Timestamp-Based Selective Cache Allocation

Martin Karlsson and Erik Hagersten
Dept. of Information Technology
Uppsala University
P.O. Box 325, SE-751 05 Uppsala, Sweden
Email: {martink, eh}@it.uu.se

Abstract

The behavior of the memory hierarchy is key to high performance in today's GHz microprocessors. The cache level closest to the processor is limited in size and associativity in order to match the short cycle time of the CPU. Even though only data objects reused soon again will benefit from the small cache, all accessed data objects are normally allocated in the cache.

In this paper we demonstrate how an "optimal" selective allocation algorithms, based on knowledge about the future, can drastically increase the effectiveness of a cache. The effectiveness is further enhanced if the allocation candidates are temporarily held in a small staging cache before making the allocation decision. We also present an implementable selective allocation algorithm based on knowledge about the past (RASCAL) which measures re-use distance in the new time unit Cache Allocation Ticks*, CAT. CAT is shown to be a fairly accurate and application-independent way of detecting good allocation candidates.*

0.1 Introduction

Cache systems are designed to minimize the average access time for memory references. Uniprocessor cache misses can be classified into the three categories: conflict, compulsory and capacity misses [Hil87]. The amount of conflict misses can be reduced by a more associative cache, or by the introduction of a victim cache [Jou90]. Larger cache lines and a number of prefetching algorithms have been proposed to reduce compulsory misses, while the conventional approach for reducing capacity misses is simply to increase the size of the cache - a brute force approach often enabled by

a manufacturing process shrink. However, at the lower levels of the cache hierarchy, a larger cache may not be feasible, since the cache size can be limited by the speed requirements of the CPU. The access time of the L1 is often tied to the pipeline architecture such that a larger and slower L1 cache would effectively slow down the CPU pipeline.

The introduction of the VIPT, Virtually Indexed Physically Tagged, scheme [WBL89], that removes the TLB lookup from the critical path, also limits the cache size. Aliasing problems arise if the L1 cache size is larger than the page size.[1] Sometimes this is circumvented by a more associative cache, but there is also a limit to the degree of associativity achievable in the fast L1. Agarwal et al. predict that due to advances in chip technology the CPU performance will be bound by communication constraints rather than by capacity limitations [AHKB00]. They predict the number of SRAM bits reachable in one CPU cycle to decrease over time – yet another negative impact for the L1 cache size. Chip Multiprocessors (CMP) with several CPUs, each with its own L1 cache, sharing the same die is another reason to keep the L1 caches small.

We conclude that the first-level caches are likely to remain small relative to the active working set of most applications and that selective cache allocation should be studied. By a more selective L1 allocation, the data objects well suited for the L1 cache will reside longer in the L1 cache. This will increase the *effective cache* size of the L1 cache and remove some of the *capacity misses*.

The contributions of this paper is three-fold:

- We suggest streaming the data through a small staging cache before deciding about the L1 allocation and demonstrate its effect on an optimal allocation algorithm.
- We suggest a new time-stamp based allocation algorithm based on the new time unit *cache allocation ticks*, CAT.
- We compare three different implementation options for selective cache allocation.

The rest of this paper is outlined as follows. First, we discuss different selective allocation schemes and discuss their advantages and drawbacks. Secondly, we evaluate the "optimal" allocation algorithm and show how a small staging cache can drastically improve its effectiveness. We then propose the new timestamp based algorithm, RASCAL. Finally we propose a feasible implementation of the algorithm and compare it's performance to some of the other algorithms presented.

[1] Some computer vendors employ restrictions on the virtual-to-physical mapping that relax this requirement somewhat.

0.2 Related work

The importance of cache allocation decisions have already been partly addressed in some CPU architectures by the introduction of dedicated load and store instructions hinting where in the memory hierarchy an accessed datum should be installed. One example is the UltraSPARC's VIS instruction set, which has *block load* and *block store* instructions that bypass the cache hierarchy. These instructions can for example be used in a bcopy loop to avoid polluting the caches with copy data, which are unlikely to be reused soon again. The UltraSPARC III CPU also has a special *prefetch once* instruction that installs the data in a fully associative 2 KB prefetch cache accessed in parallel with the L1 cache. That way, the larger L1 will not get polluted from prefetched data that are likely to be used only once. The instruction *prefetch many* is used to prefetch data that should be installed in the normal L1 data cache. Another approach to selective caching has been taken in the implementation of the HP PA7200 CPU [KCZ+94], which has a small parallel *assist cache* in addition to a large, one-cycle latency off-chip cache. All cache lines are initially allocated in the assist cache and, upon replacement, allocated in the off-chip cache, unless a certain *spatial only* hint was specified in the instruction fetching the data. If so, the data will bypass the off-chip cache. While the allocation decision could be controlled by static compiler analysis, such analysis can sometimes be hard. We therefore believe that there is a need for a hardware algorithm, which dynamically can identify the data objects worthy of allocation in the L1 cache.

Several approaches for efficient dynamic management of the L1 data cache have been proposed lately. In the algorithms, allocation decisions are based either on the address of the instruction accessing the data, or on the data address. Most of the schemes propose a statically partitioned cache consisting of several sub-caches, where each sub cache is tailored for a certain category of cache blocks [GAV95][MMT96][RD96] [RTT+98][SG99]. Cache blocks are allocated in different sub-caches based on their type of reuse in terms of spatial and temporal locality. Tomasko et al. also proposed a statically partitioned cache [MT97], and allocates scalar and array data in different sub-caches. Srinivasan et al. takes a different approach where the cache is statically partitioned into critical/non-critical sub caches [SJLW01]. Critical loads are here defined as loads that must complete early in order not to degrade the pipeline performance. The drawback of using a statically partitioned cache for different categories of data is that it may perform worse than a conventional cache if the access pattern of a program doesn't suit the partitioning of the cache.

