Adapting Cache Line Size to Application Behavior *

Alexander V. Veidenbaum , Weiyu Tang, Rajesh Gupta,
Alexandru Nicolau, Xiaomei Ji
Dept. of Information and Computer Science
444 Computer Science, Building 302
University of California Irvine
Irvine, CA 92697-3425
alexv@ics.uci.edu

Abstract

A cache line size has a significant effect on miss rate and memory traffic. Today's computers use a fixed line size, typically 32B, which may not be optimal for a given application. Optimal size may also change during application execution. This paper describes a cache in which the line (fetch) size is continuously adjusted by hardware based on observed application accesses to the line. The approach can improve the miss rate, even over the optimal for the fixed line size, as well as significantly reduce the memory traffic.

1 Introduction

A design of a computer system is an optimization problem involving of a number of dependent variables in a very large design space. The variables include system performance, hardware constraints, cost, and architectural parameters. For general-purpose systems, performance is evaluated with respect to a particular benchmark program or program suite for a given set of design parameters. To simplify the problem, dependences between parameters are frequently ignored. A fixed set of design parameters is selected for implementation based on achieved performance and adherence to the constraints.

Designs are evaluated via a time-consuming optimization process based on simulation and the design space is never completely explored. The optimization process searches the design space guided by manual parameter selection based on past experience. The resulting design is "optimized" for an average behavior of the benchmarks used. It thus comes as no surprise that such a design is not optimal for a specific application. However, the expectation is that the loss of performance is small compared to optimal. Experience has shown that this is not always the case. This leads to design of special-purpose systems when a significant gain in performance (or cost) can be made, as in DSP or graphics applications.

Another problem with a set of "fixed" design parameters is the fact that application behavior changes during execution. Thus even within an application the optimal parameter choice is not fixed but is time-dependent. This leads to another form of performance loss, but again the expectation is that the loss is small compared to optimal. This time-domain aspect of performance is much less understood and explored than the average performance, although it has been investigated in the past at IBM and CSRD for network behavior and, recently, by ourselves and [1], among others, for the memory hierarchy.

The design of a memory hierarchy for high performance, general-purpose systems is central to achieving the desired performance levels. Its design parameters, such as cache size, line size, associativity, fetch and write policy, coherence mechanism, etc. are selected using the process sketched out above. Technological constraints usually play a primary role in the selection. It is a well known fact in the application community that a memory hierarchy can fail completely on some applications whose behavior is different from those present in the workload used to optimize the design. However, given the design approach that has to arrive at a fixed set of parameters this is unavoidable.

An alternative approach is to allow a design parameter to take on a range of values and provide a mechanism for changing towards a more optimal parameter value dynamically during execution. The same approach can also be applied to an algorithm or a policy used by hardware. A general term "adaptivity" will be used to refer to this dynamic approach. Adaptivity can potentially allow each application to approach much closer an

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

optimal architecture/hardware configuration and thus an optimal performance. It can also allow the system resources to be better utilized and shared within and across applications.

Adaptivity is not a new concept in computer systems and has been applied before 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 [2, 3] to avoid congestion and route messages faster to their destination.

Adaptive traffic throttling for interconnection networks [3]. [13] show that "optimal" limit varies and suggest admitting messages into the network adaptively based on current network behavior.

Adaptive cache control or coherence protocol choice were proposed and investigated in the FLASH and JUMP-1 projects [5, 12].

Adapting branch history length in branch predictors was proposed in [8] since optimal history length was shown to vary significantly among programs.

Adaptive adjustment of data prefetch length in hardware was shown to be advantageous [4], while in [6] the prefetch lookahead distance was adjusted dynamically either purely in hardware or with compiler assistance. [10] is another version of selective prefetch, closest to our work in many ways but also quite different. It is summarized and compared to our approach in Sec. 6.

Much of the previous work mentioned above addressed a specific problem via adaptivity although not always via adaptive hardware. Adaptivity has received a lot of attention recently with a drastic increase in VLSI complexity and transistor count as well as advances in reconfigurable logic. The research presented here addresses the use of automatic, dynamic, hardware adaptivity in the design of a first-level (L1) data cache. It is a part of a more general effort, the Adaptive Memory Reconfiguration and Management Project (AMRM) at the University of California-Irvine, to apply adaptivity to the design of a memory hierarchy. This paper, however, will only deal with the L1 data cache adaptivity.

There are several possible L1 cache parameters one can dynamically adapt. They include cache size, line size, write policy, write buffering, prefetching, etc. Some of these are only adaptable in theory. For instance, the cache size is largely determined by technology parameters and desired latency and can only be adaptively decreased. This does not make a lot of sense, except possibly as a mechanism to reduce its latency and, as a result, increase the processor clock rate which the cache largely determines, as proposed in [1]. Write policy can be switched between write-through and write-back, in fact the Intel Pentium^{TM} architecture [7] already allows this on a per-page basis but not automatically. However, the parameter that is likely to deliver a significant performance improvement while being feasible to implement adaptively is the cache line size. This paper introduces a cache design with a hardware-adaptive line size. To our knowledge, such an organization has not been previously explored.

An automatic hardware adaptive system needs to monitor system behavior and performance and modify the hardware configuration based on the observed behavior. We chose to use past behavior to predict the future hardware configuration for a given program and adapt the configuration accordingly. In general, the architecture for adaptivity requires the following capabilities:

- An ability to modify hardware parameters dynamically
- An ability to monitor performance as a function of program execution and collect statistics
- An adaptivity algorithm that monitors the statistics and decides when and how to change the hardware configuration.

