hardware prefetching and software prefetching for single-threaded applications, concluding that caution should be exercised when mixing the two. In contrast to their work we have shown that hardware prefetching can be combined with software prefetching in a useful way to increase throughput performance in multicores. Dynamic optimization frameworks such as [3, 24] use extensive (Itanium-specific) architectural register support for sampling relevant memory events and performance counters to detect delinquent loads and insert software prefetches for them. Beyler et al. [3] monitor stride behavior of regularly occurring loads at runtime using a separate thread context. Once the monitoring thread notices a regular stride, it inserts software prefetch for that load on the fly. However, this approach degrades performance for several benchmarks when the prefetching can not amortize the cost of the runtime system's overhead. Such techniques that utilize architecture specific features are also not easily portable to other platforms.

Rabbah et al. [35] proposed a profile driven compiler-based method for isolating and scheduling the *precomputation sequence* (PS) (referred to as *load dependence chain* (LDC)) for both *regular* and *irregular* delinquent loads. They proposed the framework as part of the ORC (open research compiler) targeting VLIW architectures. The PS is extracted using profiler feedback, the compiler then appropriately inserts the PS instructions in scheduling slots at low-level IR. Our method, described in Paper II, of PS extraction and its speculative scheduling at the assembler level is an adaptation of their method. Other software-based LDS prefetching approaches such as [21, 26, 37] either require the programmer or the compiler to identify pointer accesses that frequently miss in the cache, through memory access profiling. Software prefetch instructions are then inserted ahead of pointer access to hide some memory latency. For pointer-chasing codes they did not observe large performance improvements. Software-based approaches such as [4, 25] use a secondary thread context constructed by compilers to run-ahead and prefetch for the main application thread on multi-threaded processors. Inagaki et al. [10] and Adl-Tabatabai et al. [1] proposed software-based prefetching methods that are limited to managed runtime systems (specifically Java) as they require information about object metadata.

Content-directed prefetching (CDP) [5] is a predictive hardware-software pointer prefetching method that uses compiler hints to identify pointer fields in newly fetched cache blocks. The pointer-addresses identified in newly fetched cache blocks are then prefetched greedily. Ebrahimi et al. [6] improved CDP by proposing profile-driven hints to identify and prefetch only the most frequently accessed pointer fields in the fetched cache blocks.

Liu and Solihin [22] have proposed analytical models for bandwidth partitioning to identify when prefetching can help in improving system performance. Sandberg et al. used reuse-distance based cache modeling to insert non-temporal prefetch instructions to cache bypass the data that is not reused from the lower level caches [38]. Similarly, Laurenzano et al. [19] proposed a runtime mechanism to find opportunities to insert non-temporal prefetch instructions in batch applications to conserve LLC space so that user-facing applications' performance in datacenters remains predictable. Jiménez et al. implemented a runtime mechanism for exploring and adjusting hardware prefetcher configuration on a POWER7 processor to maximize performance [11]. The POWER7 processor allows the prefetcher aggressiveness to be configured at 7 different levels. Their runtime method explores the best hardware prefetcher settings on per-core basis (for two cores only) and applies the one that performs best. Unlike our work, their method avoids interaction with software prefetching by explicitly disabling software prefetch insertion

10. Summary

Multicore processors are the most common general-purpose processing platform used today and consist of several processors, termed *cores*, hosted on a single chip. Typically, all cores in multicores share the last-level cache and the DRAM bandwidth. When several threads co-run across several cores, each of them affects the performance of the neighboring cores depending on how they utilize the shared resources. In such execution environment, the use of the memory hierarchy, especially the resources shared by all cores play a significant role in defining the overall performance of the multicore. This thesis investigates software-only techniques that improve the way software utilizes the memory hierarchy, especially the shared resources, in an effort to improve overall multicore performance.

Cache behavior modeling techniques, such as StatStack, have typically been used to model the way applications' data stream uses the cache hierarchy. StatStack uses profiled information about how data is reused over time to estimate (using a probabilistic model) how frequently data requests miss in the cache hierarchy. In Paper I we extend StatStack to model instruction cache behavior. Here the model uses information about instruction reuse to predict how often the application misses in the instruction cache. This is used to identify performance bottlenecks arising due to code segments that are too large to fit in the instruction cache. We also introduce an instruction-stream specific profiling method that is, on average, $10\times$ faster than the profiling method used for StatStack. With a case study we show that application phases with high instruction-cache miss ratio can be identified using this method, and optimizing for code size (instead of performance) for such phases can lower the miss ratio by half.

