"Adapting Cache Line Size to Application Behavior"[1]

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

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 resulting 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 [Albo98], 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 optimal architecture/hardware configuration and thus optimal performance. It can also allow the system resources to be better utilized and shared within and across applications.

[1] This work was supported in part by the DARPA/ITO grant DABT63-98-C-0045.

Adaptivity is not a new concept in computer systems and has been applied before in various forms. Selected examples of its use are:

- ARPANET pioneered adaptive routing in computer networks and, more recently, it has been applied to multiprocessor interconnection networks [DaAo93, ChKi92] to avoid congestion and route messages faster to their destination.
- Adaptive traffic throttling for interconnection networks [DaAo93] is another example. [TuVe94] show that "optimal" limit varies and suggest admitting messages into the network based on current network behavior.
- Choice of coherence protocols or adaptive cache controller were proposed and investigated in the FLASH and JUMP-1 projects [FLASH, MNKH96].
- Optimal history length in branch prediction was shown to vary significantly among programs and an adaptive mechanism for its adjustment proposed in [JuSN98].
- Finally, adaptively adjusting data prefetch length in hardware was shown to be advantageous [DaDS93], while in [GoVe94, GoVe99] the prefetch lookahead distance was adjusted dynamically either purely in hardware or with compiler assistance. [KuWi98] 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. "Discussion".

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 [Albo98]. Write policy can be switched between write-through and write-back, in fact the Intel Pentium™ architecture [Inte93] 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:

1. An ability to modify hardware parameters dynamically
2. An ability to monitor performance as a function of program execution and collect statistics
3. 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 adaptive system is presented followed by conclusions.

Experimental Methodology

Let us start by describing benchmarks used for performance evaluation. Our choice consists of several SPEC92 integer and floating-point codes and two additional floating-point codes. The last two are chosen because they have a significant fraction of memory accesses with long strides. The subroutine EFLUX is rather short, but is invoked over 2,500 times during FLO52 execution. The SPEC benchmarks used exhibit a sufficiently varied memory behavior to thoroughly check out our architecture. The benchmark statistics are shown in `Table 1`. SPEC codes used some of the standard inputs supplied, while the other two benchmarks have the data built it.

Program Name	Subroutine	Input	Instructions Executed (M)	Memory References (M)
GCC	CC1	stmt.i	88	31
SC	-	loada1	862.64	128.56
LI	-	li-input.lsp	10,144.98	5,367.19
FPPPP	-	natoms	1335.13	672.62
ARC3D	STEP	-	64.22	24.22
FLO52	EFLUX	-	5.56	1.45

Table 1. Benchmark Summary

The benchmarks were compiled on an SGI system for an R4000 processor using MIPS and 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 the cache sizes studied are 8KB and 16KB. The caches are direct-mapped and use a write-back policy. The line sizes considered range 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 later case, each point corresponds to the average behavior during a certain fixed interval, typically one million memory references. This interval was chosen for presentation purposes only in order to make the data graphs shown in the paper readable. The two main metrics are augmented with statistics specific to line size adaptivity.

Lack of "Optimal" Cache Line size

To demonstrate the need for adaptivity we simulate the 8K and 16K caches with a fixed cache line size. Similar results for SPEC95 have been shown in [KaKM98]. Miss rates for all benchmarks and line sizes from 8 to 256 Bytes are shown in `Figure 1`. Recall that the word size is 8Bytes, so the smallest cache line is one word.

The results clearly show that there is no single, "optimal" line size for all benchmarks. In fact, some of them have an optimum 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, and an 8KB cache. For the SC benchmark the difference between the miss rates for 32B line size and for the optimal is

7%, a 30% miss rate reduction. A 25% miss rate reduction is possible for FPPPP, 20% for ARC3D, etc. For a 16KB cache the miss rate reduction possible is even larger: 50% for SC and 25% for ARC3D, although the miss rates are already reduced from the 8KB case.