The goal of this research was to investigate the potential of such an L1 cache architecture, evaluate its performance and design, and study alternatives. An additional requirement was to seek an architecture that did not have a significant hardware overhead and did not affect the system clock rate. Such an architecture is presented here and its performance evaluated using execution-driven simulation of several standard benchmarks and compared with a non-adaptive cache.

The rest of this paper is organized as follows. A general system architecture, benchmarks used, and the simulation environment are presented first. Cache behavior for non-adaptive system is shown next supporting the claim for the need to use adaptivity. A cache architecture with adaptive cache line size is described next, followed by the adaptivity algorithm and its various design alternatives. Lastly, the performance evaluation of the adaptive system is presented followed by conclusions.

2 Experimental Methodology

The benchmark choice for performance evaluation consists of selected SPEC92 and SPEC95 integer and floating-point codes and an additional floating-point code, ARC3D. The benchmark description and some execution statistics are shown in Table 1. Except for GCC and ARC3D, a complete benchmark is executed, traced, and simulated. For GCC only subroutine CC1 and for ARC3D only subroutine STEP are traced and simulated but these account for most of the execution time. The four SPEC95 codes were simulated for the first 500M references only. ARC3D was chosen because it has a significant fraction of memory accesses with long strides.

Program Name	Input	Instr (M)	Memory (M)
GCC	stmt.i	88	31
SC	loada1	862.6	128.6
LI	li-input.lsp	10,145	5,367
FPPPP	natoms	1335.1	672.6
ARC3D	-	64.2	24.2
APSI	apsi.in	1324.1	500
IJPEG	specmun.ppm	1718.5	500
PERL	scrabbl.pl	1264.9	500
WAVE	wave5.in	1481.8	500

Table 1: Benchmark Characteristics

Inputs for SPEC codes used are also listed in the table. The benchmarks were compiled on an SGI system for an R4000 processor using MIPSPro compilers with the following flags: -n32 (MIPS-III instruction set, 32b executable) and -O2 flag. They were used for execution-driven cache and memory simulation of the architecture described in this paper. The cache simulator was invoked and driven via MINT-3 [VeFo94] which models a single-issue, statically-scheduled processor.

A system architecture used in this study consists of a processor, the L1 cache with adaptive line size, and memory. We are not modeling the execution timing in this study and therefore the processor instruction timing, lack of an L2 cache, blocking L1 cache behavior, etc. are orthogonal to the study. Given the benchmarks used and current processor implementations, a 16KB cache is studied. The cache is direct-mapped and uses a write-back policy. The line sizes ranges from 8 to 256 bytes.

Primary performance metrics used in the paper are a cache miss rate and a memory-to-L1 data fetch traffic volume, either averaged over the execution of a program or shown in the time domain. In the latter case, each data value corresponds to an average behavior during a fixed interval of N memory references. N is chosen for presentation purposes only to make data graphs readable. The two main metrics are augmented with statistics specific to line size adaptivity.

3 Lack of "Optimal" Cache Line size

It has been shown in the past that an optimal line size, the size producing minimal miss rate, varies from benchmark to benchmark. To demonstrate this for our benchmark suite and support the claimed need for adaptivity, a 16K cache with a fixed line size ranging from 8 to 256Bytes is simulated. Similar results for SPEC95 have been shown in [9]. For each benchmark, the "optimal" miss rate was determined and a normalized miss rate

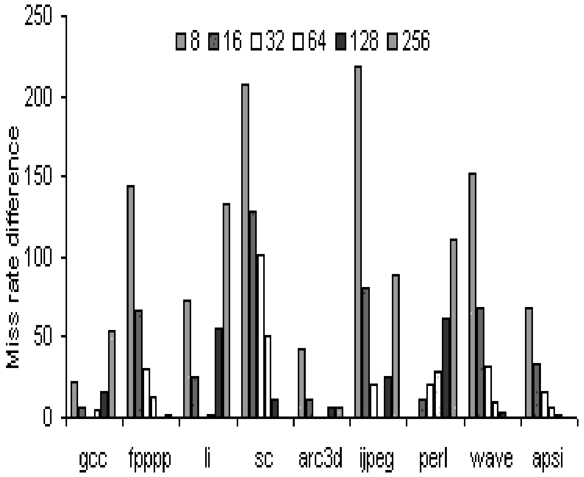

Figure 1: Normalized miss rate difference Δ_M

difference Δ_M was computed for all possible line sizes as

$$\Delta_M = \frac{M_l - M_{opt}}{M_{opt}} * 100\%$$

The normalized miss rate difference for all benchmarks is shown in Figure 3. A missing bar indicates a 0% difference and corresponds to the optimal line size.

The results show that there is no single, "optimal" line size for all benchmarks. In fact, an optimum may even be outside the range of line sizes chosen for the study. The loss of performance compared to "optimal" can be significant. Consider a 32Byte line typical of today's processors, such as Pentium or DEC Alpha. A large miss rate reduction is possible in this case: 100% for SC and 25% for FPPPP, PERL, and WAVE.

