ICS
TECHNICAL REPORT

Compiler–Directed Cache Assist Adaptivity *

Xiaomei Ji Dan Nicolaescu Alexander Veidenbaum
Alexandru Nicolau Rajesh Gupta
Department of Information and Computer Science
444 Computer Science, Building 302
University of California Irvine
Irvine, CA 92697–3425
{xji,dann,alexv,nicolau,rgupta}@ics.uci.edu

Technical Report #00–17

Information and Computer Science
University of California, Irvine

Compiler–Directed Cache Assist Adaptivity *

Xiaomei Ji Dan Nicolaescu Alexander Veidenbaum

Alexandru Nicolau Rajesh Gupta

Department of Information and Computer Science

444 Computer Science, Building 302

University of California Irvine

Irvine, CA 92697–3425

{xji,dann,alexv,nicolau,rgupta}@ics.uci.edu

Technical Report #00–17

Dept. of Information and Computer Science

Univ. of California at Irvine

May 2000

*This work was supported in part by the DARPA ITO under Grant DABT63-98-C-0045.

Abstract

The performance of a traditional cache memory hierarchy can be improved by utilizing mechanisms such as a victim cache or a stream buffer (cache assists). The amount of on–chip memory for cache assist is typically limited for technological reasons. In addition, the cache assist size is limited in order to maintain a fast access time. Performance gains from using a stream buffer or a victim cache, or a combination of the two, varies from program to program as well as within a program. Therefore, given a limited amount of cache assist memory, there is a need and a potential for "adaptivity" of the cache assists i.e., an ability to vary their relative size within the bounds of the cache assist memory size. We propose and study a compiler-driven adaptive cache assist organization and its effect on system performance. Several adaptivity mechanisms are proposed and investigated. The results show that a cache assist that is adaptive at loop level clearly improves the cache memory performance, has low overhead, and can be easily implemented.

Contents

List of Figures

1 Introduction

The area available for on–chip caches is limited and the size and associativity of a cache for a given processor cannot be significantly increased without causing an increase in the cycle time. A small area dedicated to a victim cache and/or a stream buffer [7] can increase the performance of the memory system while it may not be large enough to double the cache size. Victim caches eliminate conflicts and exploit temporal locality of the programs, while stream buffers exploit spatial locality because they fetch data that is likely to be accessed in the near future. We call a victim cache, a stream buffer or a combination of the two a *cache assist.*

A cache assist needs to have a high degree of associativity, and it needs to have an access time equal to that of the level of cache utilizing it, i.e. its access time is very small. This imposes a limit on the size of the cache assist memory. In [8] it is shown that for any CMOS process technology the cache size cannot be increased too much without causing an increase in cycle time and access time. When both a victim cache and a stream buffer are desirable, their relative sizes have to be selected within the bounds of the (small) cache assist memory size.

Unfortunately, neither a victim cache nor a stream buffer are a panacea: in some programs a victim cache performs much better, in others a stream buffer performs much better. In this paper we show that a dynamic combination of the two improves the overall performance the most. This happens across different applications as well as within a single application.

We propose a simple system that allows the cache assist configuration to vary at run time. A set of four special instructions is used to change the functioning of the cache assist, making it work as a stream buffer, victim cache or a combination of the two. A compiler can insert these instructions in the code at points it determines suitable by either static code analysis or using profile–directed feedback.

While the hardware modifications are modest, the following questions determining the feasibility of the approach need to be answered:

1. when should the cache assist configuration be changed,
2. how often is it necessary to reconfigure,
3. what is the optimal reconfiguration policy?

On one hand it would not be feasible to change the cache assist configuration every few instructions as the overhead associated with such reconfiguration would make the approach prohibitively expensive. On the other hand if we reconfigure too infrequently, e.g. once per function call, we might miss some optimization opportunities because a function may contain a number of loops, each of them with a distinct cache behavior.

It has been shown that the majority of dynamic instructions in a program are executed in innermost loops. An inner loop is also likely to have reasonably stable spatial/temporal locality characteristics. This suggests that an inner loop may be a good place to change the organization of the cache assists and maintain the setting for the duration of such a loop. In this paper we propose and study different schemes of adapting the cache assist at loop level, trying to determine which one has better performance. We also propose other schemes using a more static assist memory partitioning and compare their performance with the loop–level adaptive cache assist configurations.

We currently use a profile–based mechanism for the control of adaptation by the compiler. Future work will study the opportunity to use compile–time analysis for making adaptivity decisions. The size of the cache assist memory is very important from both the access time and the effectiveness of adaptivity. The effect of varying the cache assist size on the miss rate of the memory system is studied as well.

2 Related Work

Victim cache [7] is a mechanism that is aimed specifically at conflict misses. It predicts that a replaced line of data will be accessed again shortly and stores the replaced data in a small fully-associative buffer on the refill path of the cache. On a cache miss, the victim cache is checked to see whether the data is present. If so, the data is copied from the victim cache to the cache.

A stream buffer [7] is a mechanism to prefetch and store data. It consists of a FIFO memory plus an address generator. On a cache miss, all stream buffers are searched in parallel to find whether the data is present. On a hit, the data is copied to the cache and the stream buffer is refilled from successive addresses in the lower memory hierarchy. On stream buffer miss, a buffer is allocated and addresses following the miss address will be prefetched into the buffer.

To the best of our knowledge there is no previous work in applying adaptivity to configure a cache assist memory. However, adaptivity has been applied in various forms. Selected examples of its use are:

Adaptive routing pioneered by ARPANET in computer networks and, more recently, applied to multiprocessor interconnection networks [1], [3] to avoid congestion and route messages faster to their destination.

Adaptive throttling for interconnection networks [3]. [16] shows that "optimal" limit varies and suggests admitting messages into the network adaptively based on current network behavior.

Adaptive cache control of coherence protocol choice were proposed and investigated in the FLASH and JUMP-1 projects [4], [11].

Adapting branch history length in branch predictors was proposed in [9] since optimal history length was shown to vary significantly among programs.

Adaptive page size has been proposed in [14] to improve the page management overhead and it is used in to reduce the TLB and memory overhead in [12].