Several cache bypass schemes have also been proposed where some cache blocks are not allocated in the cache upon a cache miss [TFMP95][JH97][McF92]. The cache allocation algorithms introduced by Tyson et al. [TFMP95] and Johnson and Hwu [JH97] are based on access frequency and prevent fre-

quently accessed cache blocks from being replaced by less frequently used cache blocks.

We present an address-based run-time algorithm, the RASCAL algorithm, in Section 0.5.3. The distinguishing feature of our proposal is that we stream cache blocks through a small staging cache before making the L1 allocation decision. The algorithm does not explicitly make any distinction between cache blocks of different reuse categories or access frequencies nor does it statically divide the cache into different sub-caches for different categories. Instead we monitor each allocation and adaptively make allocation decisions based on the duration between recent cache allocations. The model most similar to our proposal is the MAT model introduced by Johnson and Hwu [JH97][Joh98][JH99]. The MAT model is discussed in detail in section 0.5.1.

0.3 Evaluation methodology

All evaluations are performed using the Simics full-system simulator simulating a Sun SPARC machine running Solaris 7 [MDG+98]. Since SIMICS has a modest slowdown rate we were able to study applications from SPLASH2 and SPEC CPU2000 with (close to) realistic problem size. We have restricted our evaluation in this paper to data references only. All caches are write-around and assume a perfect write buffer. A cache block size of 64 byte is used unless otherwise stated. Since this paper focuses on reducing misses in small caches, we have opted to isolate our study to the cache performance of the first-level cache. The SPLASH-2 applications were run to completion using the problem sizes suggested by Woo et al. [WOT+95]. The SPEC CPU2000 benchmarks were run with the reduced input data sets suggested by KleinOsowski et al. [KFML00].

In order to get an upper bound for our allocation algorithm, we have first studied an optimal exclusion/allocation algorithm [McF92] based on future knowledge. The basic idea is that, if the cache line singled out by the replacement algorithm will be referenced sooner than the new cache line, the new cache line will not get allocated in the cache. Note that we do not change the replacement algorithm and will only compare the new cache block with the victim that was singled out by the existing replacement algorithm, i.e., the optimal allocation algorithm we are using is not the same as optimal replacement algorithm suggested by others [Bel66, SA93]. While the optimal allocation algorithm cannot feasibly be implemented, it represents the optimal allocation strategy and initially convinced us that this area is indeed worth exploring. This algorithm will be referred to as the *optimal* algorithm.

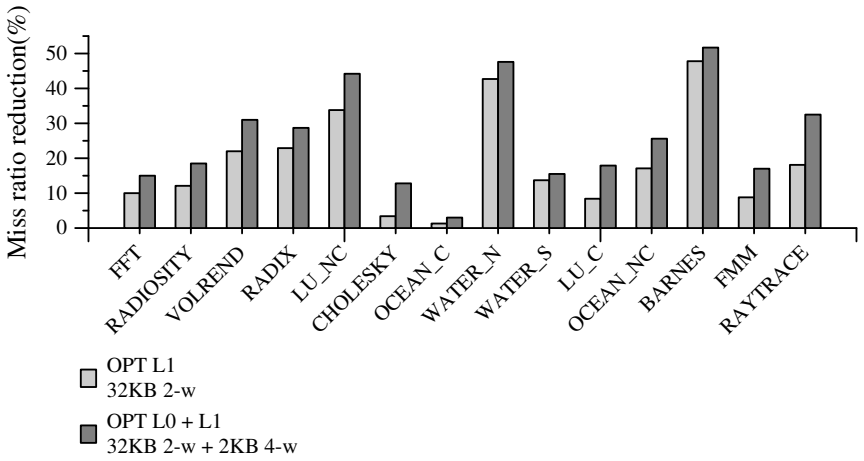

Figure 1. Comparative miss ratio reduction data for the OPT L0+L1 and OPT L1 allocation algorithms compared to a conventional cache (2-way 32 KB). Here, the L0 is a 4-way associative 2 KB cache.

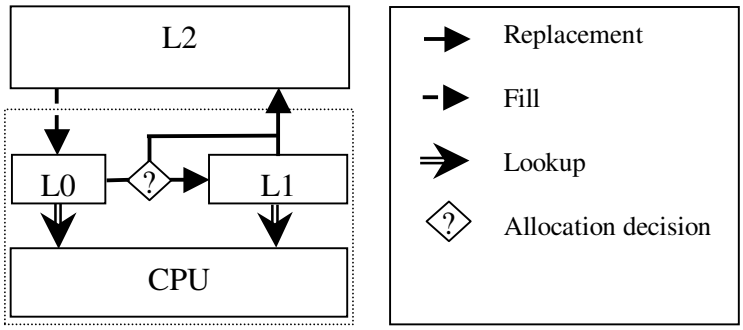

Figure 2. The baseline architecture.

0.4 Staging Cache L0

We have observed that a large fraction, often a majority of, the objects allocated in an L1 cache has temporal properties ill suited for the L1 cache. Some of these objects are never accessed before being replaced, while others have an intense, but short-lived, reuse pattern, e.g., objects with only spatial locality and read-modify-write objects with a load/store pair in a short time distance. These objects are reused shortly after the allocation but are not touched again before replacement and will spend most of their L1 tenure unused (which is later illustrated by Fig 8). Neither of these

object types make efficient use of the L1 cache. It's the objects with a long-lived temporal locality; the ones reused over and over again during a long time interval, that make the best use of the L1-cache. We label these three classes of cache lines non-temporal (NT), short-lived temporal (ST), and long-lived temporal (LT) locality.

