Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses

Andreas Sandberg, David Eklöv and Erik Hagersten

Department of Information Technology
Uppsala University, Sweden
{andreas.sandberg, david.eklov, eh}@it.uu.se

***Abstract*—Contention for shared cache resources has been recognized as a major bottleneck for multicores—especially for mixed workloads of independent applications. While most modern processors implement instructions to manage caches, these instructions are largely unused due to a lack of understanding of how to best leverage them.**

This paper introduces a classification of applications into four cache usage categories. We discuss how applications from different categories affect each other's performance indirectly through cache sharing and devise a scheme to optimize such sharing. We also propose a low-overhead method to automatically find the best per-instruction cache management policy.

We demonstrate how the indirect cache-sharing effects of mixed workloads can be tamed by automatically altering some instructions to better manage cache resources. Practical experiments demonstrate that our software-only method can improve application performance up to 35% on x86 multicore hardware.

I. Introduction

The introduction of multicore processors has significantly changed the landscape for most applications. The literature has mostly focused on parallel multithreaded applications. However, multicores are often used to run several independent applications. Such *mixed workloads* are common in a wide range of systems, spanning from cell phones to HPC servers. HPC clusters often run a large number of serial applications in parallel across their physical cores. For example, parameter studies in science and engineering where the same application is run with different input data sets.

When an application shares a multicore with other applications, new types of performance considerations are required for good system throughput. Typically, the co-scheduled applications share resources with limited capacity and bandwidth, such as a shared last-level cache (SLLC) and DRAM interfaces. An application overusing any of these resources can degrade the performance of the other applications sharing the same multicore chip.

Consider a simple example: Application *A* has an active working set that barely fits in the SLLC, and application *B* makes a copy of a data structure much larger than the SLLC. When run together, *B* will use a large portion of the SLLC and will force *A* to miss much more often than when run in isolation. Fortunately, most libraries implementing memory copying routines, e.g. memcpy, have been hand-optimized and use special *cache-bypass* instructions, such as non-temporal reads and writes. On most implementations, these instructions will avoid allocation of resources in the SLLC and subsequently will not force any replacements of application *A*'s working set in the cache.

In the example above the use of cache bypass instructions may seem obvious, and hand-tuning a common routine, such as memcpy, may motivate the use of special assembler instructions. However, many common programs also have memory accesses that allocate data with little benefit in the SLLC and may slow down co-scheduled applications. Detecting such reckless use is beyond the capability of most application programmers, as is the use of assembly coding. Ideally, both the detection and cache-bypassing should be done automatically using existing hardware support.

Several software techniques for managing caches have been proposed in the past [1], [2], [3], [4]. However, most of these methods require an expensive simulation analysis. These techniques assume the existence of heavily specialized instructions [1], [4], or extensions to the cache state and replacement policy [3], none of which can be found in today's processors. Several researchers have proposed hardware improvements to the LRU replacement algorithm [5], [6], [7], [8]. In general, such algorithms tweak the LRU order by including additional predictions about future re-references. Others have tried to predict [9] and quantify [10] interference due to cache sharing.

In this paper, we propose an efficient and practical software-only technique to automatically manage cache sharing to improve the performance of mixed workloads of common applications running on existing x86 hardware. Unlike previously proposed methods, our technique does not rely on any hardware modifications and can be applied to existing applications running on commodity hardware. This paper makes the following contributions:

- We propose a scheme to classify applications according to their impact, and dependence, on the SLLC.
- We propose an automatic and low-overhead method to find instructions that use the SLLC recklessly and automatically introduce cache bypass instructions into the binary.
- We demonstrate how this technique can change the classification of many applications, making them better mixed workload citizens.
- We evaluate the performance gain of the applications and

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

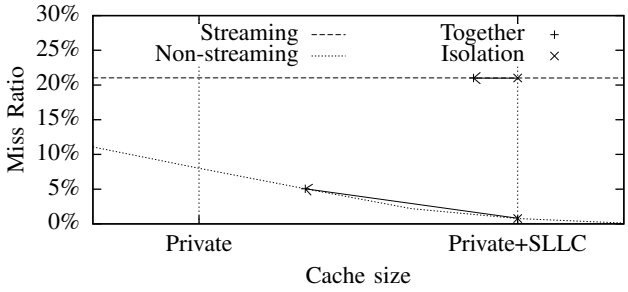


Figure 1. Miss ratio as a function of cache size for an application with streaming behavior and a typical non-streaming application that reuses most of its data. When run in isolation, each application has access to both the private cache and the entire SLLC. Running together causes the non-streaming application to receive a small fraction of the SLLC, while the streaming application receives a large fraction without decreasing its miss ratio. The change in perceived cache size and miss ratio is illustrated by the arrows.

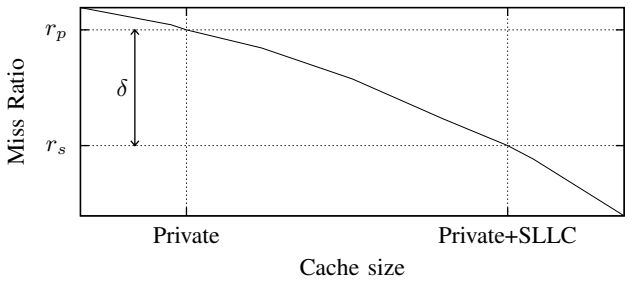


Figure 2. A generalized miss ratio curve for an application. The minimum, i.e. only the private cache, and the maximum, i.e. the private cache and the full shared cache, amount of cache available to an application are shown on the x-axis. The miss ratio when running in isolation (r_s) is the smallest miss ratio that an application can achieve on this system, while the miss ratio when running only in the private cache (r_p) is the worst miss ratio. The δ represents how much an application is affected by competition for the shared cache.

show that their improved behavior is in agreement with the classification.

II. Managing caches in software

Application performance on multicores is highly dependent on the activities of the other cores in the same chip due to contention for shared resources. In most modern processors there is no explicit hardware policy to manage these shared resources. However, there are usually instructions to manage these resources in software. By using these instructions properly, it is possible to increase the amount of data that is reused through the cache hierarchy. However, this requires being able to predict which applications, and which instructions, benefit from caching and which do not.

In order to know which application would benefit from using more of the shared cache resources, we need to know the applications' cache usage characteristics. The cache miss ratio as a function of cache size, i.e. the number of cache misses as a fraction of the total number of memory accesses as a function of cache size, is a useful metric to determine such characteristics. Figure 1 shows the miss ratio curves for a typical streaming application and an application that reuses its data. The miss ratio of the non-streaming application decreases as the amount of available cache increases. This occurs because more of the data set fits in the cache. Since the streaming application does not reuse its data, the miss ratio stays constant even when the cache size is increased.