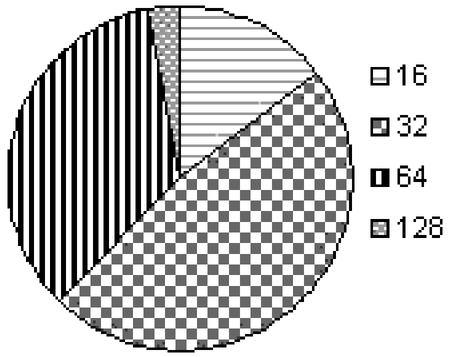

Figure 2: Distribution of optimal line size in "time"

Next, consider the intra-benchmark cache behavior. In each interval of 100K memory references a line size resulting in minimal miss rate is determined. Figure 2 shows the resulting distribution of optimal sizes in the "time-domain" for GCC. As can be seen in the figure, no single "optimal" line size exists within a benchmark as well. The optimal size is 32B in approximately half of the intervals, but it is 16B about 15% of the time and 64B approximately 30% of the time. Even 128B line is optimal a fraction of the time. Results for other benchmarks are not shown for lack of space, but their behavior is similar and strongly supports the need for adaptivity within a benchmark.

4 A Cache Organization with Adaptive Line Size

Given the need and possible performance advantages of adapting the cache line size, an architecture that can support the adaptivity is described next. The architecture has many parameters that can significantly influence performance. Our a-priori concerns were an increase in tag lookup time and the memory bandwidth and thus our choices described below are primarily influenced by the desire to keep these low.

As in the case of cache size, the line size is a built-in hardware parameter. It determines the RAM size used and the data path width, both of which are optimized for some fixed line size. Thus it cannot be "re-configured" in a standard sense as this would require changing the width of RAM blocks or redundancy and multiplexing. Instead, our approach is to have a cache with a small, "physical line, say 8Bytes (1 word), but to fetch and replace a variable number of words simultaneously as a "virtual line". The MIPS R3000 architecture [11] had a cache implementation with such a variable-size line, but the line size could only be set at hardware reset or power-up. A similar underlying hardware architecture is assumed, but allowing the line size to be changed dynamically for each (virtual) line in the L1 cache.

The issues to be resolved in the design of such a cache are:

- when to change the size
- how to change it
- what information to keep and where
- statistics needed to make the change decision

We propose to change the size on line replacement and either reduce or increase the size as follows:

- reduce the size if some fetched words were unused
- increase the size if an "adjacent" line was present in the cache. The "adjacent" line is a line of the same size that would be part of a larger-size line.

To make the decision, the virtual line usage statistics will be kept while it is in the L1 cache. Thus they reflect the line behavior from a latest miss fetch to the subsequent replacement. In particular, for each word in the line, its usage will be monitored with a counter while in the cache. An additional bit monitors the presence of the adjacent line during the line's residence in the cache. On line replacement, the statistics are used to decide what the line size should be next time it is fetched. The change can only be $2x$ or $1/2x$ of the current size in this paper.

This approach thus requires an additional memory to keep the statistics for each line. In memory we need:

- current virtual line size - log2 L, where L is the number of line sizes used
- a counter for tracking the rate of change

Each "physical" line, e. g. each 8B word, in the cache needs the following items added to the standard tag:

- current virtual line size
- "adjacent" bit
- the usage counter

One concern is the effect on system clock and any additional delays caused by adaptivity. Our approach minimizes these effects by adjusting the line size on replacement only. Thus the tag look up is not affected and can still be done in under a single clock cycle. Therefore, hit access time is not affected. Miss access time is a different story and will depend on how we use the line size information from memory. Finally, an additional memory traffic is generated on replacing a clean line when its size changes because it needs to be updated in memory. The effect of this on total memory traffic is shown later in the paper.

Finally, an initial or default virtual line size needs to be defined for cold starts. This line size can potentially have an effect on performance, but it should not be a significant one as it is only used on cold starts. However, as with some of the other choices we made in designing this architecture, it possible to set the initial line size per benchmark with compiler's help. It is also possible to rely more heavily on the default size to eliminate the need to update the size in memory more frequently. We will not pursue these issues further in the paper, however.

4.1 Line Size Adaptivity Algorithm

The algorithm and some of the alternatives possible in its design are discussed next. It is assumed that a tag is associated with each 8B word in cache and the mapping function to find the addressed word is a standard one. The current line size (3bits), the adjacent bit, and the saturating use counter (2 bits) are added to each tag. The line size can range from 8B to 256B and can only be increased/decreased by a factor of two. None of the above has an effect on the tag lookup and thus the hit case is not discussed here.

```
Given cache miss address Addr

1. Lookup cache tag at Addr
      Returns l2: line size of entry e2 at Addr
2. Start miss fetch for line e1 at Addr and its size l1
3. If l1 >= l2 Then
4.        For each line ei to be replaced Do
5.             Get an entry ei for this line and it length li
6.             Perform line_size_analysis(ei, li)
7.             If size changes or line is modified Then
8.                  Write back to memory End
          End
9. Else /*decide on the next size for line: */
10.             decrease_line_size_request (e2)
   End
11. If adjacent cache line of same size is present Then
12.         Set adjacent bit for e1 and its adjacent line
    End
```

Figure 3: The adaptivity algorithm

```
Input: line e of size l (words w0 ... w_{l-1})

If all of the words w0...w_{l/2-1} OR
      all of the w_{l/2}...w_{l-1} are not used Then
          decrease_line_size_request(e)
          If the adjacent line is in the cache Then
                Reset its adjacent bit
          End
Elseif adjacent bit is set AND
      adjacent cache line is not in cache AND
      most of the entries have low usage
  Then
      increase_line_size_request(e)
  End
```

Figure 4: Next line size analysis algorithm

The algorithm shown in Figure 3 follows the general outline of the discussion above. It starts by issuing the miss fetch request. The line size is not known until the data arrives. During the miss fetch one can read and store in a buffer of maximum line size the line(s) which will be replaced and complete the replacement process when the miss fetch data arrives. The actions taken at line replacement are discussed below. Several alternatives in the design of the algorithm are shown and explained in the text below.