Hardware prefetchers in commodity processors are good at improving performance for applications with regular memory accesses but do not handle irregular memory accesses well. Performance, for irregular applications, can improve by prefetching irregular accesses in software while hardware prefetching takes care of regular memory accesses. In Paper II we describe a low-overhead framework that enables accurate (resource-efficient) software prefetching for applications with irregular access patterns. The framework identifies memory accesses i) that miss frequently in the cache hierarchy, ii) that are irregular and can not be handled by the hardware prefetcher, and iii) that benefit performance when targeted by software prefetching. The framework uses sampled trace information to determine the chain of instructions required to prefetch data for the targeted irregular loads in software and ensure resource-conscious prefetching. As a result, the additional prefetching (targeting irregular loads) improves performance (by 10% on average) without increasing the pressure on shared resources, even when several irregular applications co-run on a multicore.

Hardware prefetchers in high performance multicore processors are aggressive and increase DRAM traffic significantly. This increases pressure on shared resources and impacts the performance of applications co-running on neighboring cores. Overall, multicore throughput performance is affected negatively from useless prefetching activity. In Paper III we describe a software prefetching mechanism that utilizes shared resources more efficiently, coupled with timely prefetching of data. As a result it performs better than hardware prefetching on fully utilized multicores. Using a cost-benefit analysis the method singles-out loads (with regular strides) where software prefetching benefits performance and targets them only. In addition to accurate prefetching, the method also makes use of cache-bypassing, whenever possible, to maintain temporally useful data longer in the cache hierarchy. In a fully utilized modern multicore processor, we show that using this shared resource conscious software prefetching approach improves performance by 10%, on average, over hardware prefetching.

In Paper IV we show that software prefetching alone is sub-optimal for maximizing performance in highly utilized multicores, and hardware prefetching performs better in some cases. The optimal data prefetching strategy is context dependent and differs across varying multicore execution environments. This paper describes a framework, called *Adaptive Resource-Efficient Prefetching* (AREP), that dynamically adapts the prefetching strategy, uniformly across all cores, at runtime. AREP chooses from 5 different prefetching options (combinations of hardware and software prefetching) that exert varying pressure on shared resources. The framework selects between the 5 prefetching options by profiling their performance at runtime and applying the one that performs best. By applying the right prefetching strategy at runtime AREP is able to increase throughput by up to 49% and more than 8% on average.

Paper V introduces a scalable method *Perf-Insight* to achieve near-optimal performance by varying prefetch options independently across all cores in a multicore. AREP (Paper IV) chooses from 5 prefetch settings and applies the selected prefetch setting uniformly across all cores. This strategy was adapted to avoid exploring the large combination space, that results from varying the prefetch settings at the per-core level, at runtime. The unique combinations grow exponentially in the number of cores and are impossible to explore exhaustively at runtime. Perf-Insight introduces a method that uses the off-chip bandwidth behavior of applications in a mix as a function of their prefetch setting to predict the bandwidth share of each application and its performance in the mix. The method only requires to profile a handful of prefetch combinations to use as a baseline to model the bandwidth sharing and performance scaling effects. Using the bandwidth and performance models Perf-Insight accurately predicts throughput performance for all prefetch combinations. This guides in choosing the right prefetch combination that delivers optimal performance.

Efficient utilization of the memory hierarchy is key to better performance in multicore processors. This work presents a software approach towards achieving efficient utilization of memory resources in multicore processors to maximize performance. *First*, we present a method to identify performance bottlenecks arising in the instruction stream of software and propose a possible way to resolve it with application code compaction. *Second*, we propose a low-overhead framework for efficient prefetching of data for irregular applications. This software technique prefetches the data required by the software without stressing the memory resources shared by all cores, which results in significant performance gains. *Third*, we show that hardware prefetchers in commodity processors are sub-optimal and propose a resource efficient software-only prefetching method that performs better in fully utilized multicores. *Fourth*, we combine hardware and software prefetching in a runtime framework that enables us to explore and apply the best prefetch option in a given execution environment. *Finally*, we develop a method that enables us to predict the right prefetch options on a per-core basis to achieve near optimal performance with a heterogeneous mix of prefetch options applied across the cores.

11. Svensk Sammanfattning

Idag finns flerkärninga (såkallade multicore) processorer in nästan alla datorer, från smartphones till högprestande servrar. En flerkärnig processor består av flera kärnor (där varje kärna är som en traditionel processor) som sitter på ett och samma chip. Dessa kärnor jobbar tillsammans för att leverera samma beräkningskapacitet som flera datorer fast på ett chip. Den här paradigmen har möjligjort att vi kan använda hårdvaran på ett energieffektivare sätt samt leverera bättre prestanda.