When the applications are run in isolation, they will get access to both the core-local private cache and the SLLC. Assuming that the cache hierarchy is exclusive, the amount of cache available to an application running in isolation is the sum of the private cache and the SLLC. When two applications share the cache, they will perceive the SLLC as being smaller. In the case illustrated by Figure 1, the streaming application misses much more frequently than the non-streaming application. The frequent misses causes the streaming application to install more data in the cache than the non-streaming application. The non-streaming application will therefore perceive the cache as being much smaller than when run in isolation. The change in perceived cache size, and how this affects miss ratio is illustrated by the arrows in Figure 1.

Decreasing the perceived cache size for the streaming application does not affect its miss ratio. The non-streaming application, however, sees an increased miss ratio when access to the SLLC is restricted. As the number of misses increase, the bandwidth requirements also increase, which affects the performance of all the applications sharing the same memory interface. If we could make sure that the streaming application does not install any of its streaming data into the cache, the miss ratio, and bandwidth requirement, of the non-streaming applications would decrease without sacrificing any performance. In fact, the streaming application would run faster since the total bandwidth requirement would be decreased.

Using the miss ratio curves we can classify applications based on how they affect others and how they are affected by competition for the shared cache. We base this classification on the base miss ratio, r_s, when the application is run in isolation and has access to both its private cache and the entire SLLC, and the miss ratio, r_p, when it only has access to the private cache, see Figure 2. The r_p miss ratio can be thought of as the maximum miss ratio that an application can get due to cache contention, while r_s is the ideal case when the application is run in isolation. To capture the sensitivity to cache contention we define the cache sensitivity, δ, to be the difference between the two miss ratios. A large δ indicates that an application benefits from using the shared cache, while a small δ means that the application exhibits streaming behavior and does not benefit from additional cache resources.

Using the $\mathrm{r_s}$ and δ we can classify applications based on how they use the cache. This classification allows us to predict how applications will affect each other and how the system will be affected by software cache management. We define the following categories:

Don't care
: Small $\mathrm{r_s}$ and small δ—Applications that are largely unaffected by contention for the shared cache level.

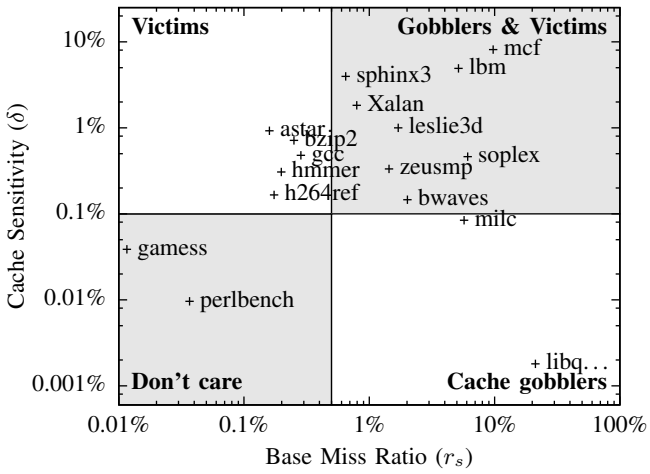


Figure 3. Classification map of a subset of the SPEC2006 benchmarks running with the reference input set on a system with a 576 kB private cache and 6 MB shared cache. The quadrants signify different behaviors when running together with other applications. Applications to the left tend to reuse almost all of their data in the shared cache and generally work well with other applications, applications to the right tend to use large parts of the shared cache for data that is never reused and are generally troublesome in mixes with other applications. Applications in the upper half are sensitive to the amount of data that can be stored in the shared cache, while applications on the bottom are insensitive.

These applications fit their entire data set in the private cache, they are therefore largely unaffected by contention for the shared cache and memory bandwidth.

Victims

Small $\mathrm{r_s}$ and large δ—Applications that suffer badly if the amount of cache at the shared level is restricted. The data they manage to install in the shared resource is almost always reused. Applications with a working set larger than the private cache, but smaller than the total cache size belong in this category.

Gobblers & Victims

Large $\mathrm{r_s}$ and large δ—Applications that suffer from SLLC cache contention, but store large amounts of data that is never reused in the shared cache. For example, applications traversing a small and a large data structure in parallel may reuse data in the cache when accessing the small structure, while accesses to the large data structure always miss. Disabling caching for the accesses to the large data structure would allow more of the smaller data structure to be cached. Managing the cache for these applications is likely to improve throughput, both when they are running in isolation and in a mix with other applications.

Cache Gobblers

Large $\mathrm{r_s}$ and small δ—Applications that do not benefit from the shared cache at all, but still install large amounts of data in it. Applications in this category work on streaming data or data structures that are much larger than the cache. These applications are good candidates for software cache management. Since they do not reuse the data they install in the shared cache, their throughput is generally not improved when running in isolation. Managing these applications will improve the full system throughput by allowing applications from other categories to use more of the shared cache.

Figure 3 shows the classification of several SPEC2006 benchmarks according to these categories. Applications classified as wasting cache resources, i.e. applications on the right-hand side of the map, are obvious targets for cache management. The large base miss ratio in such applications is due to memory accesses that touch data that is never reused while it resides in the cache. Disabling caching for such instructions does not introduce new misses since data is not reused, instead it will free up cache space for other accesses.

III. Cache management instructions

Most modern instruction sets include instructions to manage caches. These instructions can typically be classified into three different categories: *non-temporal memory accesses*, *forced cache eviction* and *non-temporal prefetches*. Many processors support at least one of these instruction classes. However, their semantics may not always make them suitable for cache management for performance.

Examples from the first category are the memory accesses in the PA-RISC which can be annotated with caching hints, e.g. only spatial locality or write only. Similar instruction annotations exist for Itanium. Other instruction sets, such as some of the SIMD extensions to the x86, contain completely separate instructions for handling non-temporal data. The hardware may, based on these hints, decide not to install write-only cache lines in the cache and use write-combining buffers instead. Non-temporal reads can be handled using separate non-temporal buffers or by installing the accessed cache line in such a way that it is the next line to be evicted from a set.