A problem not present in the fixed-size case is replacement when the size of the miss line is different from the size of a line(s) to be replaced. Different line size reconfiguration decisions are made depending on the relationship between the miss line and the replaced line sizes. One or more lines may be replaced, if the incoming line size is larger than or equal to the existing line size. Every cache line to be replaced is analyzed and its next line size selected using a line size analysis algorithm shown in Figure 4. When the miss line size is smaller than the line being replaced, there is a choice of replacing the entire existing line or replacing only half of it and leaving the other half in the cache. In the former case, cache under-utilization may occur since half of the cache entry may remain unused. In the latter case, half of the existing line stays in the cache but a change in the line size occurs not based on the line's behavior. The latter approach is used in the algorithm (line 10) to conserve the memory bandwidth.

The line size analysis algorithm (Figure 4) consists of two parts. First, the word usage in each half of the line is checked. The future line size is changed to 1/2x, if one of the halves is not used and the adjacent line is reset not to grow if it is in the cache. Otherwise, if the adjacent line has been present in the cache at some point then line size needs to be increased. If the adjacent line is in the cache the increase action can be delayed until its replacement. If not, given a low usage of the words in the line its size is increased to 2X. This last check is made in an attempt to reduce frequent line size changes if the miss rate on the line is already high. The actual test is "if 50% of the word usage counters are ≤ 2".

Finally, there are several alternatives as to when the actual increase or decrease in the line size occurs. It does not have to occur immediately when one of the two functions, *decrease_line_size_request(e)* or *increase_line_size_request(e)* , is invoked. This decision has an effect on performance. Several possible choices are described next, but other possibilities exist as well.

- Change the line size immediately (direct). Line size adapts fast, but thrashing may occur.
- Change the line size only after N consecutive increase or decrease requests to prevent thrashing. An up-down counter or a finite-state machine can be used to implement this with the state stored in memory.
- Increase the line size immediately but decrease the line size only after N consecutive decrease requests (inc-fast). This alternative works better than the previous two. Line size is increased immediately to exploit spatial locality and delayed line size decrease can prevent loss of spatial locality from ran-

dom events such as conflict misses. A drawback of this alternative is an increase in bandwidth due to delay in line size decrease.

- Similarly, a decr-fast algorithm can be defined which decreases the line size immediately, while increasing the size after N consecutive increase requests.
- Apply the inc-fast for a small line size and the decr-fast mechanism for the large line size (partial-fast).

The last alternative was found to be the most effective in bandwidth reduction. The performance results presented in the next section were obtained using the partial-fast mechanism with N=2.

5 Performance Evaluation

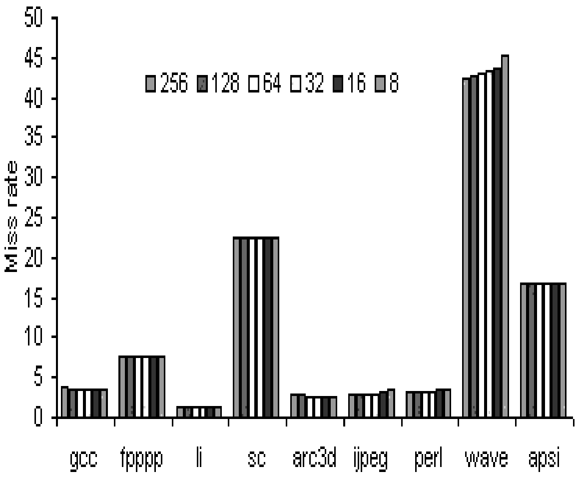

Figure 5: Miss rate vs initial adaptive line size

Let us start by presenting the miss rates for the adaptive case as a function of initial line size. The results shown in Figure 5 clearly demonstrate the adaptivity working: the miss rates are almost independent of the initial line size. The miss rate for each benchmark does not have a distinct minimum as in the case of fixed-size lines. This indicates that the initial adaptive line size selection is not important, although the smallest size should, perhaps, be avoided (more on this below).

The miss rate is not always improved by adaptivity, however. Figure 6 shows the miss rate for 32B fixed, optimal fixed, and 256B adaptive line sizes. In GCC and PERL the miss rate is lower than in the optimal fixed case. In LI and ARC3D it is slightly worse, while other benchmarks have a miss rate noticeably worse than optimal.