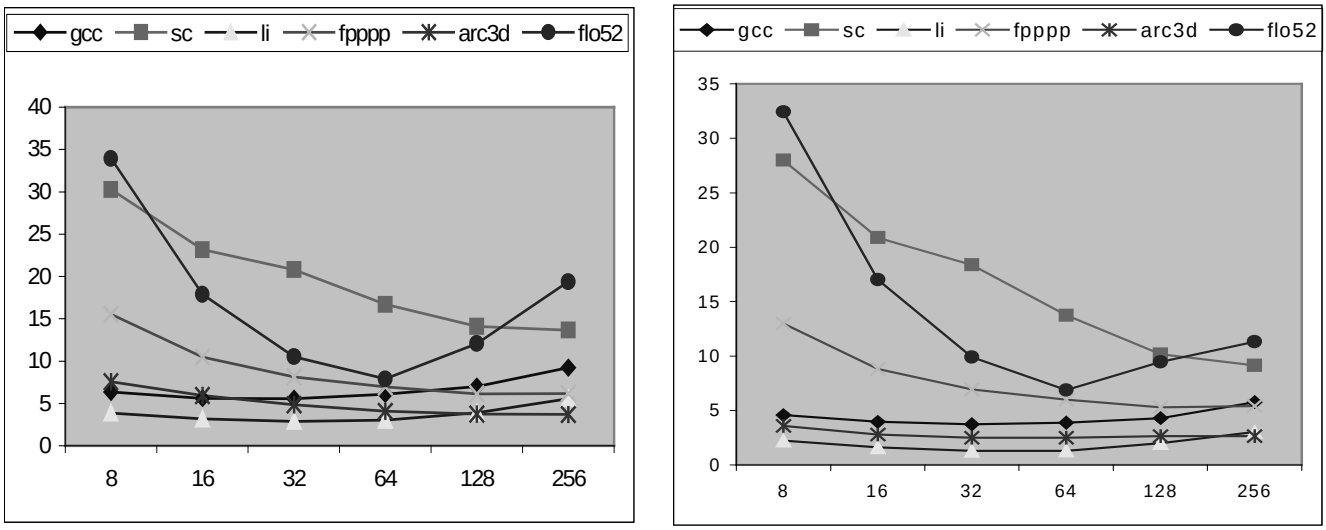


Figure 1. Miss rates for fixed-size cache line for 8KB (a) and 16KB (b) caches

Next we turn to intra-benchmark cache behavior. `Figure 2` shows the miss rates in the "time-domain" for the GCC benchmark and all six fixed line sizes. As can be seen in the figure, no "optimal" line size exists within a single benchmark as well. The optimal size changes from one interval of 100K memory references (a single graph point) to the next. 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. Time-domain results show the need to consider the

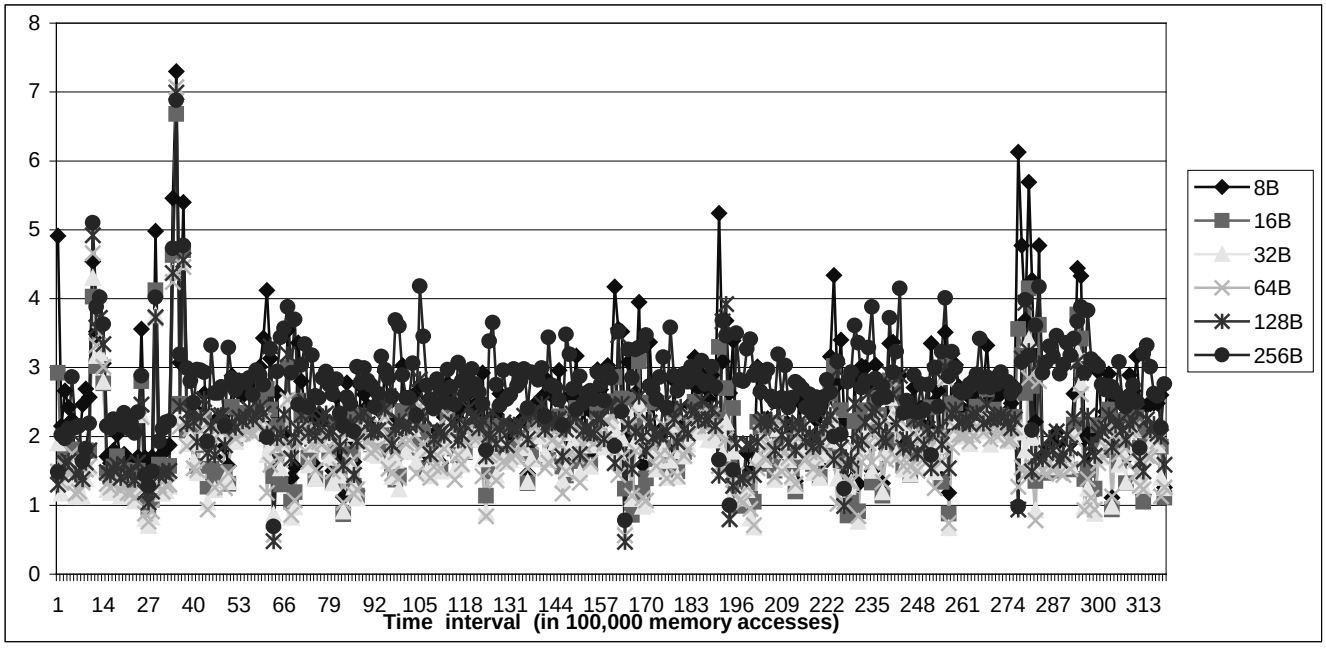


Figure 2 Time-domain miss rate for GCC and all line sizes (100K mem. References/interval)

adaptation time as another parameter in the adaptation process.

Cache Organization with an Adaptive Line Size

Having shown the need and possible performance advantages of adapting the cache line size, we now describe an architecture that can support it. The architecture has many parameters that can significantly influence performance. Some of the choices are described below, but for the purposes of the study a specific choice is made and used in the rest of the paper. Our apriori concern was the 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 "reconfigured" 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, fixed line size, say 8Bytes (1 word), but fetch and replace a variable number of words simultaneously as a "virtual line". The MIPS R3000 architecture [MIPSR3K] had a cache implementation with such a variable-size line, but the line size could only be set at hardware reset or power-up. We assume similar underlying hardware architecture, but will allow the line size to be changed dynamically for each individual (virtual) line in the L1 cache.

