A Software Technique for Reducing Cache Pollution

Andreas Sandberg
Department of Information Technology
Uppsala University, Sweden
andreas.sandberg@it.uu.se

David Eklöv
Department of Information Technology
Uppsala University, Sweden
david.eklov@it.uu.se

Erik Hagersten
Department of Information Technology
Uppsala University, Sweden
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.

We propose an automatic, low-overhead, method to reduce cache contention by finding instructions that are prone to cache trashing and a method to automatically disable caching for such instructions. Practical experiments demonstrate that our software-only method can improve application performance up to 35% on x86 multicore hardware.

1. 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.

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 a working set that barely fits in the SLLC, and application B traverses 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 than when run in isolation.

Several software techniques for managing caches have been proposed in the past [11, 9, 13, 12]. However, most of these methods require an expensive simulation. These techniques assume the existence of heavily specialized instructions [11, 12], or extensions to the cache state and replacement policy [13], none of which can be found in today's processors. Several researchers have proposed improvements to the LRU replacement algorithm [7, 3, 6, 4]. In general, such algorithms tweak the LRU order by including additional predictions about future re-references. Others have tried to predict [14] and quantify [10] interference due to cache sharing.

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.

2. LOW-OVERHEAD CACHE MODELING

A natural starting point for modeling LRU caches is the *stack distance* [5]. 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.

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 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. We use StatStack [2] to estimate stack distances and miss ratios. 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% [1].

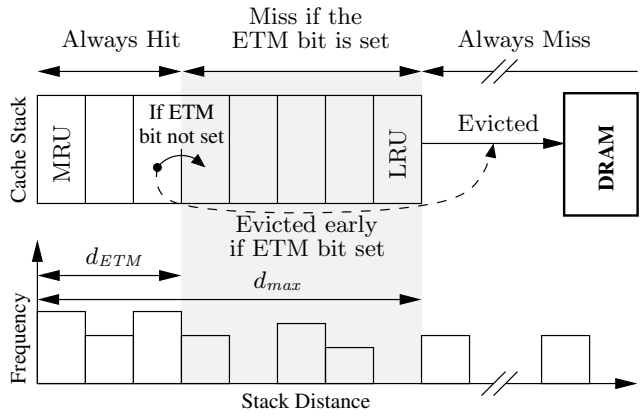

Figure 1: LRU stack (top) and the forward stack distance distribution of a memory accessing instruction (bottom). Cache lines are evicted earlier than normal if the ETM bit is set. Stack distances within the shaded area are memory accesses that will result in misses if the ETM bit is set.

3. CLASSIFYING MEMORY 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*. We use a use a compiler post-processing step to disable caching for such memory accesses, which has the effect of eliminating cache pollution caused by those instructions.

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. We will describe our algorithm to find non-temporal accesses in four steps. Each of the first three steps adds more detail to the model and brings it closer to the hardware, the fourth step is included to take effects from sampled stack distances into account.

3.1 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. An instruction has non-temporal behavior if all forward stack distances are larger or equal to the size of the cache. In that case, we know that the next instruction to touch the same data will be a cache miss.

This approach has a major drawback. Most applications, even purely streaming ones that do not reuse data, usually reuse cache lines. 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.

3.2 Refining the simple approach

Most hardware implementations of cache management instructions allow 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. We assume that whenever a non-temporal memory access touches a cache line, it 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 1. Instead of waiting for the line to reach the depth d_{max} it is evicted when it reaches d_{ETM}.

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 1, 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.

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.

3.3 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 the previous section are no longer sufficient. If instruction X sets the ETM bit on a cache line, 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 3.2 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 shorter than d_{ETM}. For example, consider an instruction, Y, that reuses a cache line previously accessed by a non-temporal access X (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. Since the ETM bit is still set after Y has executed, the state of the cache is the same as if Y would have set the ETM bit. Therefore, we have to apply the non-temporal conditions to all instructions reusing the cache line accessed by X.