Figure 1 shows the performance of the *optimal* algorithm, OPT L1, using the metric *miss ratio reduction*, which is defined as (1 - MissRatioXCache / MissRatioConventionalCache). Where the studied caches and the Conventional cache they are compared to have the same size and organization. The optimal algorithm will effectively avoid allocation of NT objects. However, it will happily allocate the ST objects.

In order to neither allocate the NT nor the ST objects the L1 cache, all cache blocks can be streamed through a small staging cache, called L0, before the L1 allocation decision is made. On a cache lookup, the L0 is accessed in parallel with, and has the same access time as, the L1 cache. On a cache miss, the cache block is allocated in L0. The L0 victims are, based on the selection algorithm, either allocated in the L1 or bypassed. Figure 2 shows the organization of the L0 and L1 cache. By delaying the allocation decision until after the L0 cache, most of the NT and ST objects have become inactive and will not get allocated in the L1. The graph in Figure 1 shows that adding the L0 cache significantly improves the effect of the optimal algorithm.

The potential performance gain of selective allocation is further shown in Figure 3, comparing three options for improving a 2-way, 32 KB cache: doubling the cache size, doubling the associativity, and optimal allocation decision in combination with small staging cache L0. For all of the applications, optimal allocation with L0 performs better than twice the associativity (a 4-way LRU cache of the same size), and for ten of the fourteen applications the optimal algorithm performs comparably with a cache twice the size, while maintaining the same degree of associativity. We conclude that a selective allocation in combination with a small staging cache can have a huge impact on the miss rate of a small 2-way cache. Next, we will study different practical algorithms for implementing selective allocation.

0.5 Selective Allocation

0.5.1 The MAT model

The MAT model by Johnson and Hwu bases its allocation decision on access frequency [JH97][Joh98][JH99]. The MAT model monitors accesses per macro block, which is defined as a contiguous block of memory small enough that cache blocks belonging to the same macro block are likely to display the same usage pattern. Each macro block has a hit counter associated with it, which is incremented upon a hit to a cache block belonging to the macro

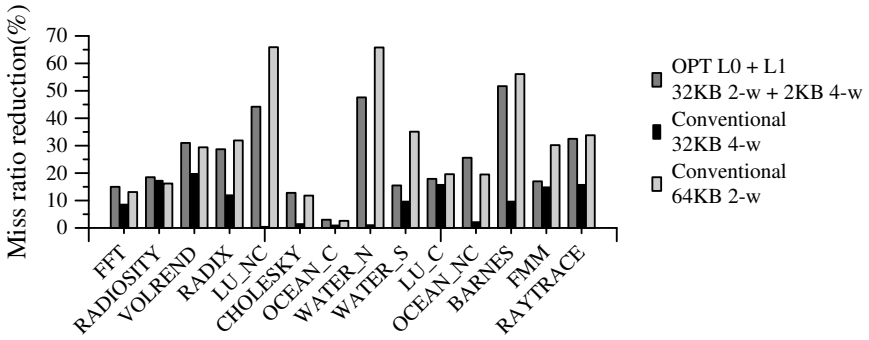

Figure 3. Miss ratio reduction compared to a conventional cache (2-way 32 kbyte).

block. The hit counters are stored in a cache structure called the Memory Address Table, MAT, which stores access frequency information for some of the macro blocks. Upon a cache miss the macro block hit counter of the victim selected by the replacement algorithm is decremented and compared to the new cache block. If the victim has the highest macro block counter value the cache block generating the miss will not be allocated in the main cache but instead in a separate smaller cache called the bypass buffer.

Since the first MAT publication[JH97], the MAT model has been enhanced by adding the notion of a decrementing counter, *decr_ctr*, per macro block in the MAT [Joh98][JH99]. The decrementing counter of a macro block is incremented by one on every conflict for a cache location held by the macro block and cleared to zero upon a cache hit to the macro block. Upon a conflict, the access counter is decremented by the value of *decr_ctr* plus one instead of just decrementing by one as in the original MAT model. The MAT models require quite complex hardware circuitry since on every cache hit a counter must be incremented through a read-modify-write operation. It also requires, as previously mentioned, a separate cache structure holding the access and decrementing counters.

0.5.2 The AAA algorithm

This algorithm is based on the existence of a staging cache L0, as shown in Figure 2. The algorithm audits each cache block during its tenure in the L1. The audition result is kept in the L2 cache and will allow for allocation into L1 for as long as the cache block "performs well". We call this the Audition-based Allocation Algorithm, AAA. The algorithm uses an *allocation history bit* per cache block in the L1 cache. When a cache block is accessed in the L1 cache the *allocation history bit* is set. The *allocation history bit* value of the last L1 tenure is stored in L2 as meta data and follows the cache line into the L0 from the L2. Cache blocks that are evicted from the L0 cache with their *allocation history bit* cleared are bypassed, while cache blocks

with the bit set are allocated in the L2 with their *allocation history bit* set to zero. In this study we have assumed that storing meta data in memory is expensive and have opted to "forget" the last audition result upon L2 eviction. Cache blocks that are allocated directly from memory get their *allocation history bit* set in the L0, which will allow for a new L1 audition.

While the advantage of the AAA algorithm is its simplicity and low implementation cost, an obvious problem with this scheme is that the algorithm has no way of detecting and changing its decision if a cache block was wrongfully classified as a bypass type. This may cause severe performance penalty from repeated bypasses of cache blocks that would benefit from allocation in the L1 cache. This problem is somewhat eased since whenever a cache block generates an L2 miss the cache block is given a new audition.