11.1 Bakgrund

Dessvärre finns det en begränsning med flerkärninga processors då kärnorna delar vissa kritiska resurser. Utöver det, så kan varje kärna använda data mycket snabbare än den kan hämta ny data från huvudminnet (DRAM). Då minnet också är en delad resurs skapar det en märkbar prestandaflaskhals i systemet som begränsar hur fort varje kärna kan arbeta. Den här minnesbegränsningen har varit känt sedan länge, långt innan flerkärninga processorer började användas. Ett sätt att minska latensen att hämta data och öka bandbrädden (antalet parallella hämtningar) från huvudminnet är att använda en så kallade cache. En cache är ett mindre minne som är mycket snabbare än huvudminnet (pga av dess storlek) som placeras mellan processorn och huvudminnet. En cache lagrar den senast använda datan. Det gör att kärnan kan hämta data snabbt från cacheminnet istället för huvudminnet då det är vanligt att samma data återanvänds flera gånger. För att maximera prestandan används flera cachar av olika storlek och egenskaper (latens, energi, bandbredd). Figur 1.1 visar en typisk bild av en modern flerkärnig processor. Alla kärnor har två nivår av egna privata cacher (L1 och L2) och en delad stor sista nivå cache (LLC). Eftersom sista nivå cachen (LLC) är delad av alla kärnor så blir den i likhet med huvudminnet en begränsad
a tgång som alla kärnor tävlar om. Tillsammans har de olika cacharna och minnessystemet en viktig roll i hur fort en flerkärning processor kan arbeta. Det här arbetet fokuserar huvudsakligen på hur man med hjälp av mjukvara (datorprogram) kan använda de delade resurserna (cacheminnen och huvudminne) på ett effektivt sätt för att förbättre prestandan hos moderna flerkärniga processorer.

För att noggrant och kritiskt analysera datorprograms prestanda är en viktig del att förstå hur programmets minnesbeteende ser ut. Det finns flera sätt att ta fram information om programs minnesbeteende, från långsam detaljerad simulering till snabb minnesmodellering. Cachemodellering använder information om hur ett program återanvänder data för att modellera till exempel programmet prestanda som en funktion av de olika cacharnas storlek (se Figure 2.1). Sådana modeller ger bra intuition om hur bra ett program använder cachehierakin och kan därför användas för att vägleda olika optimeringar som förbättrar prestandan.

11.2 Sammanfattning av Forskningen

Varje kärna har i verkligheten två olika L1 cachar. En för instruktioner och en för data. Detta gör att en processor kan hämta instruktioner samtidigt som den hämtar data. Ett datorprograms instruktioner talar om för processorn vilka operationer som ska utföras på datan. Både instruktionscachen och datacachen spelar en viktig roll i ett programs prestanda. Om ett program missar i L1 instruktionscachen (dvs instruktionen finns inte sparad i L1 cachen), stannar processorn up tills instruktionen som den väntar på hämtas från någon av de andra nivåerna i minneshierarkin (till exempel L2, LLC eller DRAM). Dvs, ju oftare ett program missar i L1 cachen ju långsammare går programmet. Cachemodellering har huvudsakligen används för att analysera ett programs databeteende. Men, samma metod kan också användas för att analysera programs instruktionensbeteende. I artikel 1 utökar vi en cachemodelleringsteknik, kallad StatStack, till att modellera instruktionscacher. Den nya modellen använder information om hur programmets instruktioner återanvänds över tid for att förutspå om programmet kommer missa i instruktionscachen (om programmet är för stort för att få plats i instruktionscachen). Informationen används sedan för att hitta vilka delar av programmet som får prestandaproblem. I artikeln föreslår vi också en snabb metod med låg overhead för att spara under körningen specifik information som sedan används av modellen. I studien visar vi också att vi kan identifiera delarna av programmet med prestandaproblem och optimera dem så att de får plats i instrucktionscachen. Resultatet är att vi minskar antalet missar i instruktionscachen med 50%.