The issues to be resolved in the design of such a cache are:
1) when to change the size
2) how to change it
3) what information to keep and where
4) what statistics are 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 not all the words fetched were used or
- increase the size if the "adjacent" line was present in the cache. "The adjacent" line is the 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 is assumed to be 2x or ½x 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 in memory - as low as one bit marking the beginning of a line
- virtual line size – $\log_2 L$, where L is number of possible line sizes
- counter for tracking the rate of change (see below)

Each "physical" line, e. g. each 8B word, in the cache needs the following, in addition to the standard tag:
- current virtual line size
- "adjacent" bit
- the usage counter

A question arises as to how much overhead does this entail. The virtual line size will be allowed to change from 8B to 256B in this study, thus 3 bits are sufficient to encode the size. Each 8B word will need a usage counter but

the counter does not have to be large, from as little as 1 bit to a few bits per word. A 2-bit counter is used in this study. This should not be a big concern apiori given the available and projected VLSI technology.

A more valid concern may be 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 will be analyzed.

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.

Adaptivity Algorithm

The algorithm and some of the alternatives possible in its design are discussed next. As mentioned above, the alternatives can have a significant impact on performance and thus are discussed in some detail. 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) is added to each tag. The line size can range from 8B to 256B and will be increased/decreased in size 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.

The algorithm ***Miss_fetch*** 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. At the same time we 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 ***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".

Miss_fetch:
Input: cache miss address ***addr***

1. Tag lookup for ***addr*** returns l_2, the line size for the cache entry ***e*** currently in the cache
2. Miss fetch is initiated which brings in the line at ***addr*** and its line size l_1
3. If $l_1 >= l_2$ then
4. for each line to be replaced by line at ***addr*** do
5. Get an entry e_i for this line and it length l_i
6. ***line_size_analysis*** (e_i, l_i)
7. If size change or modified then write to memory Endif
8. Endfor
9. Else decide on the next size for replaced line:
10. ***decrease_line_size_request (e)***
11. Endif
12. If adjacent cache line of same size is in the cache now then
13. set ***adjacent*** bit for the line at ***addr*** and its adjacent line
14. Endif

Figure 3. Adaptivity algorithm

Finally, there are several alternatives as to when the actual increase or decrease in the line size occurs. It is 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. It turns out that this decision has an important effect on performance. The choices explored are described next, but other possibilities exist as well.

1. Change the line size immediately (direct). Line size adapts fast, but thrashing may occur.
2. 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.
3. 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 random events such as conflict misses. A draw back of this alternative is an increase in bandwidth due to delay in line size decrease.
4. Similarly, a decr-fast algorithm can be defined which decreases the line size immediately, while increasing the size after N consecutive increase requests.
5. Apply the inc-fast for a small line size and the decr-fast mechanism for the large line size (partial-fast). This alternative is the most effective in bandwidth reduction.

The performance results presented in the next section were obtained using the partial-fast mechanism with N=2. It may not always be the best adaptive algorithm, but it uses the least amount of bandwidth and was chosen for this reason. For one or two benchmarks a better miss rate was achieved by incr-fast algorithm, especially for larger initial line size (will not be shown).

line_size_analysis:
Input: a cache entry ***e*** and its size ***l***
A cache line consists of words $e_1,\ldots,e_l$

1. If all of the words $e_1 \ldots e_{l/2}$ or all of $e_{l/2+1},\ldots,e_l$ are not used then
2. ***decrease_line_size_request(e)***
3. If the "adjacent"cache line is in the cache then reset its adjacent bit
4. Endif
5. Elseif *adjacent bit is set* AND
6. *the "adjacent" cache line is not in the cache* AND
7. *most of the entries have low usage*
8. then
9. ***increase_line_size_request(e)***
10. Endif

Figure 4. Next line size analysis algorithm

Performance Evaluation

We begin the evaluation of the adaptive cache line size organization by presenting the miss rates for all benchmarks in `Figure 5` and `Figure 6` for 8KB and 16KB caches, respectively. All the data is for partial-fast algorithm. The data is organized as a bar graph and for each benchmark we show fixed size and adaptive size organizations sized by side. All line sizes are plotted for each benchmark, where for adaptive the line size means the *initial* line size. The results show adaptivity to be working. The initial line size does not matter in all but one