The reason lies in the nature of the (partial-fast) algorithm used. It is trying to avoid thrashing and prevent frequent line size increase. The worst results in the

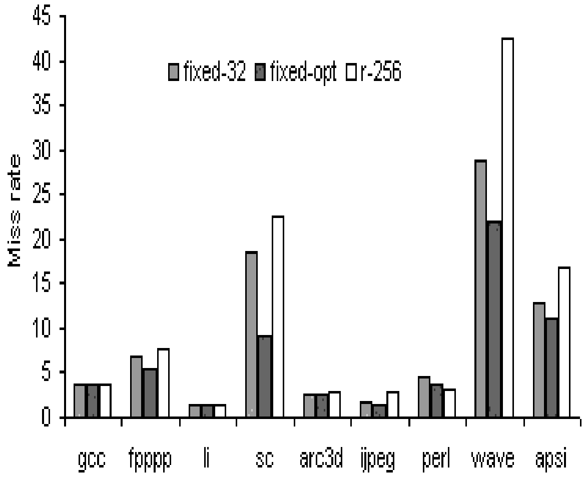

Figure 6: Miss rate

adaptive case are obtained for WAVE, APSI, and SC: benchmarks which in the fixed case achieve best performance with a 256B line. To investigate the effect, the inc-fast algorithm was used instead, which grows the line size immediately but decreases it slowly. The performance of some benchmarks became comparable to the fixed case, others improved, although not sufficiently.

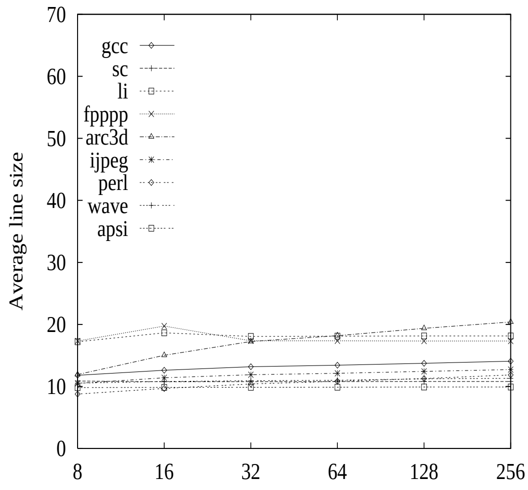

Figure 7: Average line size for adaptive organization

To understand the reasons for the behavior of the adaptive cache under the part-fast algorithm, the average line size observed during program execution is shown Figure 7. The average line size is between 10 and 20Bytes. Overall, the optimal line size is also flat vis a vis the initial line size for all benchmarks. This is another indication that the adaptivity is working well.

Still, one would expect the average line size for benchmarks such as SC and FPPPP to become larger with adaptivity since larger fixed size line have lower miss rate. It does not happen primarily for two reasons: the forced decrease in the line size when a shorter line is miss fetched and a limited definition of adaptivity and allowed line growth.

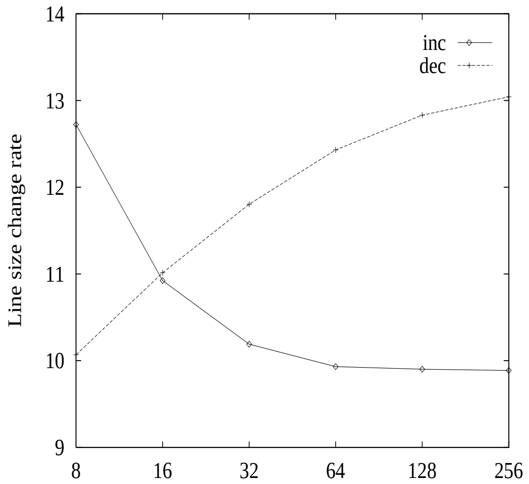

Figure 8: % misses producing line size change in GCC

While the average line size may be relatively constant with adaptivity, it does not mean that size change is infrequent or that the line size is approximately the same throughout program execution. Figure 8 shows the frequency of line size change during the execution of GCC. It is measured as a fraction of miss fetches resulting in replacement with a size change. The adjustment is frequent, with over 20% of all the lines replaced, either increasing or decreasing on replacement. As can be expected, the relative number of increases vs decreases changes with the initial virtual line size.

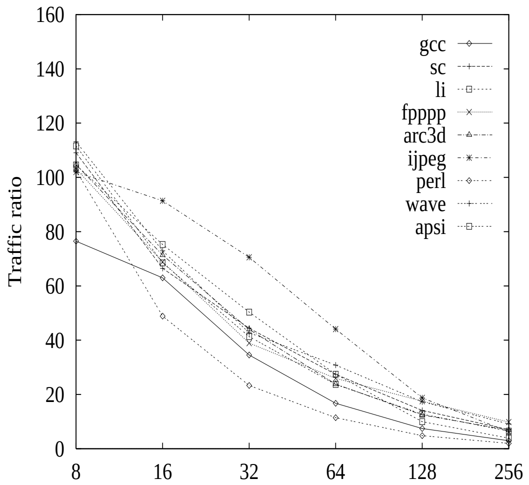

Figure 9: Fetch traffic ratio R^l_T to same-size fixed case

The last performance metric presented is a normalized memory fetch traffic ratio, R^l_T, for each line size l. It is defined as

$$R^l_T = \frac{T^l_{adapt.}}{T^l_{fixed}} * 100\%$$

R^l_T is shown in Figure 9. As should be clear from the above discussion, it was of great concern to us and influenced our adaptive algorithm design a great deal. The cache to memory "traffic" counts the total number of data bytes moved by miss fetches. The effect of adaptivity is very pronounced, it automatically reduces the utilization of the memory interface, one of the critical memory hierarchy resources.

The traffic is significantly reduced for all initial line sizes, except 8B. For the 8Byte case, adaptive organization actually ends up using a larger line size and thus generates more traffic. In general, starting at 32B initial line size, the total traffic is, on average, close to 1/2 of the traffic for the same fixed-size case. The results are not entirely surprising given the average line size observed for various codes (see Figure 7).