Moderna processorar använder speciell hårdvara, så kallad förhämtare (eng. prefetcher), för att förbättre prestandan. En förhämtare förutspår om ett program kommer att använda viss data och hämtar datan innan programmet bejär det. Datan finns således i det närmaste cacheminnet istället för i huvudminnet vilket förbättrar prestandan då den kan hämta datan mycket snabbare. För att göra det, upptäcker förhämtaren mönster i programmet beteende. Förhämtare är för det mesta bra på att upptäcka och hämta reguljära mönster men får problem med irreguljära vilket skadar prestandan. Ett sätt att lösa det här problemet är att låta en hårdvarubaserad förhämtare hantera reguljära mönster och mjukvarubaserade förhämtare hantera irreguljära mönster. I artikel 2 beskriver vi ett effektivt mjukvarubaserat verktyg för att noggrant (resurseffektivt) förämta irreguljära mönster i datorprogram. Verktyget identifierar minnesaccesser som: 1) missar ofta i cachehierarkin och har hög minneslatens, 2) är irreguljära och som hårdvaruförhämtaren inte kan hantera, och 3) förbättar prestandan när dem hämtas med den mjukvarubaserade förhämtaren. Verktyget använder en snabb profileringsteknik med låg overhead för att samla den information krävs för att förhämta data. Resultatet av studien visar att med hjälp av vår mjukvarubaserde förhämtning (som endast fokuserar på irreguljära mönster) kan prestandan förbättras med 10%. Den gör det utan att öka trycket på delade resurser, även när flera program körs samtidigt.

Hårdvarubaserade förhämtare i högprestanda datorer är ofta aggressiva och öker minnestrafiken markant. Det beror på att de hämtar mer data än vad programmet använder då den hittar mönster som inte alltid stämmer. Det här ökar trycket på delade resurser och försämrar prestandan hos de andra programmen som körs samtidigt på de andra kärnorna. I artikel 3 beskriver vi en mjukvarubaserad förhämtningsmetod som använder the delade resurserna mer effektivt. Den använder en kostnad/prestanda analys som hittar instruktioner som hämtar data med reguljära mönster i programmet och där mjukvarubaserad förhämtning förbättrar presendan. Med program som stressar de delade resurserna förbättar den nya metoden prestandan med 10% i jämnförelse med en hårdvarubaserad förhämtning.

I artikel 4 visar vi att mjukvarubasered förhämtning inte alltid är bäst för prestande i högt belastade system där många program körs samtidigt, samt att hårvarubaserad förhämtning presterar bäst i vissa fall. En optimal förhämtningsstrategi beror på vilka program som körs och vilka flerkärninga datorer de körs på. I artikeln bestriker vi verktyget Adaptive Resource Efficient Prefetching (AREP) som dynamiskt anpassar vilken förhämtningsstrategi som ska användas medan programmet körs. AREP väljer bland 5 olika strategier (olika blandningar av hårdvaru- och mjukvarubaserade förhämtare) som sätter olika tryck på de delade resurserna. Verktyget bevakar de olika strategierna och väljer den som har den bästa prestandan. Studien visar att AREP förbättrar prestandan med up till 49% och i genomsnitt 8% i jämnförelse men endast hårdvarubaserad förhämtning.

I artikel 5 beskriver vi en skalbar metod Perf-Insight för att uppnå nästan optimal prestanda genom att variera de olika förhämtningsstrategierna oberoende för alla kärnor i en flerkärnig dator. AREP (artikel 4) valde den bästa av 5 strategier för alla kärnor (dvs samma strategi för alla kärnor). AREP använde den metoden för att minska antalet kombinationer som måste testas. Till exempel, i en flerkärnig dator med 4 kärnor och 5 olika strategier blir det totala antalet unika kombinationer 625. Antalet unika kombinationer växer exponentialt med antalet kärnor vilket blir omöjligt att utforska under körningens gång. Perf-Insight använder en ny metod som använder minnesbandbreddsbeteendet i en blandning av program som en funktion för att förutspå bandbreddsbehovet och prestandan av varje program i blandningen. Metoden behöver bara testa ett fåtal kombinationer för att modellera bandbreddsdelningen och hur prestandan påverkas. Med bandbredd och prestanda modeller kan Perf-Insight noggrant förutspå gemensam prestanda och välja rätt förhämtningsstrategi. I genomsnitt förbättras prestandan med mer än 12% i jämnförelse med endast hårdvarubaserad förhämning.