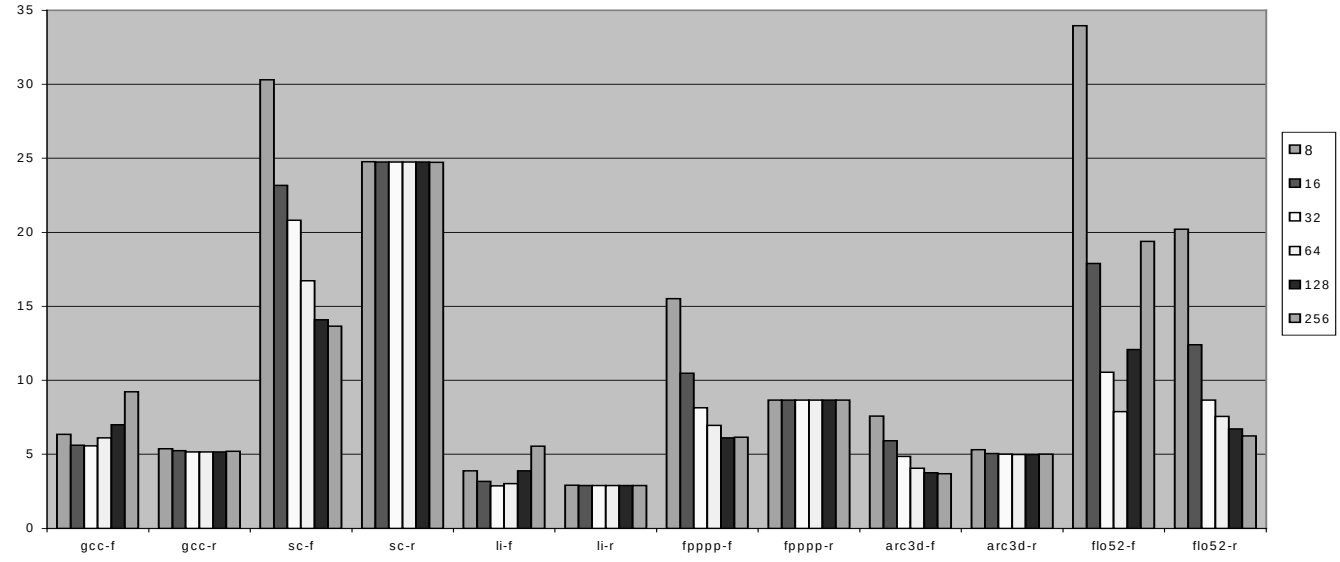


Figure 5. Miss rates for fixed (-f) and adaptive (-r) line size for 8KB cache

case. The adaptivity quickly adjusts the size so that miss rate is almost independent of the line size. The miss rate curves are now almost flat, without a distinct minimum as in the case of fixed-size lines. This indicates that the initial line size selection is not important, although the smallest size should, perhaps, be avoided (more on this below). Also, the same performance can possibly be achieved with fewer possible sizes as well.

FLO52 is a noticeable exception, the miss rate improves significantly with initial line size increase. The reason why the initial size has such a large effect here is the small number of memory references. The cold start misses are the bulk of all misses. However, after the initial decrease from 8B to 32B initial line size, the miss rate continues to decrease, unlike in the case of the fixed line size. This points to a better cache space utilization by the adaptive algorithm, which allows different line sizes to coexist in the cache. Recall that FLO52 was selected because of long-stride accesses which, for a larger line size favored by stride-1 accesses, can cause poor utilization. Overall, we believe that with a reasonable-size data set the miss rates will become flat here as well.

The overall performance is not 100% better, however. GCC, LI, and FLO52 have the same or better performance, while ARC3D is slightly worse off. The other two benchmarks, SC and FPPPP, show a performance degradation for almost all line sizes. The reason lies in the nature of the algorithm used. It is trying to avoid thrashing and prevent frequent line size increase. As the results for fixed line size indicate, SC can benefit from larger line size. To investigate the effect, the inc-fast algorithm was used instead, which grows the line size immediately but decreases it slowly. ARC3D performance became comparable to the fixed case and the performance of SC was improved, although not sufficiently, but not that of FPPP.