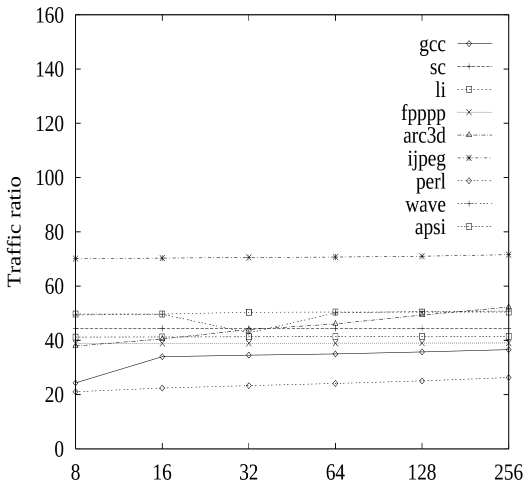

Figure 10: Fetch traffic ratio R^{32}_T to 32B fixed case

Next, consider the memory traffic for each initial adaptive line size over a 32B fixed case, R^{32}_T, shown in Figure 10. The traffic is quite flat for all benchmarks. Except for one benchmark, IJPEG, the decrease is in excess of 50% over the widely-used 32B line size.

Another view of the memory traffic reduction for 32B adaptive line size is shown in Figures 11 and 12 for ARC3D and GCC, respectively. The traffic changes significantly within an application as well. The traffic ratio to the 32B fixed line is computed and presented for each time interval (100K memory references). The decrease can be in excess of 50% over a significant portion of application's execution time. Large variations are also observed from one interval to another in GCC, showing the adaptivity adjustment to occur fast enough to make an "instantaneous" effect.

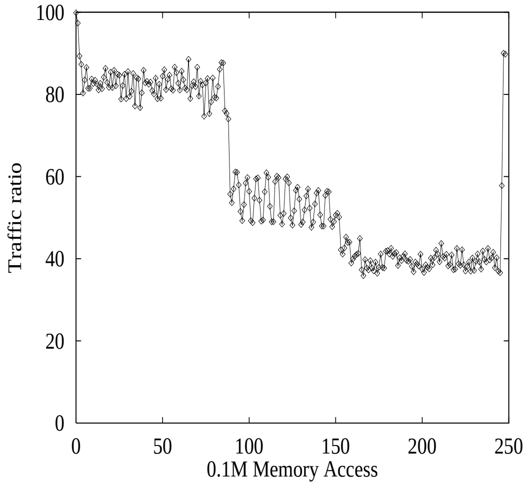

Figure 11: Time-domain traffic ratio R_T^{32} for ARC3D

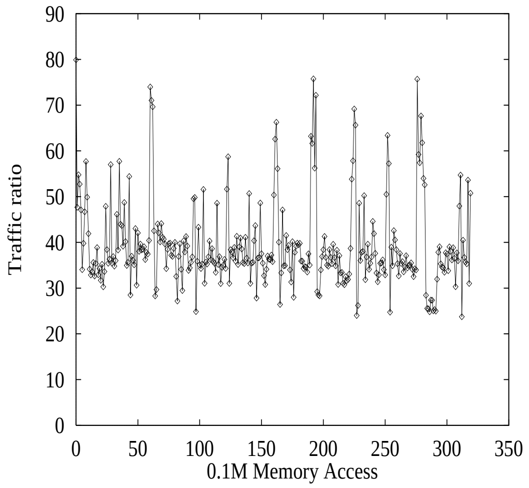

Figure 12: Time-domain traffic ratio R_T^{32} for GCC

6 Discussion

Design choices for caches with adaptive line size were already discussed in various earlier sections. This section re-examines some of them in light of the performance results obtained. The overall system architecture is revisited to discuss the effect of having an L2 cache.

Let's start with the question of hardware overheads. Much of the overall adaptivity design was influenced by hardware choices to keep it implementable. The two main sources of overhead in the cache itself are use counters and extra address bits in the tag due the "physical" line size of 8B. The use counters simulated in this study are only 2 bits. However, it appears that they may not be really needed and a single "use" bit would work equally well. The overhead of extra tag address bits is higher. However, it can be significantly reduced by setting the minimal line size to be 16B. This is quantified in the next section. Finally, the current line size stored in each physical line can be reduced to 1 bit by marking just the start of each new line.

An important issue related to the timing is how the replacement is done once the miss fetch is completed. If one waits to know the incoming line size, then the cache will be busy reading out and possibly writing back the replaced line(s) after the fetch is completed. Ignoring the expensive, brute-force approach of multi-porting the cache, one can use a replacement buffer of the size equal to maximum allowed line size. It can be filled from the cache during the miss fetch assuming the maximum size line is being replaced. Upon completion of the miss fetch and determination of precisely which lines need to be replaced, the replacement can proceed from the replacement buffer. This largely eliminates the need to access the cache except for setting some of the adjacent bits. Those can likely be placed in a separate register(s) rather than the tag itself to eliminate the need for this extra access to the cache.

Overheads in the memory are harder to reduce beyond what is suggested above for the cache. The primary concern in memory is, perhaps, the additional accesses needed to just adjust the size. They do not appear to be numerous enough but further investigation is required. Overall, given the projected capacity of DRAM chips in the next few years, the size should not be a source for concern.