Adaptive adjustment of data prefetch length in hardware was shown to be advantageous [2], while in [5] the prefetch lookahead distance was adjusted dynamically either purely in hardware or with compiler assistance. A cache with a fixed large cache line is used in [10] in association with a predictor to only fetch the parts of the cache line that are likely to be used.

Adaptive cache line size was shown to improve the miss rate without an appreciable increase in bandwidth in [18], [19] and [6]. A scheme for adapting the cache line size dynamically was proposed in [18]. A special adaptive controller is incorporated in the cache access controller to monitor the memory access pattern of an application and change the line size to double or half its original size at a time in order to suit the application's needs. In [18] the cache line is truly variable, whereas [19] uses a set of four predefined values for the line size. A scheme that uses two fixed sizes was proposed in [6].

A method to use compiler provided information to do software assistance for data caches was proposed in [15]. The compiler decides through static analysis when data exhibits spatial or temporal locality and generates code to attach a special spatial/temporal tag. The tag is used by the hardware when deciding if cache lines replaced from the cache should be placed in a victim cache.

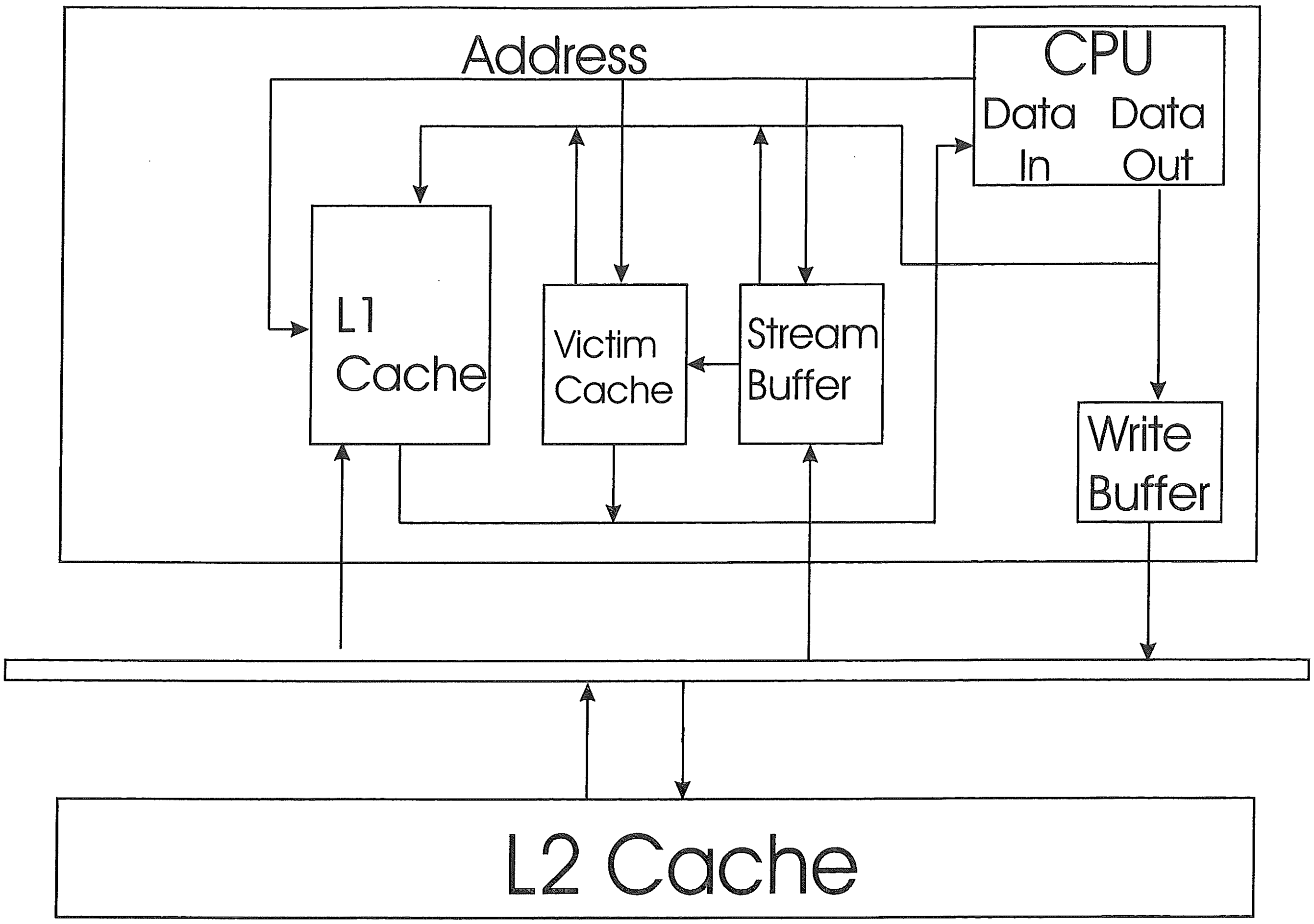

Figure 1: System Design

3 System Organization

Figure 1 shows the components of the system being studied. It consists of a 3–level memory hierachy plus a partitionable cache assist memory that can function as either a stream buffer, a victim cache or a combination of the two. The cache assist memory consists of N cache–line sized buffers connected to L1 fill path. Separate control units utilize the allocated memory as a victim cache or as a stream buffer. A fully associative write buffer with a line size identical to the L1 line size is also used.

The L1 cache is direct mapped and the hit latency is assumed to be 1 cycle. The L1 bus transfer takes 2 cycles. L2 is a 2–way set–associative with the access latency of 15 cycles. The main memory access latency is 100 cycles.

When the processor requests data, the L1 cached is searched. On a miss the victim cache and the stream buffer are searched in parallel. If both miss the request is sent to the next level of memory, otherwise the cache assist supplies the data.

Associated with the cache assist area are configuration registers. The registers contain the size of the victim cache, the size of the stream buffer, and hit counters for both of them. The configuration for the cache assist can be changed dynamically at run time using four operations:

- shrink_stream_buffer(cache_lines_to_shrink)
- shrink_victim_cache(cache_lines_to_shrink)
- extend_stream_buffer(cache_lines_to_enlarge)
- extend_victim_cache(cache_lines_to_enlarge).