3.4 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 introduce a sample threshold, t_s, which is the smallest number of samples originating from an instruction that can be considered to be non-temporal.

Table 1: 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

4. EVALUATION METHODOLOGY

4.1 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 3.3 where the ETM-bit is sticky.

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

4.2 Benchmark preparation

The benchmarks were first compiled normally for initial reference runs. Reuse distance 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 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.

4.3 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 1 k 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 k lines in total, i.e. the total cache size in lines. We let d_{max} be the depth of the 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.

5. RESULTS AND ANALYSIS

The results for runs of three mixes of four SPEC2006 benchmarks each are shown in Table 2. 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.

As seen by comparing the IPC for managed and unmanaged applications in isolation, inserting additional `prefetchnta` instructions does not negatively impact performance in isolation in most cases. In fact, the IPC of LBM is increased by almost 12%. This effect is to be expected for certain classes of applications where additional data can be reused by removing non-temporal data from the cache. This situation is in many ways similar to having two applications with different cache behavior competing for cache resources, but instead having different applications competing, different parts of the same application compete for shared resources.

It's worth noting that all benchmarks are not equally affected by competition for the shared cache. For example, Gamess and Perlbench are both insensitive to competition for the shared resource. A likely explanation for this is that both benchmarks have a relatively small working set that fits well within the private caches. Benchmarks which ex-

Table 2: Benchmark performance

		IPC Mix			IPC Isolation		
		Unmanaged	Managed	Speedup	Unmanaged	Managed	Speedup
Mix 1	401.bzip2	0.78	0.93	19.0%	1.12	1.12	0.1%
	470.lbm	0.38	0.51	34.8%	0.67	0.74	11.6%
	462.libquantum	0.37	0.44	19.0%	0.64	0.66	2.8%
	416.gamess	1.59	1.59	0.3%	1.59	1.61	1.0%
Mix 2	433.milc	0.45	0.46	3.5%	0.60	0.60	−1.2%
	462.libquantum	0.46	0.48	5.0%	0.64	0.66	2.8%
	416.gamess	1.60	1.60	0.3%	1.59	1.61	1.0%
	400.perlbench	1.51	1.51	−0.1%	1.52	1.56	2.7%
Mix 3	433.milc	0.28	0.29	5.4%	0.60	0.60	−1.2%
	462.libquantum	0.27	0.29	6.0%	0.64	0.66	2.8%
	470.lbm	0.28	0.29	6.4%	0.67	0.74	11.6%
	437.leslie3d	0.67	0.70	4.7%	1.15	1.16	0.3%

perience a larger change IPC due to competition generally have a working set that is slightly smaller than the available cache when running in isolation. One notable exception is libquantum, which has a working set that is much larger than the available cache. In this case, most of the speedup in the mix runs can likely be attributed to a decrease in bandwidth usage from the rest of the mix. An in-depth discussion of how applications affect each other and how this can be changed by software cache management is provided in [8].

6. REFERENCES

[1] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. *SIGMETRICS Perform. Eval. Rev.*, 33(1):169–180, 2005.

[2] 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.

[3] 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*, pages 208–219, Toronto, Ontario, Canada, 2008. ACM.

[4] 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*, pages 60–71, New York, NY, USA, 2010. ACM.

[5] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques in storage hierarchies. *IBM Journal of Research and Development*, 9(2):78–117, 1970.

[6] 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*, pages 49–58, Samos, Greece, 2009. IEEE Press.

[7] 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*, pages 381–391, San Diego, California, USA, 2007. ACM.

[8] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, New Orleans, LA, USA, Nov. 2010. *(to appear)*.

[9] 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*, pages 155–164, Rhodes, Greece, 1999. ACM.

[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] 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*, pages 93–103, 1995.

[12] 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*, pages 199–208, 2002.

[13] 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*, pages 49–60, 2000.

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