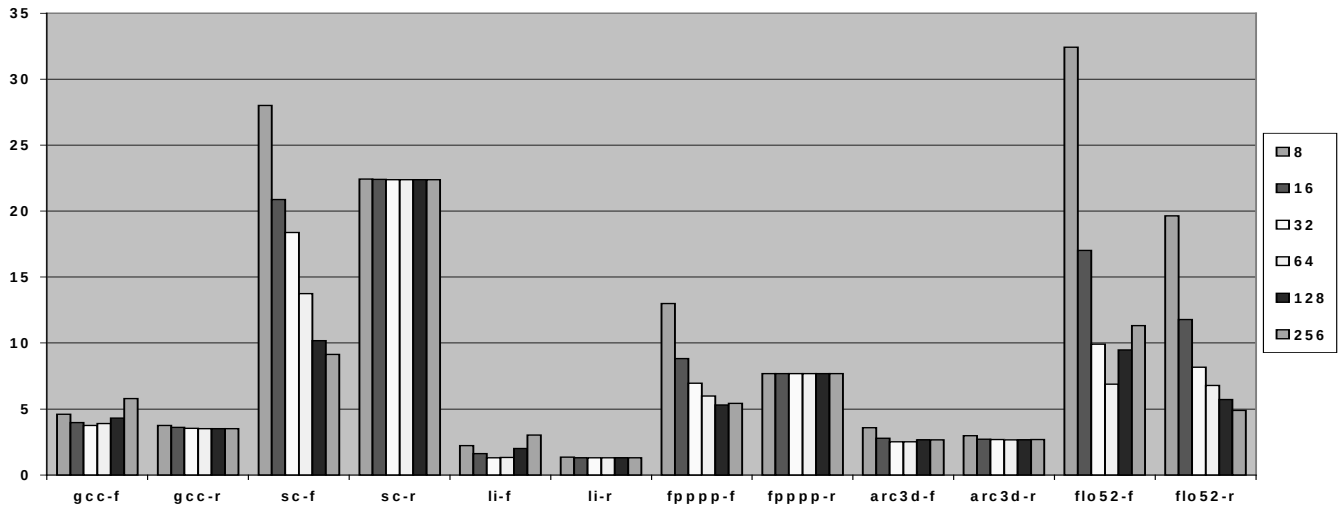
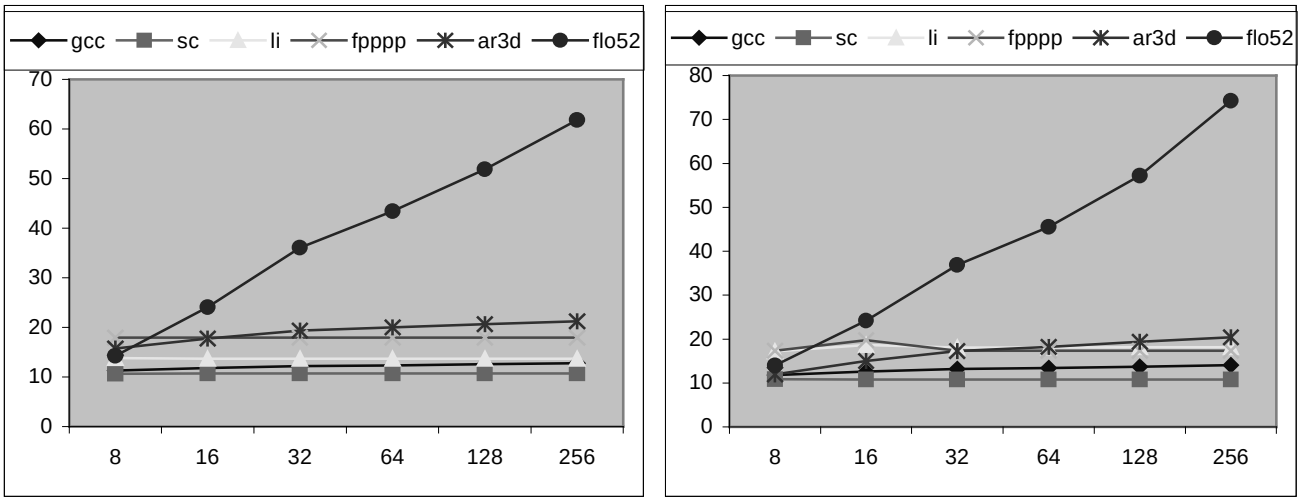


Figure 6. Miss rates for fixed (-f) and adaptive (-r) line size for 16KB cache

To understand the reasons for the behavior of the adaptive cache (under the part-fast algorithm), the average line size observed during program execution was investigated. It is shown `Figure 7` for all the benchmarks. Of all the benchmarks used, FLO52 demands and, due to adaptivity, obtains a larger effective line size. The performance keeps improving due to this. However, the optimal line size is 60 or 70B, rather than the initial max. size of 256B. Overall, the optimal line size is also pretty flat vis a vis the initial line size for all other benchmarks. This is another indication that the adaptivity is working well, although the averaging interval is rather large.

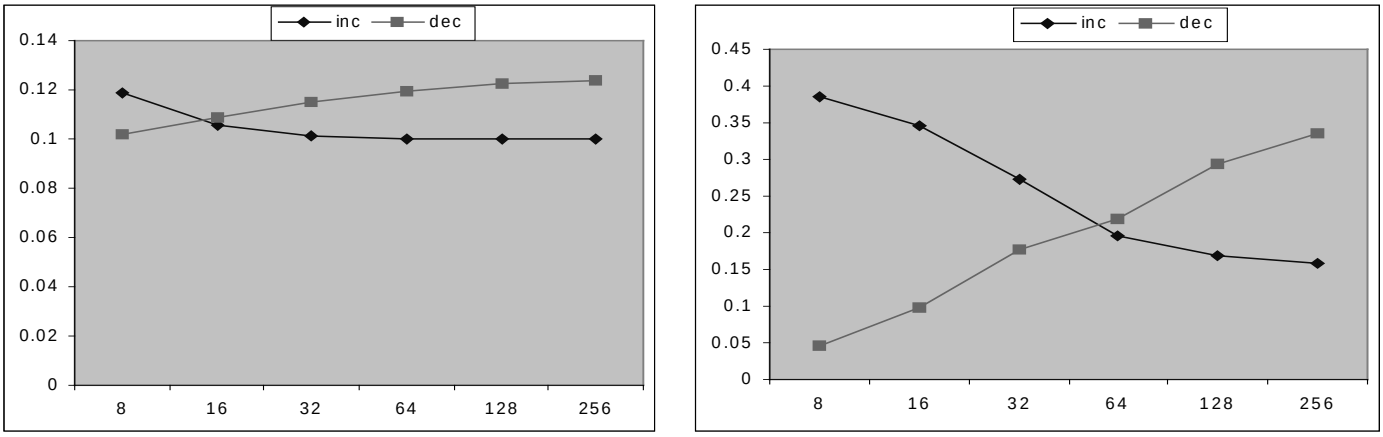


a) b)