Extending the stream buffer marks the new entries as invalid, shrinking it does the same and deletes any pending requests from the "issued prefetch" queue. Shrinking and extending the victim cache sets the victim cache size register to the new value and marks the added entries as invalid in the case of extending.

The compiler can insert these instructions in places in the program where static anlysis or profile based feedback determine that changing the configuration and relative sizes of the cache assists will improve the performance.

4 Experimental Infrastructure

4.1 Simulator

The framework provided by the ABSS [13] simulation system is used in this study. ABSS is a simulator that runs on SUN Sparc systems and is derived from the MINT simulator [17].

The ABSS simulator consists of 5 parts: augmentor, thread management, cycle-counting libraries, user-defined simulator of the memory system and the application program.

The augmentor program (called *doctor*) parses the original application assembly code, and adds instrumentation code that sends information about the loads and stores executed by the program to the simulator.

Our custom memory architecture simulator simulates a 3–level memory hierarchy plus a highly configurable memory cache assist with modules for modeling a stream buffer and a victim cache. The sizes of the victim cache and stream buffer are changeable at run time via commands embedded in the simulated program.

4.2 Compilation

We have used version 2.95 of the GCC compiler collection to conduct all the experiments. The compiler back–end was modified to emit special code sequences before entering a loop, or on the code path for exiting a loop. Given that the compiler back–end is common to the C and Fortran77 compiler we were able to use this instrumentation for compiling all the SPEC95 benchmarks.

The code sequences were used for adjusting the cache assist allocation, and for collecting statistics and identifying the loop (source file name and line number), and signaling to the cache simulator that a loop is being entered or exited.

In order not to modify the behavior of the program, the code sequences leave the processor in the same state as it was before the sequence in question has run. This is achieved by saving and restoring all the registers that the code sequence uses, including the flag registers. Furthermore, the loop instrumentation is done in the assembly emitting pass of the compiler (the last compilation pass), so it does not affect the code generation.

All the benchmarks where compiled using the -O2 optimization flag, the target instruction set was SPARC V8plus.

4.3 Benchmarks

The set of benchmarks shown in Table 1 was chosen because it has a good mix of both numeric and non–numeric programs, because they are fairly memory hierarchy intensive, and because SPEC95 is a standard set of benchmarks. All benchmark programs were simulated until completion.

For some of the experiments profiling was used to select an "optimal" cache assist configuration. Profiling was performed using the SPEC training input set. The profile information was then used to run the benchmarks with the reference input set. We have verified that such profiling is accurate.

Table 1: Benchmarks used

Benchmark	Decription	Instructions	Memory references
go	Plays the game GO	3.20e+10	7.76e+09
ijpeg	Image compression	2.70e+10	7.39e+09
perl	Perl interpreter	1.42e+10	3.42e+07
apsi	Calculates statistics on temperature	3.74e+10	1.20e+10
fpppp	Performs multi-electron derivatives	3.18e+11	1.03e+11
swim	Solves shallow water equations	3.21e+10	1.32e+10
turb3d	Simulates turbulence	1.13e+11	2.86e+10
wave	Solves Maxwell's equations	3.80e+10	1.20e+10

5 Performance Evaluation

To compare the relative performance of different cache assist configurations we use two main metrics: miss rate and execution time. For each experiment we gather the following kinds of data in order to evaluate the cache and cache assist performance.

- L1 and L2 miss rates
- number of hits in assist buffer
- miss rate reduction

We define the following equation to determine the overall performance improvement for the system:

$$\begin{aligned} miss\_rate\_reduction &= (old\_miss\_rate - new\_miss\_rate) \\ &\quad *100.0/old\_miss\_rate \end{aligned} \qquad (1)$$

We simulate a base cache hierarchy with a 16KB direct mapped L1 cache and a 256KB 2 way set–associative L2 cache. The line size is 32 bytes for L1 and 64 bytes for L2. We will call this the *base system configuration.* Figures 2 and 3 show the L1 and L2 miss rates respectively, for the benchmarks using the base configuration. Only *swim* and *wave* have L1 miss rates that are greater than 15% and, except for *apsi*, all of them have L2 miss rates less than 3%.