Att effektivt använda minneshierarkin är nyckeln till bättre prestanda i flerkärninga datorer. Det här arbetet presenterar en mjukvarubaserad ansats att uppnå effektivt användande av minnesresurser i flerkärninga datorer för att maximera prestandan. Första, vi presenterar en metod som identifierar prestandaproblem med instruktioner i program och föreslår en lösning att minska storleken på programmet. Andra, vi presenterar ett verktyg med låg overhead för att effektivt förhämnta data med irreguljära mönster. Den här tekniken förhämtar data utan att stressa delade minnesresurser vilket resulterar i markanta prestandavinster. Tredje, vi visar att hårdvarubaserade förhämtare inte alltid är optimala och föreslår en resurseffektivare mjukvarubaserad förhämtare som presterar bättre på belastade system. Fjärde, vi kombinerar hårdvarubasereda förhämtare med mjukvarubaserade i ett verktyg som gör att vi kan utforska och tillämpa den bästa strategin i ett givet system. Slutligen, vi utvecklar en metod som kan förutspå de bästa förhämtningsparametrarna för varje kärna för att uppnå nästan optimal prestanda bland många inställningar.

12. Acknowledgments

As I sit down to pen this and think about those who I should thank and acknowledge for helping and supporting me reach this point, I realize the list is long if not endless. I would like to start by thanking my advisor Erik Hagersten. I have been his student since 2007 when I first attended the famed "Advanced Computer Architecture" class. In early 2009 I became his Masters thesis student and later the same year his PhD student. I am extremely grateful to him for opening up the avenue towards research and for his commitment to mentoring me for more than 6 years. He has inspired me to reason and think creatively. He certainly has a huge role in shaping me into an independent researcher. I would like to thank my co-advisor David Black-Schaffer for helping me with my research through constructive feedback and well-directed discussions. He has helped me to think about my research ideas more clearly and evaluate and present them in a very structured way. His influence will have a significant impact on the way I evaluate and present my ideas. I would like to thank my co-advisor Bengt Jonsson for being very supportive and committed to my research plan and goals throughout my PhD. I would thank Stefanos Kaxiras for his support and his attitude to inspiring novel research ideas among all members in the group. He has given good advice on several of my research ideas and other issues.

I am proud that today my research contributes to some interesting topics in the "Advanced Computer Architecture" class at Uppsala University. The same class I once attended as a student. My research has contributed a significant new branch to UART's (Uppsala Architecture Research Team) research expertise and I hope that it will influence good research that comes out of this group in future.

Many others have contributed directly or indirectly to my research efforts. I would like to thank all who have been a member of UART since I joined. David Eklöv for insightful discussions and critical approach to research. Andreas Sandberg and Andreas Sembrant for interesting collaborations. Nikos Nikoleris and I shared the office space for quite some time, we have had many interesting and helpful discussions all along. Pan for keeping me up to date on whats going on around and of course for the interesting research discussions. Kostas (Konstantinos Koukos) for fun discussions about prefetching and life; at some point, life and prefetching meant the same for both of us. Its different now. Vasilis for great insightful research discussions; his constant calm through all stages of research is an example for all. Him never being irate with research is still a mystery to me. Mahdad for interesting discussions on coherence and FPGAs. Magnus Själander for his useful insights

into implementation level details about the processor and the memory hierarchy. The young blood in UART – Moncef, Ricordo and German for great research discussions while keeping the environment lively. Alberto Ross for fun at all times, whether it be research, partying, traveling, conferences etc.; it has always been fun around him. Alexandra Jimborean for many interesting research and non-research discussions. Trevor has definitely added to both research and fun at UART. His insights in computer architecture research have helped all in UART.

I pay my respects to late Ivan Christoff. He had been a personal mentor and a good friend ever since I came to Uppsala. He gave me his unparalleled support in achieving my research and academic goals. Its sad to see him gone when I have finally reached this point. Thank you Ivan.

In my ever growing list of people to thank, I can not forget my long-time friends from Pakistan who have always been very supportive. My parents and brother have obviously played a vital role, it goes without saying. My loving wife and my dearest kids have been a source of great comfort for me. The Pakistani community in Uppsala and friends in Stockholm have always been a source of comfort, and have played the role of a family. We have engaged in uncountable activities, ranging from Cricket, family barbecues, fishing, canoeing, parties, boating, bowling, sky-diving, table-tennis, start-ups and God knows what not. The so called "International Pakistani Students Uppsala World Championship", a premier Table Tennis league (in our mind), has always been a very refreshing event. Thank you, you are all very important to me. It also goes without saying that I wouldn't have survived a foreign land without this community, no Pakistani can.

Finally, as expression of my faith and tradition becomes harder in today's world, I have been fortunate to be part of an environment free of such issues. At UART I have never felt uncomfortable. In my final words I thank the Almighty for His favors and blessings despite my frailty.