Instructions from the second category, *forced cache eviction*, appear in some form in most architectures. However, not all architectures expose such instructions to user space. Yet other implementations may have undesired semantics that limit their usefulness in code optimizations, e.g. the x86 *Flush Cache Line* (`CLFLUSH`) instruction forces *all* caches in a coherence domain to be flushed. There are some architectures that implement instructions in this class that are specifically intended for code optimizations. For example, the Alpha ISA specifies an instruction, *Evict Data Cache Block* (`ECB`), that gives the memory system a hint that a specific cache line will not be reused in the near future. A similar instruction, *Write Hint* (`WH64`), tells the memory subsystem that an entire cache line will be overwritten before being read again, this allows the memory system to allocate the cache line without actually reading its old contents. The `ECB` and `WH64` instructions are in many ways similar to the caching hints in the previous category, but instead of annotating the load or store instruction,

the hints are given after or, in case of a store, before the memory accesses in question.

The third category, non-temporal prefetches, is also included in several different ISAs. The SPARC ISA has both read and write prefetch variants for data that is not temporally reused. Similar prefetch instructions are also available in both Itanium and x86. Some implementations may choose to prefetch into the cache such that the fetched line is the next to be evicted from that set; others may prevent the data from propagating from the L1 to a higher level in the cache hierarchy.

In the remainder of this paper, we will assume an architecture with a non-temporal hint that is implemented such that non-temporal data is fetched into the L1 cache, but never installed in higher levels. This is how the AMD system we target implement support for non-temporal prefetches.

IV. Low-overhead cache modeling

A natural starting point for modeling LRU caches is the *stack distance* [11]. A stack distance is the number of unique cache lines accessed between two successive memory accesses to the same cache line. It can be directly used to determine if a memory access results in a cache hit or a cache miss for a fully-associative LRU cache: if the stack distance is less than the cache size, the access will be a hit, otherwise it will miss. Therefore, the stack distance distribution enables the application's miss ratio to be computed for any given cache size, by simply computing the fraction of memory accesses with a stack distances greater than the desired cache size.

In this work, we need to differentiate between what we call backward and forward stack distance. Let A and B be two successive memory accesses to the same cache line. Suppose that there are S unique cache lines accessed by the memory accesses executed between A and B. Here, we say that A has a forward stack distance of S, and that B has a backward stack distance of S.

Measuring stack distances is generally very expensive. In this paper, we use StatStack [12] to estimate stack distances and miss ratios. StatStack is a statistical cache model that models fully associative caches with LRU replacement. Modeling fully associative LRU caches is, for most applications, a good approximation of the set associative pseudo LRU caches implemented in hardware. StatStack estimates an application's stack distances using only a sparse sample of the application's *reuse distances*, i.e. the number of memory accesses performed between two accesses to the same cache line. This approach to modeling caches has been shown to be several orders of magnitude faster than full cache simulation, and almost as accurate. The runtime profile of an application can be collected with an overhead of only 40% [13], and the execution time of the cache model is only a few seconds [12].

To understand how StatStack works, consider the access sequence shown in Figure 4. Here the arcs connect subsequent accesses to the same cache line, and represent the reuse of data. In this example, the second memory access to cache line A has a reuse distance of five, since there are five memory accesses executed between the two accesses to A, and a backward

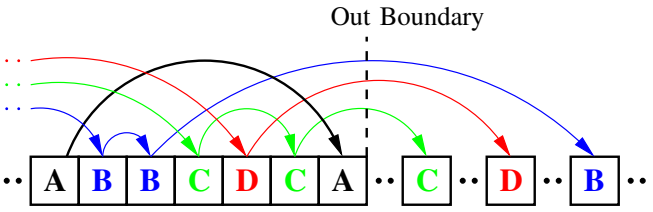


Figure 4. Reuse distance in a memory access stream. The arcs connect successive memory accesses to the same cache line, and represents the reuse of cache lines. The stack distance of the second memory access to A is equal to the number of arcs that cross "Out Boundary".

stack distance of three, since there are three unique cache lines (B, C and D) accessed between the two accesses to A. Furthermore, we see that there are three arcs that cross the vertical line labeled "Out Boundary", which is the same as the stack distance of the second access to A. This observation holds true in general. Based on it we can compute the stack distance of any memory access, given that we know the reuse distances of all memory access performed between it and the previous access to the same cache line.

The input to StatStack is a sparse reuse distance sample that only contains the reuse distances of a sparse random selection of an application's memory accesses, and therefore does not contain enough information for the above observation to be directly applied. Instead, StatStack uses the reuse distance sample to estimate the application's reuse distance distribution. This distribution is then used to estimate the likelihood that a memory access has a reuse distance greater than a given length. Since the length of a reuse distance determines if its outbound arc reaches beyond the "Out Boundary", we can use these likelihoods to estimate the stack distance of any memory access. For example, to estimate the stack distance of the second access to A in Figure 4, we sum the estimated likelihoods that the reuse distance of the memory accesses executed between the two accesses to A have reuse distances such that their corresponding arcs reach beyond "Out Boundary".

StatStack uses this approach to estimate the stack distances of all memory accesses in a reuse distance sample, effectively estimating a stack distance distribution. StatStack uses this distribution to estimate the miss ratio for any given cache size, C, as the fraction of stack distances in the estimated stack distance distribution that are greater than C.

V. Identifying non-temporal accesses

Using the stack distance profile of an application we can determine which memory accesses do not benefit from caching. We will refer to memory accessing instructions whose data is never reused during its lifetime in the cache hierarchy as *non-temporal memory accesses*.

If these non-temporal accesses can be identified, the compiler, a post processing pass, or a dynamic instrumentation engine can alter the application to use non-temporal instructions in these locations without hurting performance.

The system we model implements a non-temporal hint that causes a cache line to be installed in the L1, but never in any of

the higher cache levels. It turns out that modeling this system is fairly complicated, we will therefore describe our algorithm to find non-temporal accesses in three steps. Each step adds more detail to the model and brings it closer to the hardware. A fourth step is included to take effects from sampled stack distances into account.

A. A first simplified approach

By looking at the forward stack distances of an instruction we can easily determine if the next access to the data used by that instruction will be a cache miss, i.e. the instruction is non-temporal. An instruction has non-temporal behavior if all forward stack distances, i.e. the number of unique cache lines accessed between this instruction and the next access to the same cache line, are larger or equal to the size of the cache. In that case, we know that the next instruction to touch the same data is very likely to be a cache miss. Therefore, we can use a non-temporal instruction to bypass the entire cache hierarchy for such accesses.