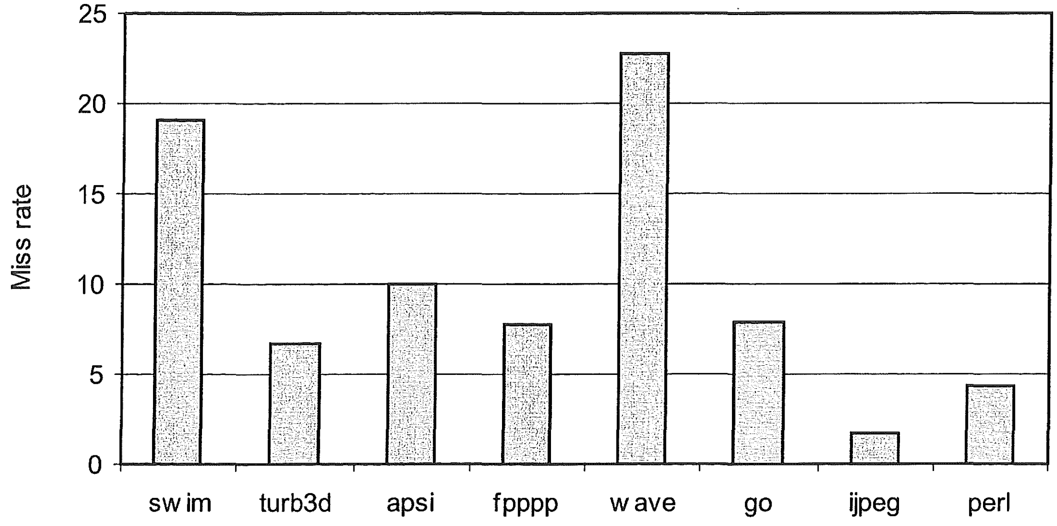

Figure 2: L1 miss rate of 16KB direct-mapped, 32B line size cache

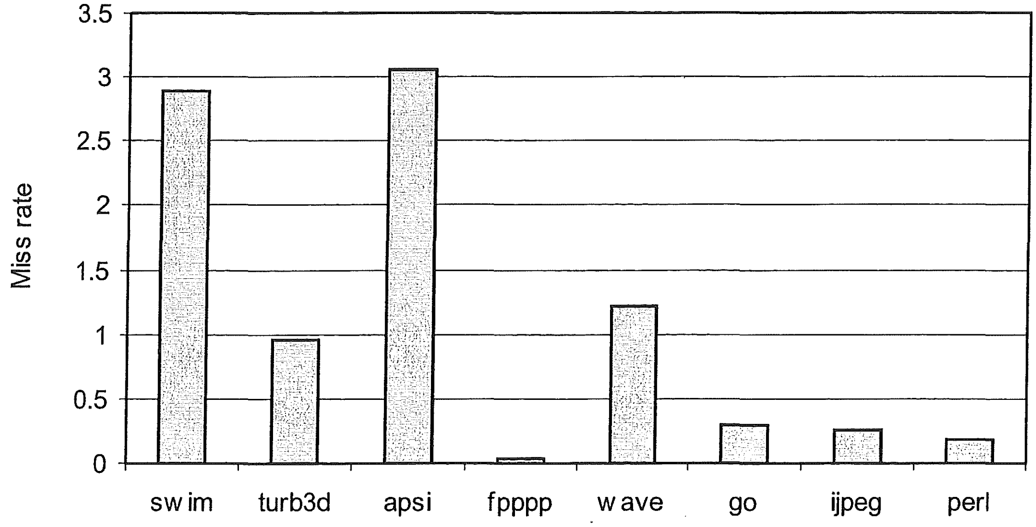

Figure 3: L2 miss rate of 256KB, 2-way set associative, 64B line size L2 cache

5.1 The Performance of Individual Cache Assists

The performance of the individual cache assists is evaluated using the base system configuration and either a 1KB victim cache or a 1KB stream buffer. Figure 4 shows the miss rate reduction for each of the assists when compared to the base configuration.

The effect varies from program to program. In *go* the stream buffer barely has an impact (under 5% miss rate reduction), but the victim cache reduces the miss rate by 50%. The same is observed for *perl* and *fpppp* where the victim cache reduces the miss rate much more than the stream buffer. The reverse is observed in the case of *turb3d* where the stream buffer reduces the miss rate by 55%, but the victim cache only reduces it by 23%. For *apsi*, *ijpeg* and *wave* the difference is not as pronounced.

The above results confirm the advantage of using a cache assist, but the type of cache assist that is most useful varies from application to application. Thus we conjecture that a system that has a cache assist that can be reconfigured between a victim cache or a stream buffer at run time on a per program basis would improve performance.

The fact that memory accesses in a program very seldom follow a uniform pattern suggests that the effect of cache assists also varies within a program. To evaluate the effect of cache assists on different portions of the code we instrument and collect performance data for all the inner loops in a program. The inner loops' memory access behavior is indicative of the entire program behavior since instructions executed in the inner loops often account for more than 98% of the memory reference instructions executed by a program.

Figure 5 shows the miss rate reduction per loop for the *apsi* benchmark when using either a 1KB stream buffer or a 1KB victim cache for a given loop. For some loops the victim cache reduces the miss rate much more than the stream buffer, whereas the opposite is true for other loops.

Figure 6 shows the miss rate reduction compared to a normal cache hierarchy for different instantiations of the loop at line 276 from file jidcting.c in the *ijpeg* benchmark when using a 1KB victim cache or stream buffer. The miss rate reduction varies a lot between loop instantiations, with some instances preferring a victim cache and others preferring a stream buffer.

The miss rate reduction when using a cache assist varies widely between different loops, and between instantiations of the same loop. We can now conclude that cache assist adaptivity is not

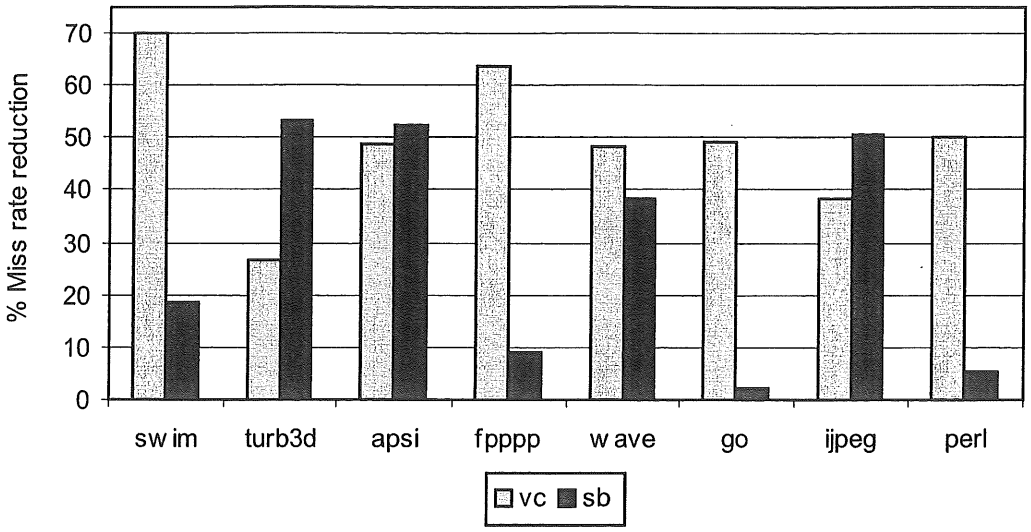

Figure 4: Miss reduction rate for a 1KB cache assist

only desirable at the program level, but it should also be applied dynamically within a program.

5.2 Dynamic Combination of Cache Assist Techniques

So far we discussed using the cache assist memory either as a stream buffer or as a victim cache. Given the fact that few program exhibit pure temporal locality or spatial locality, but rather a mix of them, one can expect that using both cache assists at the same time would have a better performance. To take advantage of the facts presented above a program could change the cache assist structure either initially or before entering a loop so that it is either a victim cache or a stream buffer, depending on what configuration results in a lower miss rate. The question is, given limited cache assist memory, what is the best way to partition it.