0.5.3 RASCAL– timestamp-based allocation

The Runtime Adaptive Cache ALlocation, RASCAL, algorithm is also based on the existence of a staging cache L0 and has some meta data stored together with the cache block. Each cache block has a timestamp storing its last *time of allocation decision*[2] stored together with the *allocation history bit* in the meta data. A cache block with a cleared *allocation history bit* will still be allocated in L1 if the elapsed time since the last L0 eviction is short enough. We call the elapsed time the *reuse distance*. If the reuse distance is shorter than the expected *survival time*[3] in the L1 cache, we concludes that the previous allocation decision was either an incorrect bypass decision, or that the cache block was prematurely evicted due to a conflict, and that the cache block should indeed be allocated in the L1. This makes up for the problem identified for the AAA algorithm. However, there are two practical problems to be solved for such an algorithm: long timestamps are expensive to store as meta data and the threshold for a short enough reuse distance must be determined.

The problem is that the expected *survival time* in a cache varies for different applications, as can be seen in Figure 4. It shows the distribution of cache s*urvival time*, measured in number of memory references, for each replaced cache line in a conventional 2-way 32 KB cache. In other words, how long time did each cache line survive untouched before replacement? As can be seen in Figure 4 there is not a generally applicable upper limit where the cache survival time converges across the applications and subsequently no universal *reuse distance* threshold to be used in our algorithm.

[2]The time of the last eviction from L0 See Fig 2.

[3]Defined as the elapsed time between a cache block's last hit and it's replacement.

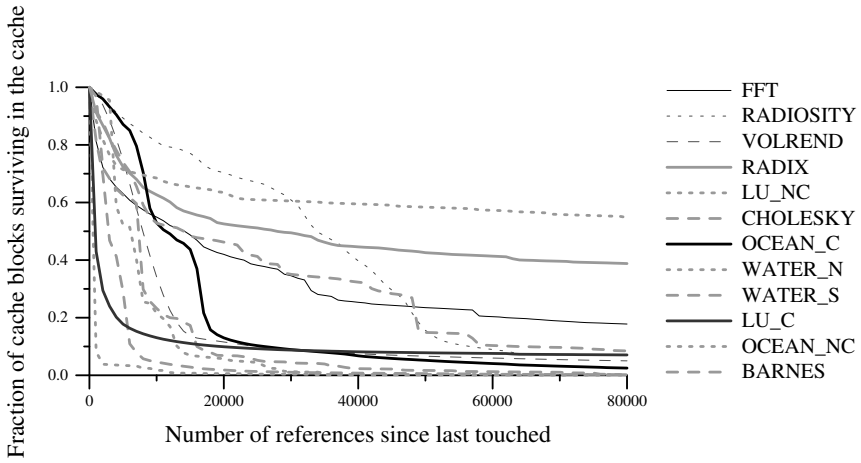

Figure 4. Distribution of survival time in a cache for a conventional 32 kbyte 2-w cache with 64 byte cache blocks measured in number of references.

Using CAT time

If we measure *reuse distance* in the time unit *cache allocation ticks* (CAT), i.e., a time unit incremented each time a cache line is allocated in the cache, the applications share a similar behavior in terms of upper bound for the survival time, as we can see in Figure 5. In the CAT time system the *survival time* of a cache block is less than twice the number of blocks in the cache, i.e., 1024 in our example, for approximately 90% of the cache blocks over all the applications. We'll use this value as the *reuse threshold* in RASCAL. If a cache block has a *reuse distance* larger than the *reuse threshold*, we can conclude that it is unlikely that the cache block would have survived if allocated. We also decide not to allocate the cache block in L1 upon L0 eviction since we expect that the next *reuse distance* of the cache block will be similar to its previous reuse distance.

The intuitive explanation to why *survival time* measured in CAT time, instead of wall clock time, is more application independent is helped by thinking about the average lifetime for a cache block in a cache. The lifetime[4] of a cache block is on average B CAT, where B is the number of cache blocks that can reside in the cache[5]. This holds since all the B objects in the cache age one CAT unit each time a cache object is replaced and since

[4]Lifetime is defined as the time from allocation to replacement.

[5]This further assumes that the entire cache contains valid data and does therefore not hold for cold start-up and multiprocessors.

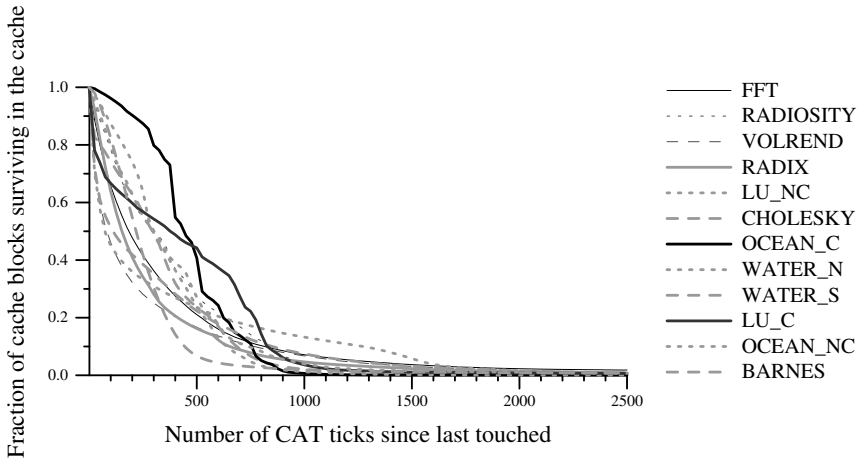

Figure 5. Distribution of cache survival time[4] for a conventional 32 kbyte 2-way cache with 64 byte cache blocks measured in cache allocation ticks (CAT).

exactly one object is inserted and replaced in each time unit. In fact, the average lifetime is independent of the cache organization. Since the average lifetime B CAT, extends to all cache organizations and since the average survival time by definition is always less or equal to the average lifetime. The average survival time is always less or equal to B.