This approach has a major drawback. Most applications, even purely streaming ones that do not reuse data, may still exhibit short temporal reuse, e.g. spatial locality where neighboring data items on the same cache line are accessed in close succession. Since cache management is done at a cache line granularity, this clearly restricts the number of possible instructions that can be treated as non-temporal.

B. Refining the simple approach

Most hardware implementations of cache management instructions allow the non-temporal data to live in parts of the cache hierarchy, such as the L1, before it is evicted to memory. We can exploit this to accommodate short temporal reuse of cache lines. We assume that whenever a non-temporal memory access touches a cache line, the cache line is installed in the MRU-position of the LRU stack, and a special bit on the cache line, the *evict to memory* (ETM) bit, is set. Whenever a normal memory access touches a cache line, the ETM bit is cleared. Cache lines with the ETM bit set are evicted earlier than other lines, see Figure 5. Instead of waiting for the line to reach the depth d_{max} it is evicted when it reaches a shallower depth, d_{ETM}. This allows us to model implementations that allow non-temporal data to live in parts of the memory hierarchy. For example, the memory controller in our AMD system evicts ETM tagged cache lines from the L1 to main memory, and would therefore be modeled with d_{ETM} being the size of the L1 and d_{max} the total combined cache size.

The model with the ETM bit allows us to consider memory accesses as non-temporal even if they have short reuses that hit in the small ETM area. Instead of requiring that all forward stack distances are larger than the cache size, we require that there is at least one such access and that the number of accesses that reuse data in the area of the LRU stack outside the ETM area, the gray area in Figure 5, is small, i.e. the number of misses introduced if the access is treated as non-temporal is small. We thus require that one stack distance is greater or equal to d_{max}, and that the number of stack distances that are larger or equal to d_{ETM} but smaller than d_{max} is smaller than some threshold, t_m. In most implementations t_m will not be a single value for all accesses, but depend on factors such as how many additional cache hits can be created by disabling caching for a memory access.

The hardware we want to model does not, unfortunately, reset the ETM bit when a temporal access reuses ETM data. This new situation can be thought of as sticky ETM bits, as they are only reset on cache line eviction.

C. Handling sticky ETM bits

When the ETM bit is retained for the cache lines' entire lifetime in the cache, the conditions for a memory accessing instruction to be non-temporal developed in section V-B are no longer sufficient. If instruction *X* sets the ETM bit on a cache line, then the ETM status applies to all subsequent reuses of the cache line as well. To correctly model this, we need to make sure that the non-temporal condition from section V-B applies, not only to *X*, but also to all instructions that reuse the cache lines accessed by *X*.

The sticky ETM bit is only a problem for non-temporal accesses that have forward reuse distances less than d_{ETM}. For example, consider a memory accessing instruction, *Y*, that reuses the cache line previously accessed by a non-temporal access *X* (here *Y* is a cache hit). When *Y* accesses the cache line it is moved to the MRU position of the LRU stack, and the sticky ETM bit is retained. Now, since *Y* would have resulted in a cache hit no matter if *X* had set the sticky ETM bit or not, this is the same as if we would have set the sticky ETM bit for the cache line when it was accessed by *Y*.

Therefore, instead of applying the non-temporal condition to a single instruction, we have to apply it to all instructions reusing the cache line accessed by the first instruction.

In a machine, such as our AMD system, where d_{ETM} corresponds to the L1 cache, this new condition allows us to categorize a memory access as non-temporal if all the data it touches is reused through the L1 cache or misses in the entire cache hierarchy. Due to the stickiness of the non-temporal status, this condition must also hold for any memory access that reuses the same data through the L1 cache.

D. Handling sampled data

To avoid the overhead of measuring exact stack distances, we use StatStack to calculate stack distances from sampled reuse distances. Sampled stack distances can generally be used in place of a full stack distance trace with only a small decrease in *average* accuracy. However, there is always a risk of missing some critical behavior. This could potentially lead to flagging an access as non-temporal, even though the instruction in fact has some temporal behavior in some cases, and thereby introducing an unwanted cache miss.

In order to reduce the likelihood of introducing misses due to sampling, we need to make sure that flagging an instruction as non-temporal is always based on reliable data. We do this by introducing a sample threshold, t_s, which is the smallest number of samples originating from an instruction that can be considered to be non-temporal.

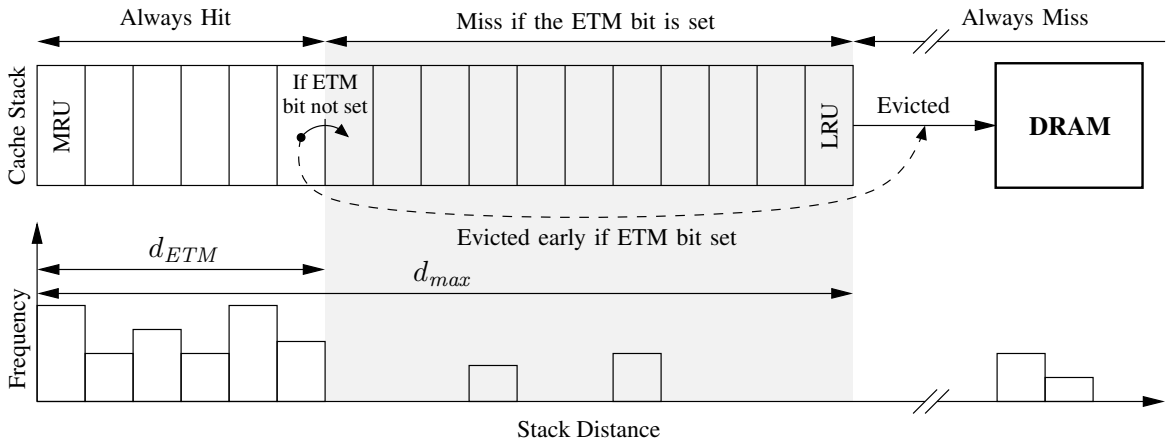


Figure 5. LRU stack (top) and the forward stack distance distribution of a memory accessing instruction (bottom). If the ETM bit is set the cache lines are evicted early to DRAM when they reach d_{ETM}. The bars within the shaded area of the forward stack distances distribution represent memory accesses that will result in cache misses if the ETM bit is set. An instruction is classified as non-temporal if there are less than t_m forward stack distances between d_{ETM} and d_{max} and at least one forward stack distance greater than d_{max}.