To investigate different possibilities of adaptation we propose four approaches to partitioning the total (limited) cache assist space between the victim cache and the stream buffer. They are:

1. Use the entire cache assist memory either as a victim cache or a stream buffer, changing the use for each loop (the *dyna_loop* approach). The decision to use one configuration or the other is taken based on which achieves a greater miss removal rate for that loop. The miss reduction information comes from profiling.

   In the *dyna_loop* case the cache assist can be used either as a stream buffer or as a victim cache for any loop. We conjecture that splitting the cache assist and using a part of it as a

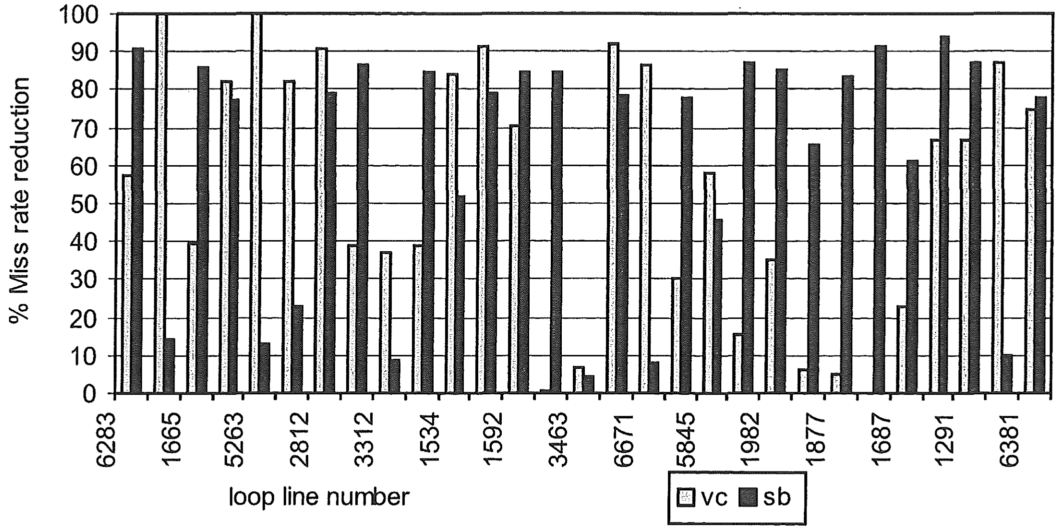

Figure 5: Miss rate reduction per loop (a 1KB assist, the *apsi* benchmark)

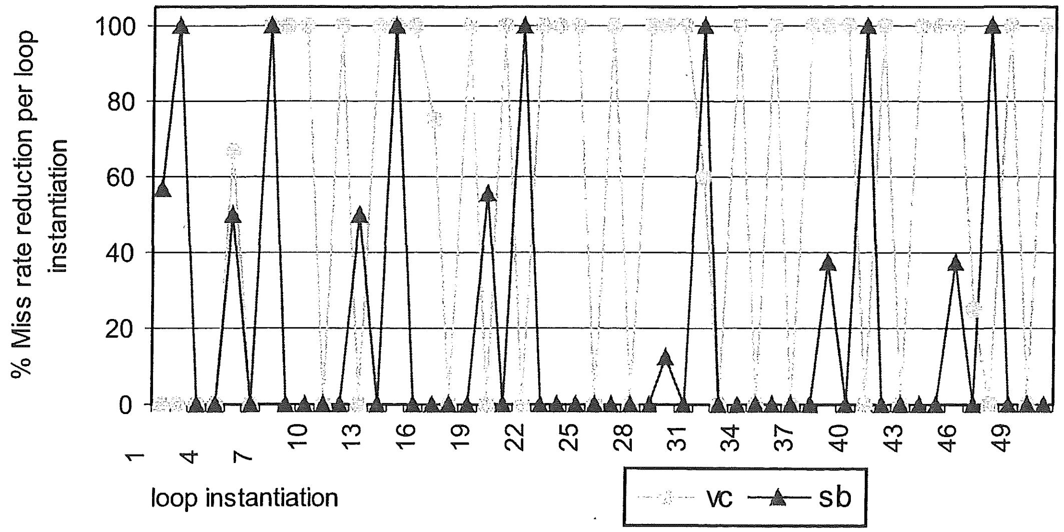

Figure 6: Miss rate reduction per loop instantiation in *ijpeg* benchmark

stream buffer, and another as a victim cache would further improve the performance. The following three strategies use this kind of partitioning.

2. Partition the cache assist memory between the victim cache and the stream buffer in the same ratio as the miss reduction rate of the victim cache and the stream buffer for the whole program (the *part_buf* approach). The partition is fixed for the duration of the program.

3. The *dyna_buf* approach partitions the cache assist memory between the victim cache or stream buffer per inner loop, proportionally to the miss removal rate ratio of victim cache and stream buffer for that loop.

4. The *half_buf* approach uses one half of the cache assist memory as a victim cache, and the other half as a stream buffer for the whole program.

dyna_loop and *dyna_buf* are dynamically adapting the cache assist configuration whereas *part_buf* and *half_buf* are not adaptive approaches, they are studied for comparison.

Figure 7 shows the miss reduction rates in the *dyna_loop* case. Profiling information gathered in the experiments summarized in Fig. 5 is used to choose the cache assist as a stream buffer or as a victim cache for each loop. The performance improvement compared to the best of either a stream buffer or a victim cache for the entire program ranges from 25% to 49%. Thus adaptivity improves performance when performed at loop level. However the miss rate got marginally worse for *fpppp* (decreased from 63.54% to 62.27%). Almost all memory accesses (98%) are executed inside one loop, and for this loop the cache assist is configured in the optimal way, the loss of performance comes from the other loops in the program.

For the programs in which the stream buffer has a very small improvement as compared to a victim cache (*go*, *perl*) the additional miss reduction rate is minimal because any possible gain from using a stream buffer is minimal.

The results for *part_buf* appear in Figure 8. With the exception of *fpppp* all the benchmarks show gains when compared to just using victim cache or a stream buffer. *Fpppp*'s loss is determined by the fact that its most dominant loop would need a bigger victim cache than what the part_buf approach allocates. However, the degradation is again minimal, a 2% decrease in miss rate reduction.