The CAT time in RASCAL is implemented by a single counter in the L1 cache which is incremented for each L1 allocation. The value of the CAT time at L0 eviction is written into the cache block's meta data in L1, if allocated, or in L2 if bypassed. The value does not change during the cache block's tenure in L1 and L2 and will remain the same until its next eviction from L0. At this point in time, its value will be compared to the current CAT time in order to make its next allocation decision.

Allocation history counter

We have found that using a 3-bit *allocation history counter* provides more stable results than using a single *allocation history bit*. A cache block with a history counter set to zero is bypassed while cache blocks with positive history counters are allocated. When a cache block is reused within the reuse threshold or hit during its L1 tenure, the allocation history counter is set to 7. If a cache block with a non-zero history counter is evicted without the hit-bit set, the history counter is decremented. A cache block generating an L2 miss is allocated in L1.

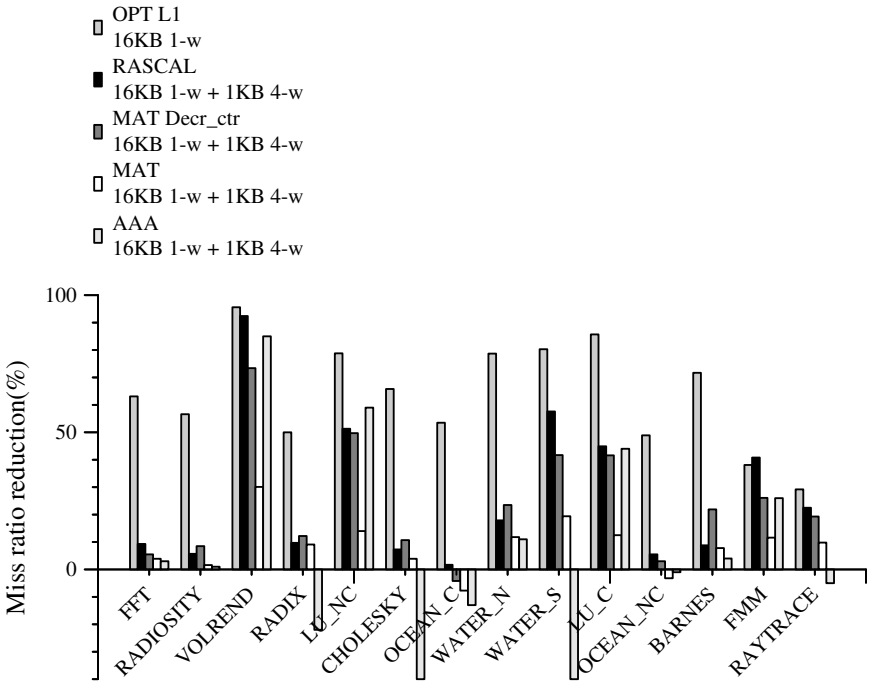

Figure 6. Miss ratio reduction compared with a conventional cache (16 KB 1-w with 32 byte cache block size) using 1KB L0 and bypass buffer.

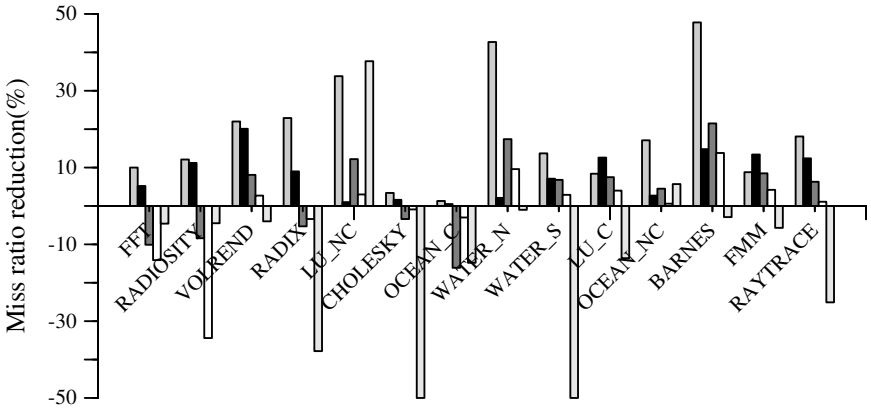

Figure 7. Miss ratio reduction compared with a conventional cache (32 KB 2-w with 64 byte cache block size).

CAT simulation parameters

The CAT counter is implemented by a 5-bit CAT counter. We have found that using 5 bits generates a tolerable amount of false detections[6]. The

[6]The CAT is incremented every $\frac{reusethreshold}{2}$ replacement. Since we are using 5 timestamp bits it will spin around every $\frac{reusethreshold}{2} \times 2^5$ replacement, which will lead to some false detection.

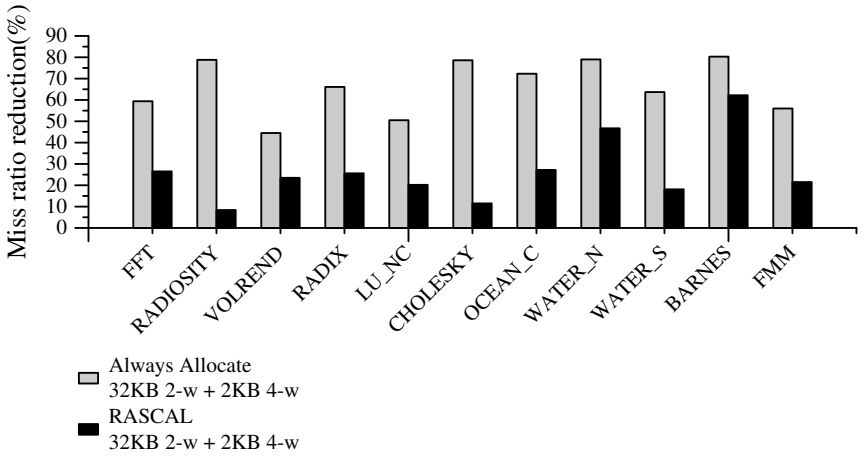

Figure 8. Fraction of cache blocks never touched before replacement from L1. For all but one application more than half of the replaced objects are never touched in an L1 using an always allocate scheme. RASCAL allocation reduces that number by about two thirds.