Table I
CACHE PROPERTIES OF THE MODEL SYSTEM (AMD PHENOM II X4 920)

Level	Size (kB)	Associativity	Line Size (B)	Shared
1 (data)	64	2	64	No
2	512	16	64	No
3	6144	48	64	Yes

VI. EVALUATION METHODOLOGY

A. Model system

To evaluate our model we used an x86 based system with an AMD Phenom II X4 920 processor with the AMD family 10h micro-architecture. The processor has 4-cores, each with a private L1 and L2 cache and a shared L3 cache. The processor enforces exclusion between L1 and L2, but not always between L3 and the lower levels if data is shared between cores.

According to the documentation of the `prefetchnta` instruction, data fetched using the non-temporal prefetch is not installed in the L2 unless it was fetched from the L2 in the first place. However, our experiments show that this is not the case. It turns out that data fetched from the L2 cache using the non-temporal prefetch instruction is never re-installed in the L2. The system therefore works like the system modeled in section V-C where the ETM-bit is sticky.

We used the performance counters in the processor to measure the cycles and instruction counts using the perf framework provided by recent Linux kernels.

B. Benchmark preparation

The benchmarks were first compiled normally for initial reference runs and sampling. Sampling was done on each benchmark running with the reference input set. Due to the low overhead of the sampler, the benchmarks were run to completion with the sampler attached throughout the entire run. After the initial profile run, we analyzed the profile using the algorithm in the previous section and generated a list of non-temporal memory accesses. The benchmarks were then recompiled taking this profile into account.

The cache managed versions of the benchmarks were compiled using a compiler wrapper script that hooked into the compilation process before the assembler was called. The assembly output was then modified before it was passed to the assembler. Using the debug information from the binary we were able to find the memory accesses in the assembly output corresponding the instruction addresses in the non-temporal list. Before each non-temporal memory access the script inserted a `prefetchnta` instruction to the same memory location as the original access.

C. Algorithm parameters

We model the cache behavior of our benchmarks using StatStack and a reuse distance sample with 100 000 memory access pairs per benchmark. We use a minimum samples threshold, t_s, of 50 samples. The maximum number of introduced misses, t_m, is set to 0 samples; this may seem strict at first, but remember that we are sampling memory accesses and one sample corresponds to several hundred thousand memory accesses.

Cache exclusivity guarantees that there is at most one copy of a cache line in the caches where exclusivity is enforced. For example, an access to a cache line that resides in the L2 of our system will cause that cache line to be removed from the L2 and installed in the L1. A system where cache exclusivity is not enforced would not remove the copy in the L2. When the cache line is evicted from the L1 cache it is installed in the L2 cache, i.e. it is transferred from the LRU position of the L1 to the MRU position of the L2. This behavior lets us merge the two caches and treat them as one larger LRU stack where each cache level corresponds to a contiguous section of the stack. In the model system, the first 1k lines correspond to the L1 cache, the next 8 k lines correspond to the L2 and the last 96 k lines correspond to the L3. This global stack has

$105\,\mathrm{k}$ lines in total, i.e. the total cache size in lines. We let d_{max} be the depth of this global stack.

Since we are using StatStack we have made the implicit assumption that caches can be modeled to be fully associative, i.e. conflict misses are insignificant. In most cases this is a valid assumption, especially for large caches with a high degree of associativity. A notable case where this assumption may break is for the L1 cache, which has a low degree of associativity. We therefore have to be more conservative when evaluating stack distances within this range. We use different, conservative, values of d_{ETM}, when calculating the number of introduced misses and handling the stickiness of the ETM bits. We use a d_{ETM} value of twice the L1 size, i.e. 2048 lines, when handling stickiness and half the L1 size, i.e. 512 lines, when calculating the number of misses introduced.

D. Benchmarks

Using the software classification introduced in section II we selected two benchmarks representing each category for analysis. The number of non-temporal memory accesses and the effect on other applications in the system will depend on a benchmark's position in the classification map. Applications on the left-hand side of the map, *Don't care* and *Victims*, do not install a significant amount of data in the shared cache and do not disturb other applications running on the system. As expected, our algorithm does not find any non-temporal memory accesses in such applications. Applications on the right-hand side of the map, *Gobblers & Victims* and *Cache Gobblers*, have a high base miss ratio and store a large amount of non-temporal data in the shared cache. We expect such applications to be good candidates for cache management.

There is normally no need to differentiate between cache misses and replacements. Whenever there is a cache miss, a new cache line is installed and another one is replaced. When we start to software manage the cache, we disable caching for certain instructions. This causes misses to occur, but, since the data is never cached, no replacements take place. When we classify managed applications, it therefore makes more sense to use the replacement ratio rather than the miss ratio to better capture the effect on other application.

We extended the StatStack algorithm to handle non-temporal memory accesses to calculate new replacement and miss ratios for managed applications. Looking at Figure 6a we see that libquantum's replacement ratio is reduced from approximately 20% to 0% in the shared cache, while the miss ratio stays at 20%. This can be explained by the fact that the instructions installing non-temporal data into the SLLC now bypass the cache. The fact that they bypass the cache leads to a decreased replacement ratio, i.e. fewer cache lines installed in the SLLC. We would normally expect the miss ratio to be decreased due to a reduction of non-temporal data in the SLLC, which would allow more temporal data to be stored instead. In the case of libquantum, there is no additional temporal data that can be squeezed into the cache.

When looking at cache sizes larger than d_{max}, another effect of software cache management is seen. As the data set starts to fit in the cache, the miss ratio is normally reduced. This may no longer be the case when applying cache management. When we manage the cache, we force certain accesses to bypass the cache and the reuse to become a miss, independent of cache size.

Lbm has slightly more interesting access patterns than libquantum, which includes a decrease in miss ratio around the target cache size. Looking at Figure 6b, we notice that, in addition to the features displayed by libquantum, parts of the miss ratio curve have been shifted to the left. This can be explained by the fact that removing non-temporal data from the cache allows more of the temporal data to fit in the cache.

We reclassify our benchmarks based on their new replacement ratio curves, the new classification allows us to predict how applications affect each other after we introduce the non-temporal memory accesses. The change in classification for the managed benchmarks is shown in Figure 7.