Figure 9 shows the results for the *dyna_buf.* It improves the miss ratio by 32% for *turb3d*, 43% for *apsi*, 53% for *wave*, and 51% for *ijpeg.* All the results are better than the case of using just a stream buffer or a victim cache, except for *fpppp* (see an explanation for Fig. 7). The improvement is minor for the benchmarks that show very little improvement from using a stream buffer.

Finally, the *half_buf* approach uses one half of the assist cache memory as victim cache and the other half as stream buffer for the whole program. This is not a dynamic approach, but it is used for comparison with the *dyna_loop* and *dyna_buf* approaches. The results are shown in Fig. 10 as relative percentage improvement over the miss rate reduction for the *half_buf* approach using the formula:

$$(2) \qquad \frac{miss\_rate\_reduction(dyna) - miss\_rate\_reduction(half)}{miss\_rate\_reduction(half)} * 100.0$$

The *half_buf* configuration marginally outperforms the *dyna_buf* configuration for *apsi* and *turb3d* It is significantly outperformed for *fpppp*, *go*, *ijpeg* and *perl* by up to 28%.

We can correlate this result with the experiments using the cache assist just as stream buffer or victim cache. It shows that the *dyna_buf* configuration outperforms the *half_buf* configuration in the cases where the victim cache performs clearly better than the stream buffer.

The *dyna_loop* configuration noticeably outperformed by *half_buf* in two cases, *apsi* and *ijpeg*, by up to 14%. It outperforms *half_buf* by 15 to 26% in 3 cases: *fpppp*, *go* and *perl.* Thus *dyna_loop* is not always a win.

The Figure 11 compares the miss rate reduction for all the techniques presented. Because in the previous paragraph we have compared the fixed size, non–reconfigurable cache assist *half_buf* with the reconfigurable approaches we are not going to repeat that comparison here. For *fpppp* the performance of using the cache assist as a victim cache is marginally better than any adaptive approach, but this does not happen for any other benchmark. The programs that show high miss reductions rates from using a victim cache,. but very low from using a stream buffer (*swim*, *go*, *perl*) get only minimal benefits from any of the proposed adaptive schemes. *Dyna_buf* consistently outperforms *dyna_loop* except for *swim* and *fpppp.* It also outperforms *part_buf* with the exception of *ijpeg* where the difference is negligible. Thus, the most dynamic

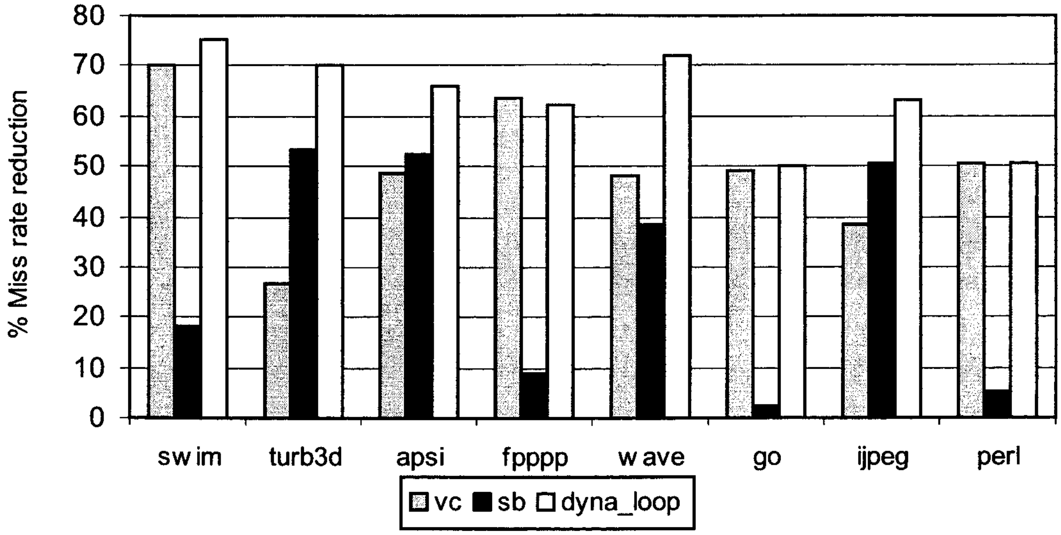

Figure 7: Miss rate reduction for dyna_loop.

approach, the *dyna_buf* is the best. Adapting the cache assist configuration is most helpful in cases when both a victim cache and a stream buffer individually show noticeable improvement.

5.3 The Effect of Cache Assist Buffer Size

The overall size of cache assist memory is an important parameter, the effectiveness of adaptation may depend on it. The miss reduction rate for a 256B cache assist memory is shown in Figure 12. Compared to a 1KB cache assist memory in Fig. 10 one can see that for the small cache assist the *dyna_buf* approach is a win in all but one case, while only in two cases the performance decreases as compared to the *half_buf* approach. Therefore, when adaptive cache assist memory space is smaller the adaptive cache assist improves performance more than it does when the cache assist memory is larger.

5.4 Compiler Support

We have shown that adaptivity of a cache assist can help reduce miss rates of programs. Furthermore, we have shown that changing the configuration of the cache assist at the point of entry in an inner loop is an excellent way to reduce the miss rate. This approach is amenable to compiler support. The compiler can determine via static analysis or via profiling feedback the optimal configuration of the cache assist for a specific loop, and it can insert the corresponding

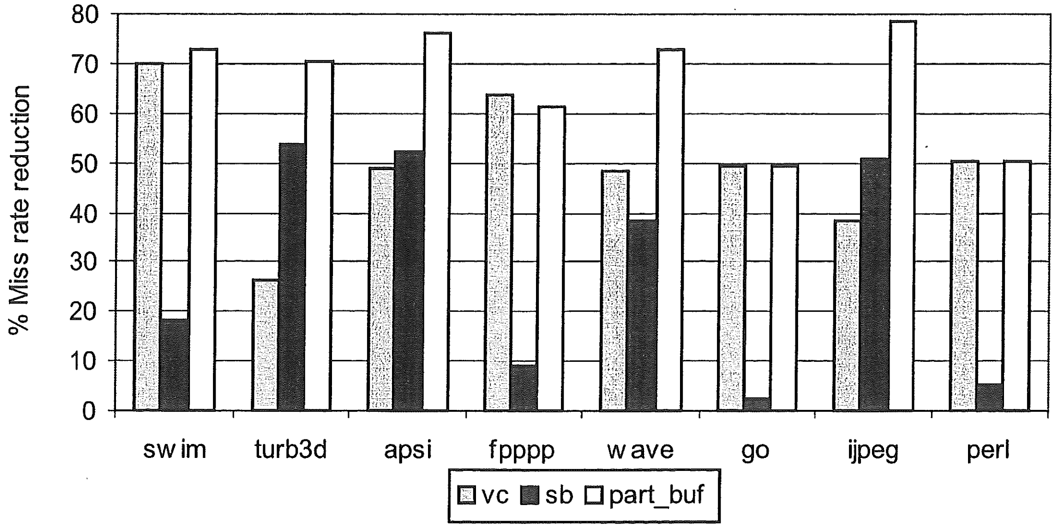

Figure 8: Miss rate reduction for part_buf.