The architecture used in this study did not have an L2 cache. An interesting question is how to use adaptivity in the L1 cache in the presence of the L2 cache, not to mention the question of using adaptivity in the L2 itself. We feel that adaptivity would be less effective in L2 given the typically very low L2 miss rates for standard benchmarks [HePa96]. But it may be that it can allow a smaller L2 cache with adaptivity to have the same performance as a larger, standard cache. Also, for codes that cannot fit into the L2 cache the approach may be worthwhile.

Regardless of the adaptivity in the L2 cache, the L2 can be used to keep the L1 line size instead of using the memory. In fact, this may allow one to forgo keeping any additional information in memory and start with the default or average size whenever an L2 miss would occur. As mentioned above, the best initial line size choice appears to be one of the large line sizes. The effect of periodically re-starting with a default value is probably not significant.

The effect of adjusting the algorithm in Figure 3, line 10, to forgo the line size decrease when the incoming line is smaller appears small. Our preliminary results show that this leads to increased memory traffic without a noticeable improvement in miss rates.

Finally, let us compare and differentiate our work with [KuWi98]. Their idea was to use a large, fixed-size line but only fetch words predicted to be used. They fetch a variable set of words in a 128B line, based on which words were used on a previous fetch. A predictor similar to 2-level branch history predictors [YePa91] is used to record word usage. One way to compare this with our approach is to say that we allow a variable size line and predict the size of a line rather than which words are used in a large, fixed-size line. Our "predictor" is simpler and requires very little hardware. It also incurs no lookup delay before a miss fetch can be issued. Finally, we fetch a standard, contiguous line from DRAM's, not a random set of words.

We believe that our approach is more efficient in avoiding conflicts and keeping more distinct lines in the cache. This is important when associativity and size are small. Possible improvements to the algorithm used so far can lead to further performance increases if the miss rate rather than the traffic reduction is targeted.

7 Possible Improvements

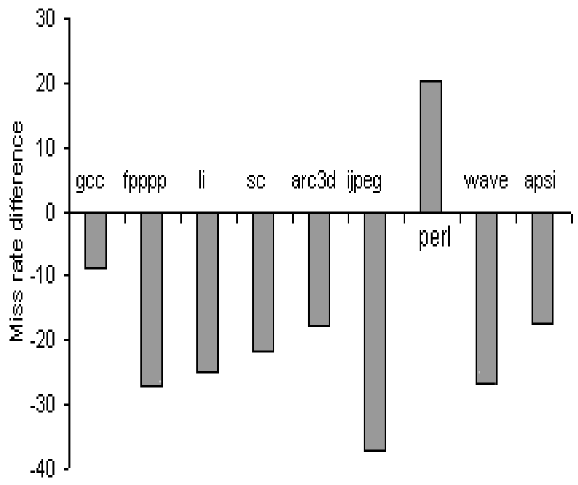

Figure 13: Miss rate difference ΔM_8^{16}: tag per 8B vs 16B

First, let us consider the performance of an adaptive cache with a 16B "physical" line. The advantage of this organization is reduced cache tag overhead. The miss rate difference, ΔM_8^{16}, for 256B adaptive line and 8B and 16B "physical" line sizes is computed as follows:

$$\Delta M_8^{16} = \frac{M_{256}^{16} - M_{256}^{8}}{M_{256}^{8}} * 100\%$$

ΔM_8^{16} for the 256B adaptive line are shown in Figure 13. They clearly demonstrate that the performance is improved when using a longer tagged line, except for one benchmark. The relative difference can be significant, up to 38%, with the average over 20%.

The traffic change due to the increase in the minimal line size/transfer unit from 8B to 16B is shown in Figure 14 as the relative traffic difference, ΔT_8^{16}:

$$\Delta T_8^{16} = \frac{T_{256}^{16} - T_{256}^{8}}{T_{256}^{8}} * 100\%$$

The traffic is increased indicating that our algorithm often reduces the line size to its minimum. The relative difference can be significant, although under 50%, and, given the improvement in miss rate may be tolerated. The 80% increase is in PERL, which also suffers increased miss rate.

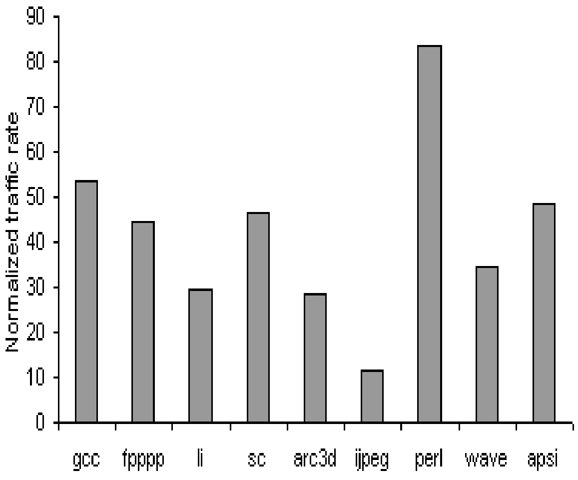

Figure 14: Traffic difference ΔT_8^{16}: tag per 8B vs 16B

As mentioned above, other modifications to the algorithm can further improve the miss rate. Three specific changes were made which are briefly summarized next. First, an adjacent line can be of equal or smaller size. In the latter case it can be located anywhere in the adjacent "half". Second, a test to increase a line size has a higher priority than a test to decrease the line size. These changes increase the chances to discover adjacent lines and double the line size. Lastly, when an incoming line is smaller than an existing line in the cache, only the exact part of the existing line to be occupied by the incoming line is replaced. Multiple "sub-lines" of a replaced line can thus remain in the cache increasing its utilization and reducing conflict misses. Using a 256B initial line size for the adaptive case, Figure 15 shows the difference $\Delta M_{fixed\_opt}^{new}$ between the miss rate of the new algorithm and the optimal miss rate of fixed line size cache:

$$\Delta M_{fixed\_opt}^{new} = \frac{M_{new} - M_{fixed\_opt}}{M_{fixed\_opt}} * 100\%$$

For all but one benchmark, the new algorithm achieves a better miss rate than the optimal fixed line size.