RASCAL algorithm therefore requires a total of 8 bits per cache block in L2. Note that the RASCAL CAT time stamp of the cache block is only accessed at the time of eviction from the L0 cache, which should be off the critical path. The reuse history bit is set on a cache hit and the reuse history counter is set to 7 on a cache hit and decremented on L0 eviction. None of these operation should add to the critical path of an LRU cache. The MAT model will however need one associative lookup to find the counter and a counter increment for each cache hit. While it will be more costly to achieve this without adding to the hit time of the cache, we still think it is doable and have not added any extra latency for the MAT hit time.

Figure 8 shows how the RASCAL allocation reduces the fraction of cache lines never touched before replacement from the L1 cache compared to a system which always allocates L0 victims in the L1.

0.6 Experimental Results

Figure 7 and 8 shows the miss rate reduction for the evaluated allocation schemes. The configuration used in the simulations presented in Figure 7 is the base configuration used in[JH99]. The MAT model was implemented with an infinite macro block table, instead of a MAT cache, an 8-bit access counter and a 4-bit decrementing counter in the MAT decr_ctr model. The comparison is made between caches of the same size and organization and with identical L0 and bypass buffer. The presented RASCAL and AAA

128 byte cacheline	Conventional	RASCAL		
	Memory	Memory Overhead Reduction		
Application	Overhead [CPI]	1-w	2-w	4-w
FFT	0,240	9%	2%	1%
RADIOSITY	1,662	4%	2%	2%
VOLREND	0,705	93%	14%	6%
RADIX	0,225	6%	3%	3%
LU_NC	1,149	59%	2%	0%
CHOLESKY	0,288	36%	6%	3%
OCEAN_C	0,863	3%	1%	1%
WATER_N	0,082	42%	3%	1%
WATER_S	0,059	66%	9%	1%
LU_C	0,095	55%	13%	0%
OCEAN_NC	0,223	14%	6%	1%
BARNES	0,175	14%	1%	1%
RAYTRACE	0,137	24%	8%	4%
EQUAKE_B	0,138	30%	4%	2%
VPR_PLACE_M	0,243	77%	67%	41%
VPR_ROUTE_M	0,435	28%	17%	11%
AMMP_B	3,231	0%	0%	0%
MCF_L	3,300	8%	1%	1%

Table 1. Memory System overhead in terms of CPI.

algorithms were evaluated with a 1 MB 4-way L2. The RASCAL algorithm requires a total of 8 bits per cache block in the L2 cache, which corresponds to less than 2 percent SRAM overhead.[7]

As can be seen in Figure 7 the performance of the AAA algorithm is very good for some applications while extremely poor for other applications. The MAT model shows some improvement for the 16 KB direct-mapped cache but for the 2-way associative 32 KB case the performance is worsened. As can be seen in table 2, 3 and 4 it is a general trend that the effectiveness of the MAT model is decreasing with more associative L1 caches. We believe that the reason for this is the effectiveness of the LRU replacement algorithm, which in itself makes sure that the least frequently used data is evicted first. Thereby reducing the effect of less frequently used data evicting highly used data. The enhanced MAT model, MAT Decr_ctr, shows an improved performance compared to the original MAT model, but some applications still shows a miss ratio increase compared to a conventional cache. The RASCAL algorithm shows a strictly positive miss rate improvement over a conventional cache although for some applications the improvement is quite modest. In order to get an idea of how the overall performance is affected by the RASCAL algorithm. We have computed the memory system overhead in terms of a highly simplified CPI model, where the CPI memory overhead is defined as $ld\_fraction \times L1\_miss\_ratio \times (L2\_hit\_ratio \times L2\_hit\_penalty + L2\_miss\_ratio \times L2\_miss\_penalty)$, and measured how the memory system overhead is affected by the RASCAL algorithm. We have assumed an in-order superscalar CPU issuing on average two instructions per cycle, an L2 miss penalty of 150 cycles and an L2 hit penalty of 15 cycles.

[7] Assuming a cache block size of 64 byte.

The memory overhead study also includes five SPEC CPU2000 benchmarks. As can be seen in Table 1 the memory overhead reduction for a direct-mapped cache is substantial for a majority of the applications, but decreases for more associative caches. The same observation can be made in the Table 2, 3, and 4 in the Appendix, where the miss ratio reduction for the RASCAL algorithm and the MAT model is presented with varying cache block sizes and associativity for a 16KB L1. Table 2, 3, and 4 also contains the absolute miss rates for the applications. By comparing Table 2, 3, and 4 one can observe that both RASCAL and MAT show a larger miss rate reduction when the cache block size is increased. This is due to the increase in capacity misses, which is targeted by the algorithms. The two SPEC benchmarks VPC_ROUTE_M and VPC_PLACE_M show the largest cut in memory overhead of all the applications.

0.7 Future Work

This paper describes our initial work with the runtime adaptive selective cache allocation algorithm, RASCAL. This field is largely unexplored to date. We believe that new algorithms can improve its performance further by more aggressive bypassing schemes. We plan on continuing this work by studying dynamic threshold-adjustment algorithms. In our study, different applications benefited from different threshold settings. We would further like to combine this scheme, which is targeted at removing capacity misses, with schemes that are targeted at conflict misses, such as a victim cache. We also intend to extend our evaluation with more benchmarks with larger working sets, where there is potentially an even bigger need for selective allocation.