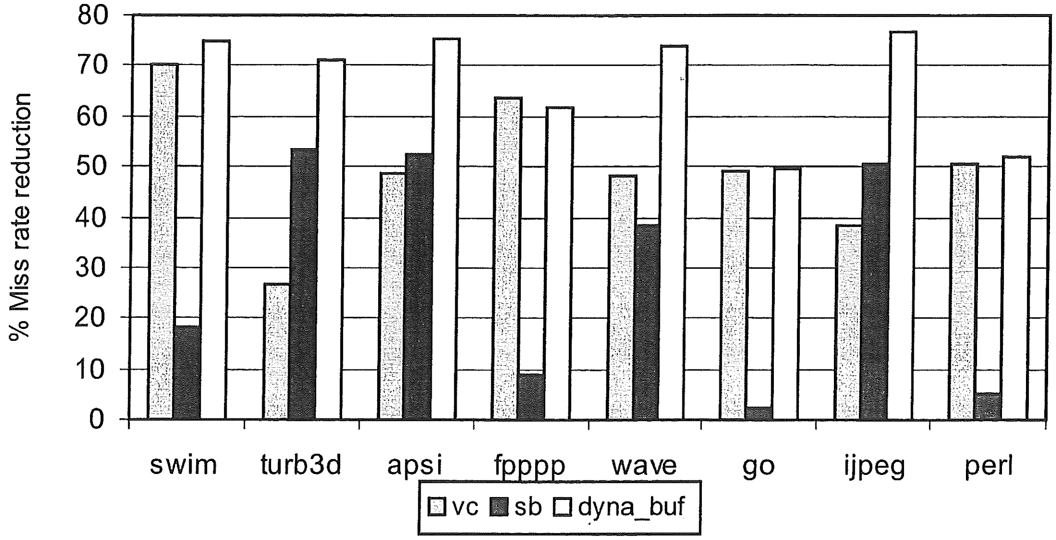

Figure 9: Miss rate reduction for dyna_buf.

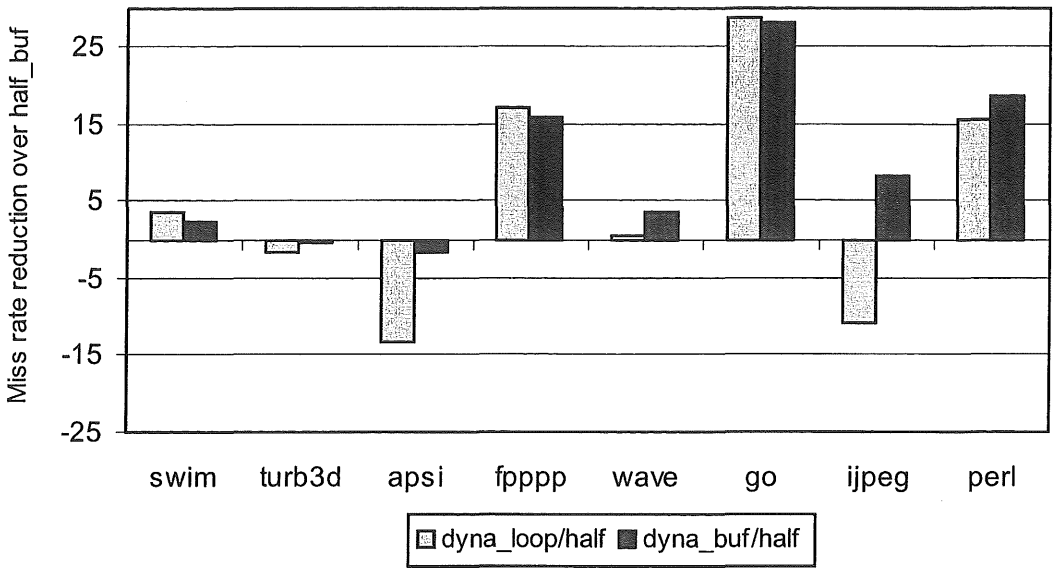

Figure 10: Dyna_loop and dyna_buf performance relative to half_buf.

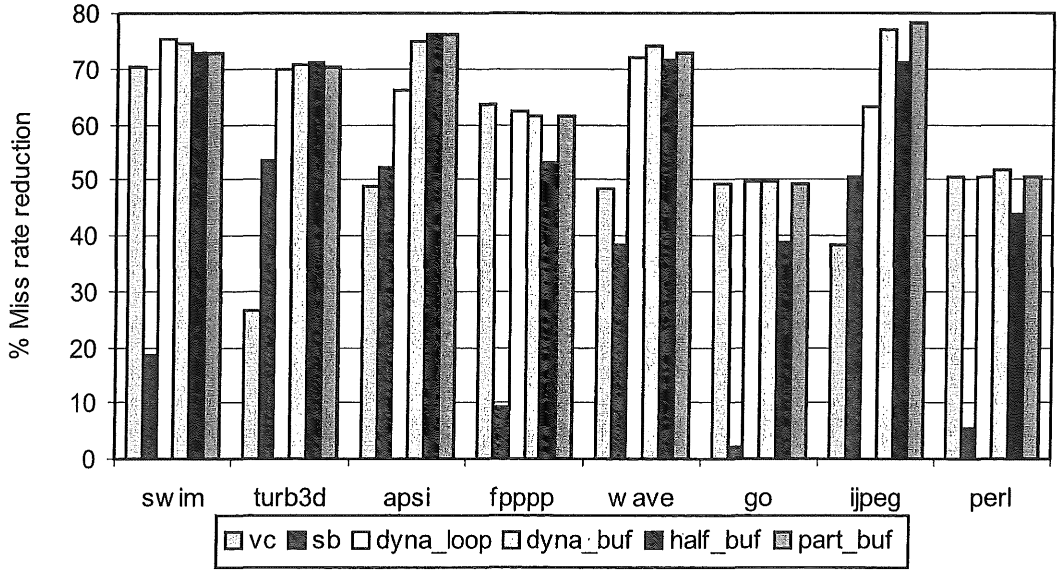

Figure 11: Miss rate reduction for all the configurations.