8 Conclusion

This paper presented a cache design in which the line size adjusts dynamically based on application behavior. A hardware algorithm to achieve this is based on monitoring the access to a given line and changing the future

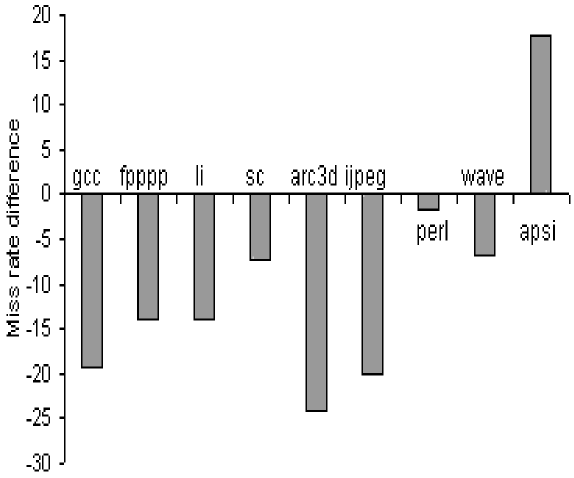

Figure 15: Miss rate difference $\Delta M^{new}_{fixed\_opt}$

line size accordingly. The size adjustment is computed during line replacement based on what was observed during the line's current stay in the cache. The size is kept in memory and takes effect on future fetches.

The performance results show that with adaptivity the miss rates are largely independent of the initial line size. The miss rate is improved in some of the benchmarks even over the optimal for the fixed size case. More importantly, the amount of memory traffic is significantly decreased in all benchmarks compared to fixed size case (except for 8B line). The traffic decreases by over 50% for an initial line size of 32B and even more for larger line size. The best strategy for applying adaptivity is to use a large initial line size and have the adaptivity decrease it as needed.

The results clearly show the feasibility of the adaptive approach. Hardware requirements of the approach are modest and can be further improved by increasing the physical line size. This also leads to significant improvement in miss rates, but the traffic increases as well. Miss rates can be improved over the optimal fixed case when traffic is not a concern.

References

[1] D. H. Albonesi, Dynamic IPC/Clock Rate Optimization, *Intl. Symposium on Computer Architecture*, pp. 282-292, June 1998.

[2] A. Chien and J. Kim, Planar Adaptive Routing: Low-cost Adaptive Networks for Multiprocessors, *Intl. Symposium on Computer Architecture*, pp. 268-277, July 1992.

[3] W. J. Dally and H. Aoki, Deadlock-free adaptive routing in multicomputer networks using virtual channels, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 466–475, Apr 1993.

[4] Fredrik Dahlgren, Michel Dubois and Per Stenstrom, Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors, *Intl. Conference on Parallel Processing*, Aug, 1993.

[5] J. Kuskin et al, The Stanford FLASH Multiprocessor, *Intl. Symposium on Computer Architecture*, pp. 302-313, April 1994.

[6] Edward H. Gornish and Alex Veidenbaum, An Integrated Hardware/Software Data Prefetching Scheme for Shared-Memory Multiprocessors, *Intl. Conference on Parallel Processing*, Aug. 1994.

[7] Pentium^{TM} Processor User's Manual, Intel Corporation, 1993.

[8] T. Juan, S. Sanjeevan, and J. Navaro, Dynamic History Length Fitting: a Third Level of Adaptivity for Branch Prediction, *Intl. Symposium on Computer Architecture*, pp.155-166, July 1998.

[9] K. Inoue, K. Kai, and K. Marukami, High Bandwidth, Variable Line-Size Cache Architecture for Merged DRAM/Logic LSIs, *Japanese IEICE Transactions on Electronics*, Vol. E81-C No. 9, pp. 1438-1447, September 1999.

[10] S. Kumar and C. Wilkerson, Exploiting Spatial Locality in Data Caches Using Spatial Footprints, *Intl. Symposium on Computer Architecture*,pp. 357-368, June 1998

[11] MIPS R3000 hardware manual, MIPS Corporation.

[12] T. Matsumoto, K. Nishimura, T. Kudoh, K. Hiraki, H. Amano, and H. Tanaka, Distributed Shared Memory Architecture for JUMP-1: A General-Purpose MPP Prototype, *Intl. Symposium on Parallel Architecures, Algorithms, and Networks*, pp. 131-137, June 1996.

[13] Steve Turner and Alex Veidenbaum, Scalability of the Cedar System, *Supercomputing*, pp. 247-254, 1994.

[14] Jack E. Veenstra and Robert J. Fowler, MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors, *Intl. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* , pp. 201-207, Jan. 1994.

[15] T.-Y. Yeh and Y. N. Patt, Two Level Adaptive Training Branch Prediction, *Intl. Symposium on Microarchitecture*, pp. 51-61, Nov. 1991.