0.8 Conclusion

We have demonstrated that a fourth cache property, *allocation policy*, is a potential cache enhancement scheme as cache size, associativity and replacement strategy. Using an optimal allocation policy, a 2-way 32 KB cache was shown to outperform a cache with twice the associativity and perform comparably to a double sized cache for many applications. We have also proposed a practical way to detect cache lines that would benefit from caching based on their past reuse history measured in the new time unit "cache allocation ticks" (CAT). We have also proposed a practical low-cost implementation of the RASCAL algorithm, which has shown a stable performance improvement across all the studied benchmarks.

Acknowledgment

This work is funded by the PAMP research program, supported by the Swedish Foundation for Strategic Research.

References

[AHKB00] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger.Clock rate versus ipc: The end of the road for conventional microarchitectures.In *ISCA00*, 2000.

[Bel66] L. A. Belady.A study of replacement algorithms for a virtual storage computer.*IBM Systems Journal*, 5:78–101, 1966.

[GAV95] A. Gonzalez, C. Aliagas, and M. Valero.A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality.In *Proceedings of International Conference of Supercomputing*, pages 338–347, 1995.

[Hil87] M. D. Hill.*Aspects of Cache Memory and Instruction Buffer Performance*.PhD thesis, University of California, Berkeley, 1987.

[JH97] T. Johnson and W. W. Hwu.Run-time Adaptive Cache Hierarchy Management via Reference Analysis.In *ISCA97*, pages 315–326, 1997.

[JH99] T. Johnson and W. W. Hwu.Run-time cache bypassing.*IEEE Transactions on Computers*, 48(12):1338–1354, December 1999.

[Joh98] T. Johnson.*Run-Time Adaptive Cache Management*.PhD thesis, University of Illinois at Urbana-Champaign, 1998.

[Jou90] N. P. Jouppi.Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.In *ISCA90*, 1990.

[KCZ+94] G. Kurpanek, K. Chan, J. Zheng, E. DeLano, and W. Bryg.PA7200, A PA-RISC Processor with Integrated High Performance MP Bus Interface.In *Proceedings of CompCon*, 1994.

[KFML00] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja.Adapting the spec 2000 benchmark suite for simulation-based computer architecture research.http://www.arctic.umn.edu/ papers/spec2000-wwc.pdf, 2000.Workshop on Workload Characterization, International Conference on Computer Design, Austin.

[McF92] S. McFarling.Cache Replacement with Dynamic Exclusion.In *ISCA92*, pages 191–200, 1992.

[MDG+98] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrm, and B. Werner.SimICS/sun4m: A Virtual Workstation.In *Proceedings of the 1998 USENIX Annual Technical Conference*, 1998.

[MMT96] V. Milutinovic, B. Markovic, and M. Tremblay.The Split Temporal Spatial Cache: Initial Performance Analysis.In *Proceedings of SCIzzL-5*, pages 63–69, 1996.

[MT97] W. A. Najjar M. Tomasko, S. Hadjiyiannis.Evaluation of a split scalar/array cache architecture.Technical Report TR-97-104, Department of Computer Science Colorado State University, 1997.

[RD96] J. A. Rivers and E. S. Davidson.Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design.In *Proceedings of the 1996 International Conference on Parallel Processing*, 1996.

[RTT+98] J. Rivers, E. Tam, G. Tyson, E. Davidson, and M. Farrens.Utilizing reuse information in data cache management.In *In Proceedings. 1998 International Conference on Supercomputing*, pages 449–456, 1998.

[SA93] R. A. Sugumar and S. G. Abraham.Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization.In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 24– 35, 1993.

[SG99] J. Sanchez and A. Gonzalez.A locality sensitive multi-module cache with explicit management.In *In Proceedings. 1999 International Conference on Supercomputing*, pages 51–59, 1999.

[SJLW01] S. T. Srinivasan, R. D. Ju, A. R. Lebeck, and C. Wilkerson.Locality vs. criticality.In *ISCA01*, 2001.

[TFMP95] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun.A Modified Approach to Data Cache Management.In *Proceedings of MICRO-28*, pages 93–103, 1995.

[WBL89] W. H. Wang, J. L. Baer, and H. M. Levy.Organization and performance of a two-level virtual-real cache hierarchy.In *ISCA89*, pages 140–148, 1989.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta.The SPLASH-2 programs: Characterization and methodological considerations.In *ISCA95*, 1995.

	32 byte cache block								
	Miss ratio	Miss ratio reduction		Miss ratio	Miss ratio reduction		Miss ratio	Miss ratio reduction	
		1-w			2-w			4-w	
Application	CONV	RASCAL	MAT	CONV	RASCAL	MAT	CONV	RASCAL	MAT
FFT	0,0424	13%	11%	0,0341	6%	4%	0,0317	5%	0%
RADIOSITY	0,2462	11%	11%	0,2004	8%	3%	0,1779	5%	-3%
VOLREND	0,0849	93%	74%	0,0069	21%	16%	0,0054	8%	5%
RADIX	0,0383	16%	17%	0,0316	12%	7%	0,0284	9%	0%
LU_NC	0,2403	51%	50%	0,1181	1%	3%	0,1172	0%	1%
CHOLESKY	0,0856	9%	13%	0,0735	7%	11%	0,0670	4%	7%
OCEAN_C	0,1449	2%	-2%	0,1360	1%	-7%	0,1391	4%	-2%
WATER_N	0,0168	20%	28%	0,0157	1%	21%	0,0155	0%	12%
WATER_S	0,0086	61%	46%	0,0032	7%	8%	0,0031	2%	6%
LU_C	0,0310	63%	65%	0,0093	13%	14%	0,0076	1%	-1%
OCEAN_NC	0,0914	7%	4%	0,0853	5%	3%	0,0923	12%	11%
BARNES	0,0365	15%	23%	0,0340	14%	28%	0,0347	17%	36%
RAYTRACE	0,0288	28%	25%	0,0207	12%	14%	0,0184	8%	11%
EQUAKE_B	0,0177	31%	33%	0,0119	4%	8%	0,0115	3%	7%
VPR_PLACE_M	0,0624	89%	80%	0,0126	67%	47%	0,0058	37%	26%
VPR_ROUTE_M	0,0787	22%	19%	0,0622	9%	6%	0,0581	5%	2%
AMMP_B	0,5125	1%	2%	0,5074	0%	2%	0,5064	0%	2%
MCF_L	0,4004	5%	7%	0,3796	2%	2%	0,3763	1%	2%