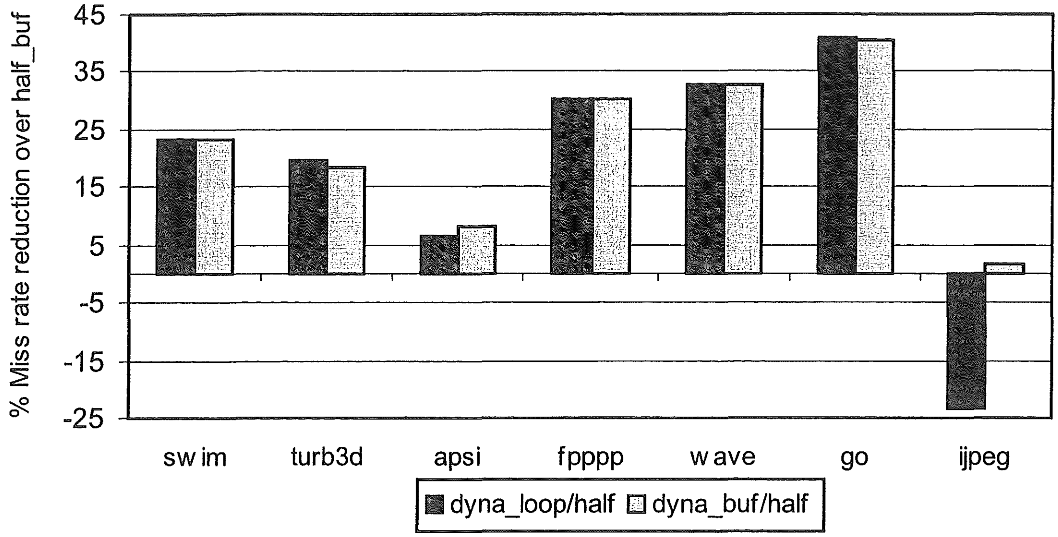

Figure 12: Dyna_loop and dyna_buf performance relative to half_buf for a 256B cache assist.

instructions at the beginning of the loop. This is the approach we advocate and we are pursuing static analysis in our compiler work. The profiling approach was used in this study.

6 Conclusions and Future Work

We have studied a memory configuration consisting of a standard cache hierarchy plus a small cache assist memory that can be used either as a stream buffer or a victim cache. The cache assist is reconfigurable at run time to allocate a certain fraction of memory to victim cache and/or to stream buffer.

We have shown that using a cache assist reduces the miss rate of the cache and that adapting the configuration of the cache assist reduces it even more. Several approaches have been studied and we have concluded that an approach that reconfigures the cache assist per inner loop at run time achieves best performance. Using a 1KB adaptive assist memory, up to 50% additional miss rate reduction is achieved by the best of the proposed methods. Simple static assist memory partitioning, on the other hand can suffer up to 15% loss of performance.

References

[1] Andrew A. Chien and Jae H. Kim. Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. In *Proc. 19th Annual Symposium on Computer Architecture*, pages 268–277, 1992.

[2] Fredrik Dahlgren, Michel Dubois, and Per Stendstrom. Fixed and adaptive sequential prefething in shared memory multiprocessors. In *Intl. Conference on Parallel Processing*, 1993.

[3] W.J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. In *IEEE Transactions on Parallel and Distributed Systems*, pages 466–475, 1993.

[4] Jeffrey Kuskin et al. The Stanford FLASH multiprocessor. In *Proc. 21st Annual Symposium on Computer Architecture*, pages 302–313, 1994.

[5] Edward H. Gornish and Alexander Veidenbaum. An integrated hardware/software data prefething scheme for shared-memory multiprocessors. In *Intl. Conference on Parallel Processing*, pages 247–254, 1994.

[6] Teresa L. Johnson and Wen mei Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[7] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-a ssociative cache and prefecth buffer.

[8] Norman P. Jouppi and Steven J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proc. 21st Annual Symposium on Computer Architecture*, 1994.

[9] Toni Juan, Sanji Sanjeevan, and Juan J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 155–166, 1998.

[10] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, 1998.

[11] T. Matsumoto, K. Nishimura, T. Kudoh, K. Hiraki, H. Amano, and H. Tanaka. Distributed shared memory architecure for JUMP-1. In *Intl. Symposium on Parallel Architecures, Algorithms, and Networks*, pages 131–137, 1996.

[12] Ted Romer, Wayne Ohlich, Anna Karlin, and Brian Bershad. Reducing TLB and memory overhead using on-line superpage promotion. 1996.

[13] D. Sunada, D. Glasco, and M. Flynn. ABSS v2.0: SPARC simulator. Technical Report CSL-TR-98-755, Stanford University, 1998.

[14] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. 1996.

[15] O. Temam and N. Drach. Software-assistance for data caches. In *Proceedings IEEE High Performance Computer Architecture*, 1995.

[16] Steve Turner and Alexander Veidenbaum. Scalability of the Cedar system. In *Supercomputing*, pages 247–254, 1994.

[17] Jack E. Veenstra and Robert J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *Intl. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 201–207, 1994.

[18] Alexander V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. Adapting cache line size to application behavior. In *Proceedings ICS'99*, June 1999.

[19] Peter Van Vleet, Eric Anderson, Lindsay Brown, Jean-Loup Baer, and Anna Karlin. Pursuing the performance potential of dynamic cache line sizes. In *Proceedings of 1999 International Conference on Computer Design*, November 1999.