References

[1] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[2] E. Berg and E. Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Int. Symposium on Performance Analysis of Systems and Software*, 2004.

[3] J. C. Beyler and P. Clauss. Performance Driven Data Cache Prefetching in a Dynamic Software Optimization System. In *Proc. Annual International Conference on Supercomputing (ICS)*, 2007.

[4] J. Collins, P. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2001.

[5] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[6] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*, 2009.

[7] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Prefetch-aware shared resource management for multi-core systems. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2011.

[8] D. Eklov and E. Hagersten. Statstack: Efficient modeling of lru caches. In *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[9] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 2006.

[10] T. Inagaki, T. Onodera, H. Komatsu, and T. Nakatani. Stride prefetching by dynamically inspecting objects. In *PLDI*, 2003.

[11] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell. Making data prefetch smarter: Adaptive prefetching on power7. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[12] M. Khan and E. Hagersten. Optimization study for multicores. In *Proc. Swedish Workshop on Multicore Computing (MCC)*, 2009.

[13] M. Khan and E. Hagersten. Resource conscious prefetching for irregular applications in multicores. In *Proc. International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (ICSAMOS)*, 2014.

[14] M. Khan, N. Nikoleris, and E. Hagersten. Investigating how simple software optimizations effect relative throughput scaling on multicores. Technical report, 2012.

[15] M. Khan, A. Sembrant, and E. Hagersten. Low overhead instruction-cache modeling using instruction reuse profiles. In *Proc. International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2012.

[16] M. Khan, A. Sandberg, and E. Hagersten. A case for resource efficient prefetching in multicores. In *Proc. International Conference on Parallel Processing (ICPP)*, 2014.

[17] M. Khan, A. Sandberg, and E. Hagersten. A case for resource efficient prefetching in multicores. In *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[18] M. Khan, M. A. Laurenzano, J. Mars, E. Hagersten, and D. Black-Schaffer. AREP: Adaptive resource efficient prefetching for maximizing multicore performance. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[19] M. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *Proc. Annual International Symposium on Microarchitecture (MICRO)*, 2014.

[20] J. Lee, H. Kim, and R. Vuduc. When Prefetching Works, When It Doesn't, and Why. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[21] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid: software prefetching in pointer- and call-intensive environments. In *Proc. Annual International Symposium on Microarchitecture (MICRO)*, 1995.

[22] F. Liu and Y. Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *Proc. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2011.

[23] S. J. Louis. CIGAR - Case Injected Genetic Algortihm. URL `http://www.cse.unr.edu/~sushil/class/gas/code/cigar/`. http://www.cse.unr.edu/ sushil/class/gas/code/cigar/.

[24] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *Proc. Annual International Symposium on Microarchitecture (MICRO)*, 2003.

[25] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2001.

[26] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[27] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn. Profile-Guided Post-Link Stride Prefetching. In *Proc. Annual International Conference on Supercomputing (ICS)*, 2002.

[28] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2004.

[29] J. Mars and R. Hundt. Scenario Based Optimization: A Framework for Statically Enabling Online Optimizations. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2009.

[30] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. Annual International Symposium on Microarchitecture (MICRO)*, MICRO-44, 2011.

[31] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 1970.

[32] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[33] X. Pan and B. Jonsson. A modeling framework for reuse distance-based estimation of cache performance. In *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[34] S. Pugsley, Z. Chishti, C. Wilkerson, P. fei Chuang, R. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *HPCA*, 2014.

[35] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[36] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2009.

[37] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proc. International Symposium on Computer Architecture (ISCA)*, 1999.

[38] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. In *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[39] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data Prefetching on the HP PA-8000. In *Proc. International Symposium on Computer Architecture (ISCA)*, 1997.

[40] A. Sembrant, D. Eklöv, and E. Hagersten. Efficient software-based online phase classification. In *Int. Symposium on Workload Characterization*, 2011.

[41] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase guided profiling for fast cache modeling. In *Int. Symposium on Code Generation and Optimization*, 2012.

[42] S. Srinath, O. Mutlu, H. Kim, and Y. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[43] Y. Wu. Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[44] Y. Wu, M. J. Serrano, R. Krishnaiyer, W. Li, and J. Fang. Value-Profile Guided Stride Prefetching for Irregular Code. In *Proc. International Conference on Compiler Construction (CC)*, 2002.

[45] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proc.*

International Symposium on Computer Architecture (ISCA), 2013.
[46] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous Memory Introspection. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2007.