Table 2. Absolute miss rates for a conventional cache and miss ratio reduction for RASCAL and MAT varied over different associativities using a 32 byte cache block size and 2kbyte L0 and bypass buffer.

	64 byte cache block								
	Miss ratio	Miss ratio reduction		Miss ratio	Miss ratio reduction		Miss ratio	Miss ratio reduction	
		1-w			2-w			4-w	
Application	CONV	RASCAL	MAT	CONV	RASCAL	MAT	CONV	RASCAL	MAT
FFT	0,0292	19%	14%	0,0223	8%	4%	0,0208	7%	1%
RADIOSITY	0,1981	13%	11%	0,1591	12%	2%	0,1424	14%	2%
VOLREND	0,0955	94%	81%	0,0069	25%	21%	0,0053	11%	11%
RADIX	0,0410	13%	18%	0,0348	9%	10%	0,0322	9%	7%
LU_NC	0,2108	63%	60%	0,0785	2%	4%	0,0773	0%	3%
CHOLESKY	0,0573	26%	26%	0,0401	8%	9%	0,0364	5%	4%
OCEAN_C	0,0773	4%	-6%	0,0710	2%	-13%	0,0725	4%	-9%
WATER_N	0,0149	31%	29%	0,0106	1%	6%	0,0104	1%	2%
WATER_S	0,0087	70%	54%	0,0025	11%	10%	0,0023	2%	5%
LU_C	0,0297	71%	73%	0,0059	22%	18%	0,0043	1%	-9%
OCEAN_NC	0,0730	12%	12%	0,0649	2%	5%	0,0648	2%	5%
BARNES	0,0345	12%	20%	0,0324	4%	20%	0,0336	4%	24%
RAYTRACE	0,0236	33%	26%	0,0161	15%	14%	0,0140	9%	11%
EQUAKE_B	0,0173	46%	42%	0,0095	6%	10%	0,0090	3%	9%
VPR_PLACE_M	0,0692	86%	79%	0,0183	75%	62%	0,0066	49%	35%
VPR_ROUTE_M	0,0833	28%	23%	0,0611	13%	9%	0,0542	7%	2%
AMMP_B	0,5184	2%	2%	0,5130	0%	1%	0,5108	0%	1%
MCF_L	0,3233	5%	7%	0,3050	2%	3%	0,3020	2%	2%

Table 3. Absolute miss rates for a conventional cache and miss ratio reduction for RASCAL and MAT varied over different associativities using a 64 byte cache block size and 2kbyte L0 and bypass buffer.

	128 byte cache block								
	Miss ratio	Miss ratio reduction		Miss ratio	Miss ratio reduction		Miss ratio	Miss ratio reduction	
		1-w			2-w			4-w	
Application	CONV	RASCAL	MAT	CONV	RASCAL	MAT	CONV	RASCAL	MAT
FFT	0,0231	32%	28%	0,0153	9%	13%	0,0144	7%	12%
RADIOSITY	0,1495	14%	13%	0,1202	10%	9%	0,1154	10%	13%
VOLREND	0,1509	97%	94%	0,0061	29%	22%	0,0047	15%	13%
RADIX	0,0443	11%	19%	0,0386	5%	12%	0,0371	5%	10%
LU_NC	0,1962	71%	68%	0,0591	2%	4%	0,0578	0%	3%
CHOLESKY	0,0477	51%	44%	0,0228	11%	11%	0,0203	6%	7%
OCEAN_C	0,0437	11%	0%	0,0376	3%	-9%	0,0381	4%	-6%
WATER_N	0,0160	49%	42%	0,0080	3%	3%	0,0077	1%	1%
WATER_S	0,0111	80%	65%	0,0022	19%	14%	0,0018	3%	3%
LU_C	0,0312	71%	77%	0,0045	39%	33%	0,0025	2%	-1%
OCEAN_NC	0,0681	25%	22%	0,0577	13%	10%	0,0513	2%	0%
BARNES	0,0302	17%	18%	0,0254	2%	8%	0,0256	1%	7%
RAYTRACE	0,0222	41%	33%	0,0138	16%	16%	0,0119	10%	14%
EQUAKE_B	0,0160	60%	50%	0,0069	13%	13%	0,0063	7%	9%
VPR_PLACE_M	0,0753	82%	74%	0,0277	79%	72%	0,0096	62%	51%
VPR_ROUTE_M	0,0970	30%	25%	0,0695	19%	15%	0,0585	13%	7%
AMMP_B	0,5171	1%	2%	0,5130	0%	1%	0,5114	0%	0%
MCF_L	0,2739	5%	7%	0,2578	1%	3%	0,2566	2%	3%

Table 4. Absolute miss rates for a conventional cache and miss ratio reduction for RASCAL and MAT varied over different associativities using a 128 byte cache block size and 2kbyte L0 and bypass buffer.