Figure 7. Average cache line size for adaptive organization for 8KB (a) and 16KB (b) caches

Still, the average line size for benchmarks such as SC and FPPPP should become larger under adaptivity since the fixed-size line of larger size have lower miss rate. This is the place where our algorithm needs improvement.

While the average line size may be relatively constant with adaptivity, it does not mean that the change in size is infrequent or that the line size is approximately the same throughout program execution. `Figure 8` shows the frequency of line size change which occurs during the execution of two benchmarks, GCC and FLOW52. The adjustment is frequent, with approximately 25% (FLO52) and 40% (GCC) of all the lines fetched either increased or decreased on replacement. As can be expected, the relative number of increases vs decreases changes with the initial virtual line size.



a) b)

Figure 8. Fraction of miss-fetches producing a line size change for GCC (a) and (b) EFLUX

The last performance metric presented is the memory *fetch* traffic. 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 results justify the

effort. The cache to memory "traffic" generated in each case is presented in Figure 9. The traffic counts the total number of data bytes moved from memory by miss fetches. For each line size, the results are normalized to the case of fixed-size line of the same size. The effect of adaptivity is very pronounced, it automatically reduces the utilization of the memory interface, one of the critical memory hierarchy resources.

The results are dramatic, with the traffic 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. FLO52 shows a somewhat slower rate of decrease but we have already explained why its behavior differs from other benchmarks. In general, starting at 32B initial line size, the total traffic is, on average, close to ½ of that for same fixed-size case and continues to decrease. The results are not entirely surprising given the average line size observed for various codes (see Figure 7).

An absolute level of memory traffic for various line sizes can be seen in Figure 10. For comparison with existing architectures, it is normalized to the 32B fixed line traffic for each benchmark. The traffic is quite flat for all benchmarks except FLO52. The decrease can be in excess of 50% over the widely-used 32B line size.

The following overall strategy for configuring an adaptive cache system may be indicated, given the relatively constant traffic levels. Use a large initial line size since the miss rate for the adaptive case is either flat across the line sizes or is best for very large line sizes, 128 or 256B, while traffic is rather flat. Adaptivity will bring it down

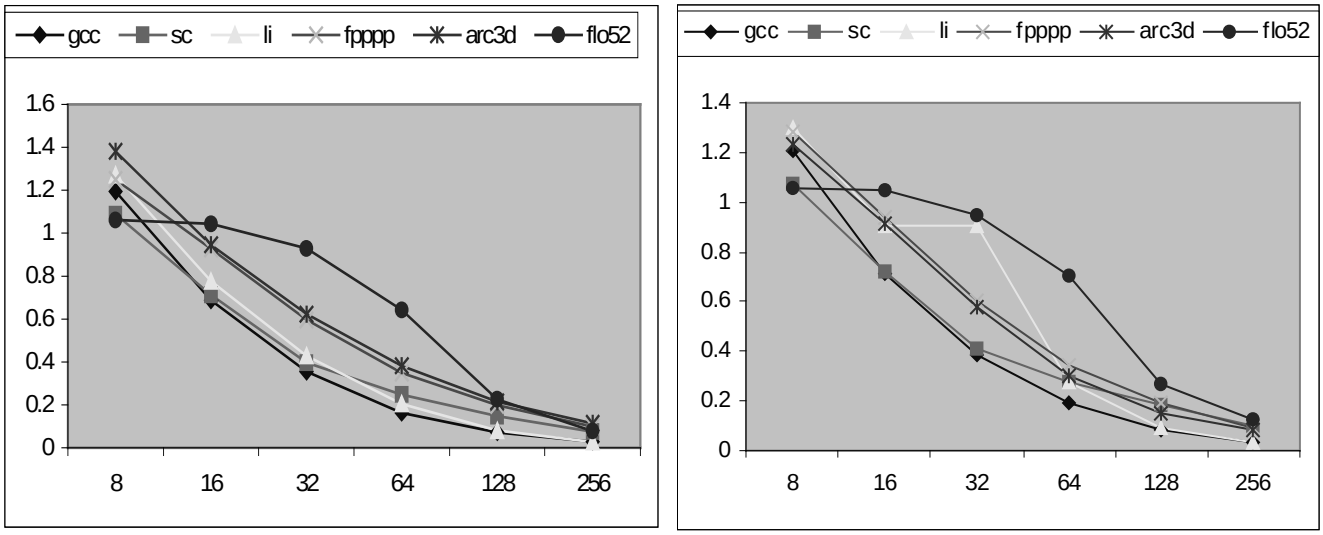


Figure 9. Memory fetch traffic for 8KB (a) and 16KB (b) caches normalized to same-size fixed case

to an average lower size while keeping the lowest traffic possible. A further study of the best way to select the initial line size is clearly indicated.