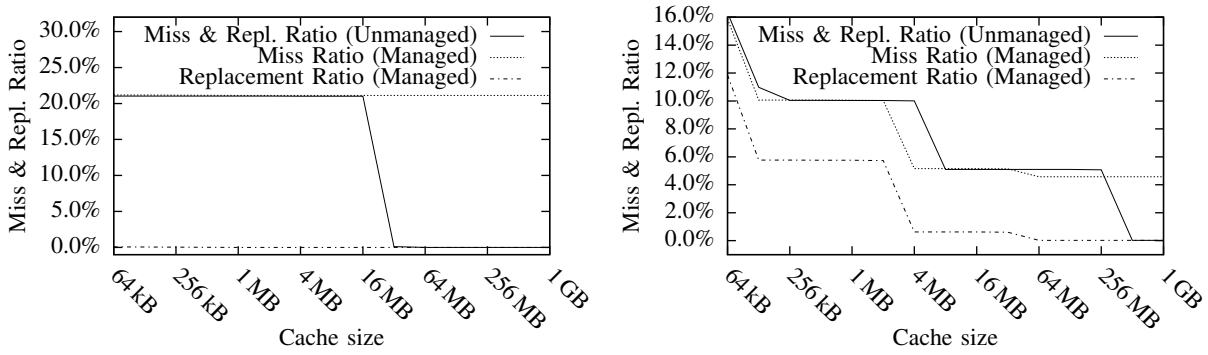
VII. Results and analysis

The results for runs of six different mixes of four SPEC2006 benchmarks running with the reference input set, with and without software cache management are shown in Figure 8 and Figure 9. Figure 8 shows a mix of four applications from different categories. Figure 9 shows five different mixes consisting of two pairs of benchmarks from different categories. The instructions per cycle, IPC, is shown for each of the benchmarks, both when running in isolation and when running in the mix and with and without cache management. The cache management instructions are not included in the instruction counts when calculating IPC for managed applications, including them would give an unfair advantage to the managed applications. The speedup is the improvement in IPC over the unmanaged version when running in a mix.

As seen by comparing the IPC for managed and unmanaged applications in isolation, Figure 8 and Figure 9, inserting additional `prefetchnta` instructions does not negatively impact performance in isolation. In fact, the IPC of LBM is increased by approximately 12%. This effect can be explained by Figure 6b, where a miss ratio knee is clearly visible between $4\,\mathrm{MB}$ and $8\,\mathrm{MB}$. Applying software cache management pushes the knee to the left, i.e. towards smaller cache sizes, and decreases the miss ratio for systems with between $4\,\mathrm{MB}$ and $8\,\mathrm{MB}$ of cache.

Looking at Figure 8 and Figure 9 we see that all applications, except for the applications in the *Don't care* category, have a lower IPC when running in a mix than running in isolation. This is to be expected since running in a mix means that all the applications compete for a shared cache and shared bandwidth. Applications in the *Don't care* category fit most of their data in the private cache, which makes their bandwidth and SLLC requirements extremely small. This explains why this category does not benefit from cache management.

The difference between the *Victims* and the *Don't care* categories is that the former uses some amounts of L3 cache, while the latter does not. This would suggest that interference between these two categories should be small when running



(a) libquantum

(b) lbm

Figure 6. Miss and replacement ratio before and after cache managing the benchmarks to avoid caching of non-temporal data. The number of replacements can be reduced by cache management in both applications. The number of misses in (b) can be reduced, particularly around the target cache size, because a reduction in the number of replacements will allow more temporal data to fit in the cache. The miss ratio normally drops to 0% once the entire data set fits in the cache, this is no longer the case for managed applications, since the non-temporal memory accesses always cause a miss.

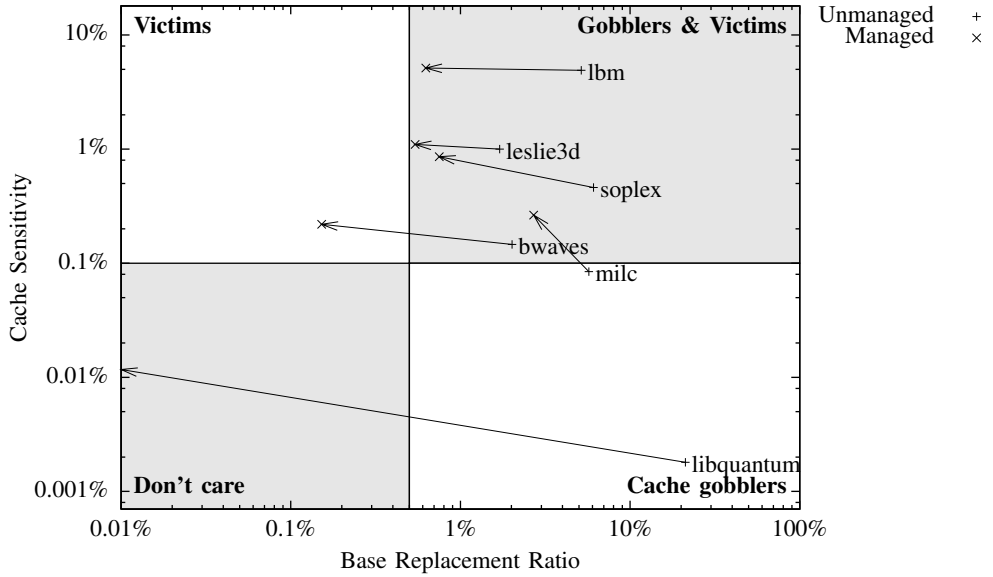


Figure 7. Changes in classification after disabling caching of non-temporal memory accesses. Note that this classification is based on the replacement ratio rather than the miss ratio.

together. This is supported by Figure 9d, where all applications in this mix run at the same speed as in isolation. There could still be some interference between applications within the *Victims* category, but this is likely to be very small since these applications have a small bandwidth and cache footprint.

Because applications in the *Cache Gobblers* and *Gobblers & Victims* categories have similar cache and bandwidth pressure they affect other applications in the same way. Looking at Figure 9a, Figure 9c and Figure 9e we see that running together with applications from these categories causes a significant decrease in IPC compared to when running in isolation. We expect that managing these categories reduces their cache footprint, and as a consequence reduce their impact on IPC. Our results indicate that some of the performance lost to contention for the shared resources can be regained using software cache management.

A somewhat surprising result might be that applications from the *Cache Gobblers* category benefit from cache management themselves. This is the case for all of the mixes, but is particularly visible in Figure 9b. The speedup when running with applications from the two victim categories can largely be attributed to a reduction in the total bandwidth requirement of the mix. The speedup when running together with the *Don't care*, Figure 9b, is harder to explain, but is likely due to small reductions in the miss ratio in the *Cache Gobblers*.