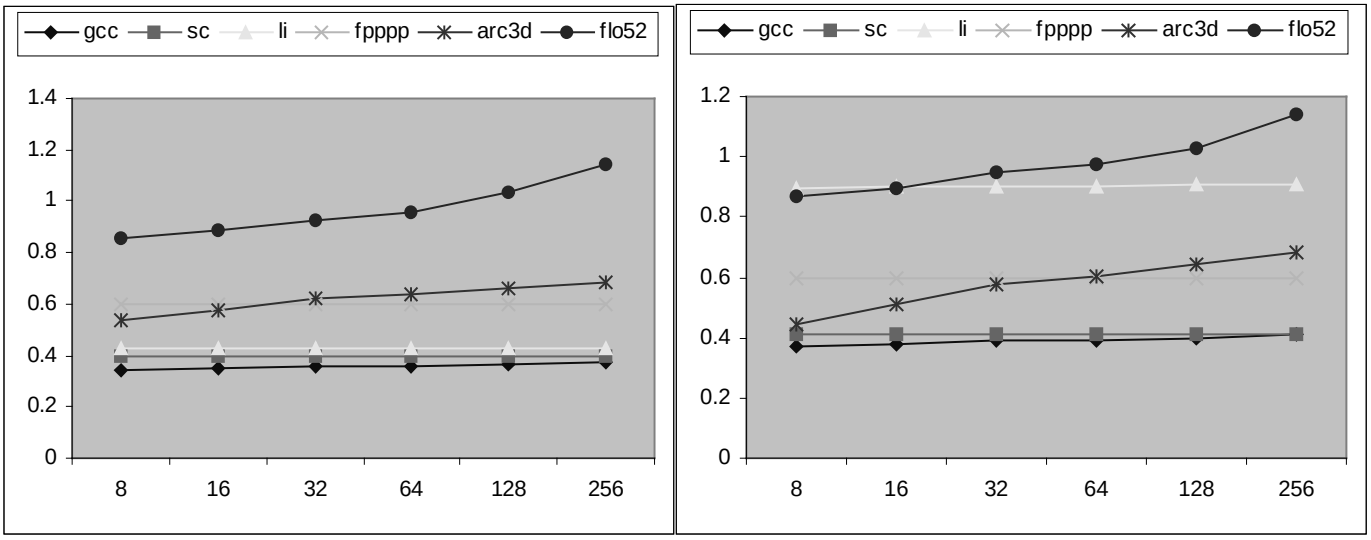


Figure 10. Fetch traffic normalized to the 32B fixed-size case for 8KB (a) and 16KB (b) caches

Another view of the memory traffic reduction for 8KB cache with 32B line is shown in `Figure 11`. The traffic, normalized to the fixed-size line of 32B, changes significantly within an application as well. 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 (100K memory references) to another in GCC, showing the adaptivity adjustment to occur fast enough to make an "instantaneous" effect.

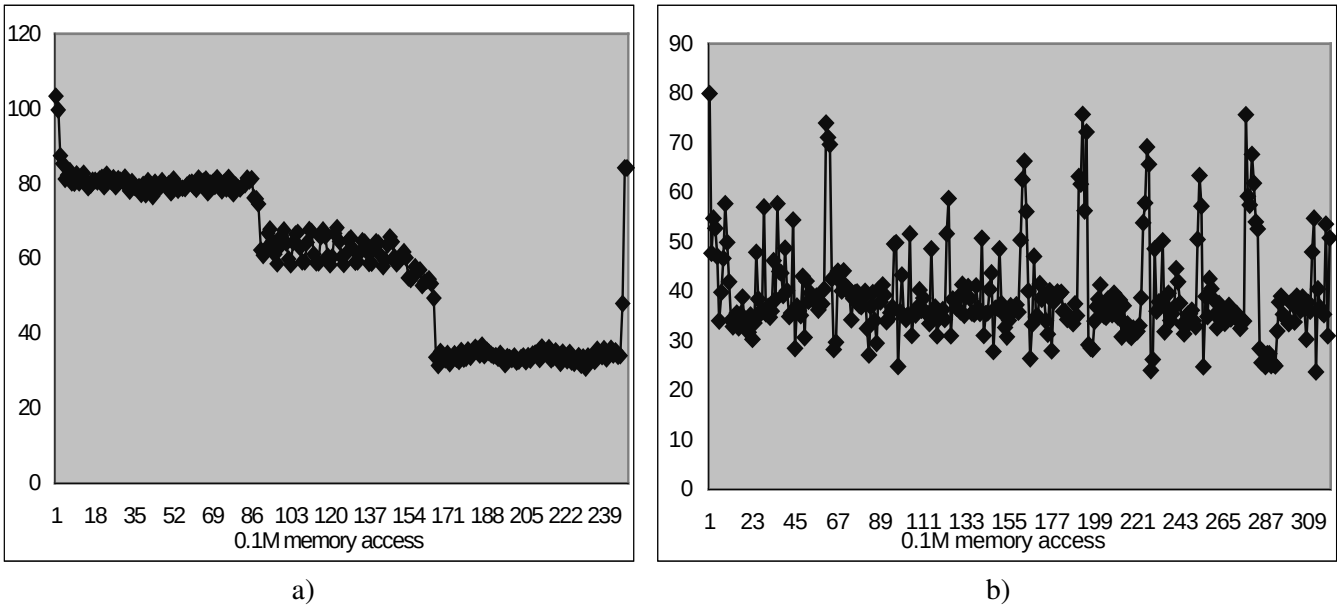


Figure 11. Time-domain normalized fetch traffic for ARC3D (a) and GCC (b) for 8KB cache, 32B line

Discussion

The 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 re-visited to explore 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 more difficult to eliminate, but one can probably set the minimal line size to be 16B without noticeable loss of performance. In fact, the total number of allowed line size can be restricted to 16, 64, and 256B. 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. We are currently investigating some of these alternatives.

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 perhaps 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, standarad cache. Also, for codes that cannot fit into the L2 cache the approach may be worthwhile. This is difficult to evaluate because of very large simulation times required.

Regardless of the adaptivity in the L2 cache, it can be used to keep the L1 line size instead of the memory when the L1 line is replaced. 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 an interesting one.

The effect of adjusting the algorithm in `Figure 3`, line 10, to forgo the line size decrease when the incoming line is smaller needs to be investigated. Our preliminary results show that this leads to increased memory traffic without a noticeable improvement in miss rates. However, this needs to be re-examined in the context of improving the overall algorithm so it can handle benchmarks, such as SC and FPPPP, much better.

Finally, let us compare and differentiate our work with [KuWi98]. Their idea is to use a large, fixed-size line but only fetch words that are going to be used. They fetch a variable subset 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 variable size lines and predict the size of a line rather than which words are use in a large, fixed-size line. Our predictor is simpler and requires very little hardware. Direct comparison of results is, unfortunately, not possible for several reasons:

- different instruction sets and compilers were used
- different associativity and write policy were used

However, 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.

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 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 choice of the exact moment to increase the size is an important parameter of the algorithm and can have a major effect on performance. The over-riding criterion for selection of algorithm features presented in the paper was the minimization of the memory traffic.

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

The adaptive approach provides a mechanism that can achieve a good balance between the miss rate and the memory traffic. The results clearly show the feasibility of the approach. There are a number of different design choices and the choice can significantly affect performance. Further research is needed to clarify the choices and obtain a higher and more uniform performance improvement.

References

[Albo98] D. H. Albonesi, "Dynamic IPC/Clock Rate Optimization", International Symposium on Computer Architecture, pp. 282-292, June 1998.

[ChKi92] A. Chien and J. Kim, "Planar Adaptive Routing: Low-cost Adaptive Networks for Multiprocessors", International Symposium on Computer Architecture, pp. 268-277, July 1992

[DaAo93] 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.

[DaDS93] Fredrik Dahlgren, Michel Dubois and Per Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors", International Conference on Parallel Processing, Aug, 1993

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

[GoVe94] Edward H. Gornish and Alex Veidenbaum, "An Integrated Hardware/Software Data Prefetching Scheme for Shared-Memory Multiprocessors", Proc. 1994 Int'l Conference on Parallel Processing, Aug. 1994.

[Inte93] Pentium™ Processor User's Manual, Intel Corporation, 1993

[JuSN98] T. Juan, S. Sanjeevan, and J. Navaro, "Dynamic History Length Fitting: a Third Level of Adaptivity for Branch Prediction", pp.155-166, International Symposium on Computer Architecture, July 1998

[KaKM98] K. Inoue, K. Kai, and K. Marukami, "High Bandwidth, Variable Line-Size Cache Architecture for Merged DRAM/Logic LSIs", IEICE Transactions on Electronics, Vol. E81-C No. 9, pp. 1483-1447, Sept. 1998.

[MIPSR3K] MIPS R3000 hardware manual, MIPS Corporation

[MNKH96] T. Matsumot, K. Nishimura, T. Kudoh, K. Hiraki, H. Amano, and H. Tanaka, "Distributed Shared Memory Architecture for JUMP-1: A General-Purpose MPP Prototype", Proc. 2nd International Symposium on Parallel Architecures, Algorithms, and Networks, IEEE Computer Society Press, pp. 131-137, June 1996

[TuVe94] Steve Turner and Alex Veidenbaum, "Scalability of the Cedar System", Supercomputing'94

[VeFo94] Jack E. Veenstra and Robert J. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors", Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94), page 201-207, January 1994

[KuWi98] S. Kumar and C. Wilkerson, "Exploiting Spatial Locality in Data Caches Using Spatial Footprints", International Symposium on Computer Architecture, pp. 357-368, June 1998

[YePa91] T.-Y. Yeh and Y. N. Patt. "Two Level Adaptive Branch Prediction." 24th ACM/IEEE International Symposium on Microarchitecture, Nov. 1991.