VIII. Related work

There has been plenty of research focused on improving cache efficiency. Most of this work has been targeting the miss ratio of private caches [1], [3], [4]. Focus has recently started to shift towards shared caches [6], [14]. These methods

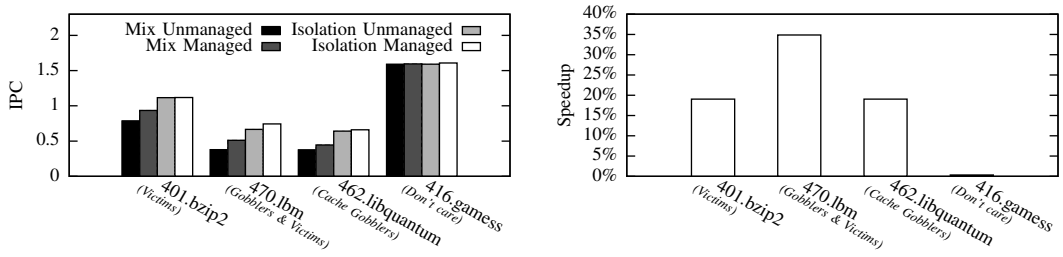


Figure 8. Performance for a mix of four applications, each from a different category. The IPC plot compares the IPC for managed and unmanaged benchmarks, both in a mix and in isolation. The speedup is relative to the unmanaged mix.

are either software driven or hardware driven. They all have one thing in common, the need to identify non-temporal data or, as in our case, memory accesses referencing non-temporal data. Once the non-temporal data is identified this information is propagated to the cache, typically by setting a non-temporal bit in the cache tags. This bit is then used by the cache replacement policy to explicitly handle non-temporal data.

Tyson *et al.* [1] propose a simulation based method to identify non-temporal memory accesses. It can be described in two steps: First, they simulate the cache and identify the instructions with miss ratios above a threshold (25%). Then, for each dynamic execution of these instructions, they keep track of how often the fetched cache line is accessed before being evicted. If this occurs less than 25% of the time the instruction is identified as being non-temporal. Our approach differs on the following key points: 1) It does not require expensive simulations; 2) It considers all instructions as potentially non-temporal, not only the ones with a miss ratio above a threshold. This increases the potential to reduce the caching of non-temporal data; 3) Our method is tailored to use existing hardware on the x86 architecture.

Wong *et al.* [3] propose a method to identify non-temporal memory accesses based on Mattson's optimal replacement algorithm (OPT) [11]. Their method is similar to ours in that it uses the forward stack distance distribution of a memory accessing instruction to determine if it is a temporal instruction. However, instead of using LRU stack distances, they use OPT stack distances, which requires expensive simulation.

Several hardware methods have been proposed [1], [3], [6], [14], that dynamically identify non-temporal data. Xie *et al.* [14] propose a replacement policy, PIPP, to effectively way-partition a shared cache, that explicitly handles non-temporal (streaming) data. To detect non-temporal data, they introduce a set of shadow tags [15] used to count the number of hits to a cache line that would have occurred if the thread was allocated all ways in the cache set. Similarly to Tyson's method [1], they identify a cache line as non-temporal if there are no accesses to it prior to its eviction. This approach is rather course grained, during a time period when the majority of the data accessed by a thread is non-temporal it assumes that all data accessed by the thread is non-temporal.

Qureshi *et al.* [5] propose an insertion policy (DIP) where on a cache miss to non-temporal data it is installed in the LRU position, instead of the MRU position of the LRU stack. To detect non-temporal data they use two sets of training cache sets. In the first training set, data is installed in the LRU position, and in the other data is installed in the MRU position. The rest of the cache sets use the insertion policy of the training set that currently has the highest hit ratio. This method has been extended [6] to be thread aware (TADIP), by using separate training sets and insertion policies (insertion in LRU or MRU) for the different threads. Both DIP and TADIP exhibit the same course grained time varying behavior as PIPP. A recent extension [8] introduces an additional policy that installs cache lines in the $\mathrm{MRU} - 1$ position.

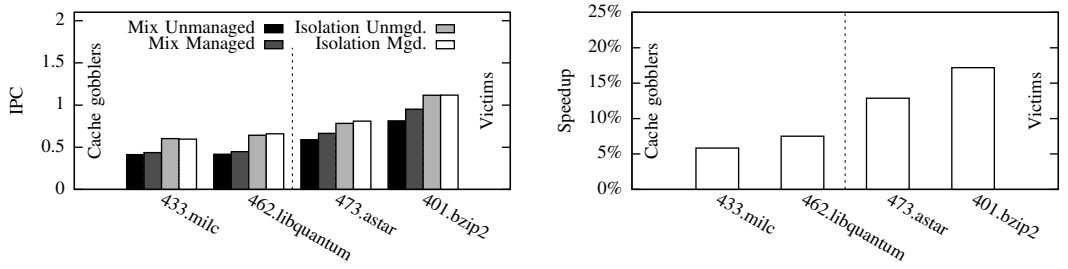
Petoumenos *et al.* [7] propose an instruction based reuse distance predictor and a replacement algorithm based on the predicted reuse distances. Their algorithm approximates the optimal algorithm by replacing the cache line that is predicted to be reused furthest into the future.

The time varying behavior of PIPP, DIP and TADIP can be effective to handle the time varying behavior of applications (program phases). However, for applications whose instruction working set is different between program phases, the static classification of memory instructions, used in this and other papers, allows for a more fine grained control, while at the same time following the time varying behavior of the applications.

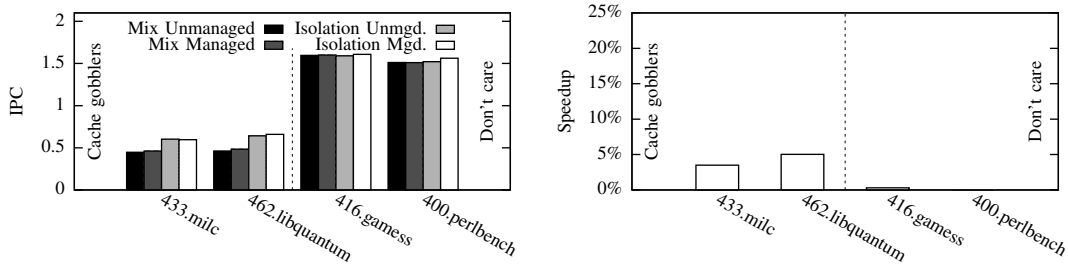
IX. Summary and future work

We describe an application classification framework that allows us to predict how applications affect each other when running on a multicore and a method for finding non-temporal memory accesses. Using a single low-overhead profile run of an application, we can acquire enough information to both classify the application and find non-temporal memory accesses for any combination of shared and private cache sizes. Our method can be used together with contemporary hardware to provide a speedup for existing applications. We show that this is the case for a selection of the SPEC2006 benchmarks. Using a modified StatStack implementation we can reclassify applications based on their replacement ratios after applying cache management, this allows us to reason about how cache management impacts performance.

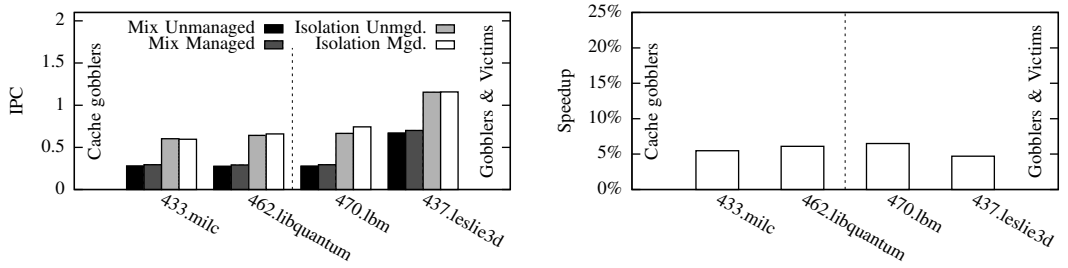
Future work will explore other hardware mechanism for handling non-temporal data hints from software and possible applications in scheduling.



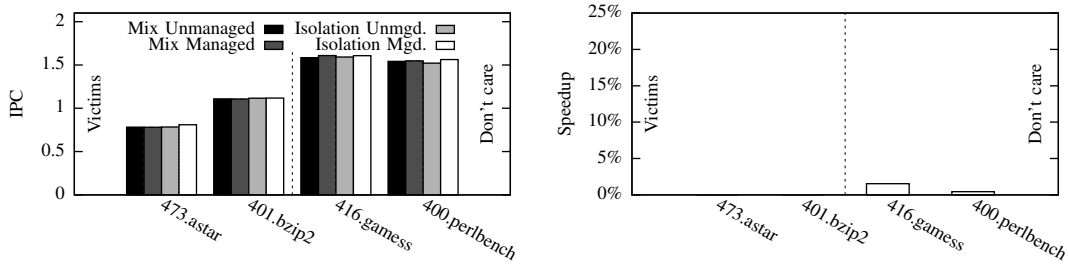
(a) *Cache Gobblers ↔ Victims*



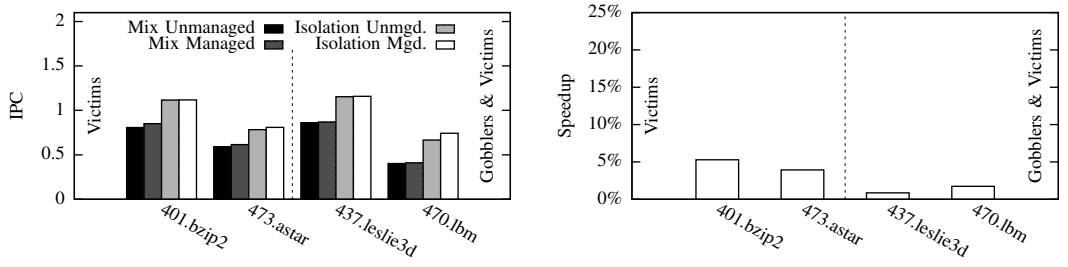
(b) *Cache Gobblers ↔ Don't care*



(c) *Cache Gobblers ↔ Gobblers & Victims*



(d) *Victims ↔ Don't care*



(e) *Victims ↔ Gobblers & Victims*

Figure 9. Performance of mixes with benchmarks from two different categories. Benchmarks from different categories are separated by a dotted line. All of the benchmarks, except for the *Don't care* category, generally run slower in mixes than in isolation. Disabling caching for non-temporal memory accesses regains some of the IPC lost to cache and bandwidth contention without any negative impact on application performance in isolation.

ACKNOWLEDGMENTS

The authors would like to thank Kelly Shaw and David Black-Schaffer for valuable comments and insights that has helped to improve this paper. This work was financially supported by the CoDeR-MP and UPMARC projects.

REFERENCES

[1] G. Tyson, M. Farrens, J. Matthews, and A. Pleszkun, "A Modified Approach to Data Cache Management," in *Microarchitecture, 1995. Proceedings of the 28th Annual International Symposium on*, 1995, pp. 93–103.

[2] T. Sherwood, B. Calder, and J. Emer, "Reducing Cache Misses Using Hardware and Software Page Placement," in *Proceedings of the 13th international conference on Supercomputing*. Rhodes, Greece: ACM, 1999, pp. 155–164.

[3] W. Wong and J. Baer, "Modified LRU Policies for Improving Second-Level Cache Behavior," in *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, 2000, pp. 49–60.

[4] Z. Wang, K. McKinley, A. Rosenberg, and C. Weems, "Using the Compiler to Improve Cache Replacement Decisions," in *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, 2002, pp. 199–208.

[5] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *Proceedings of the 34th annual international symposium on Computer architecture*. San Diego, California, USA: ACM, 2007, pp. 381–391.

[6] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, J. S. Steely, and J. Emer, "Adaptive Insertion Policies for Managing Shared Caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. Toronto, Ontario, Canada: ACM, 2008, pp. 208–219.

[7] P. Petoumenos, G. Keramidas, and S. Kaxiras, "Instruction-based Reuse-Distance Prediction for Effective Cache Management," in *Proceedings of the 9th international conference on Systems, architectures, modeling and simulation*. Samos, Greece: IEEE Press, 2009, pp. 49–58.

[8] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2010, pp. 60–71.

[9] Y. Xie and G. H. Loh, "Dynamic Classification of Program Memory Behaviors in CMPs," in *Proc. of CMP-MSI*, Jun. 2008.

[10] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing Shared L2 Caches on Multicore Systems in Software," in *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architectur*, San Diego, California, USA, 2007.

[11] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques in storage hierarchies," *IBM Journal of Research and Development*, vol. 9, no. 2, pp. 78–117, 1970.

[12] D. Eklöv and E. Hagersten, "StatStack: Efficient Modeling of LRU Caches," in *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2010)*, New York, New York, USA, Mar. 2010.

[13] E. Berg and E. Hagersten, "Fast Data-Locality Profiling of Native Execution," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 169–180, 2005.

[14] Y. Xie and G. H. Loh, "PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 174–183, 2009.

[15] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432.