

Cache With Adaptive Fetch Size ¹

Weiyu Tang Alexander Veidenbaum
Alexandru Nicolau Rajesh Gupta

ICS Technical Report

Technical Report #-00-16
April 2000

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine
{wtang, alexv, nicolau, rgupta}@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine

¹This work was supported in part by DARPA ITO under DIS program.

Abstract

Current cache designs support only one fixed line size. Fixed line size limits cache's ability in spatial/temporal locality utilization. In this report, we present a cache design with multiple fetch sizes for better cache performance. The optimal fetch size is predicted based on memory access behavior to exploit changing application locality. Overall, a better performance is achieved by this cache design. Experimental results on SPEC95 benchmarks show that this cache design significantly reduces cache miss rate. It achieves as much as 24 percent miss rate reduction for L1 caches and 57 percent miss rate reduction for L2 caches. At the same time, the increase in traffic is small, as little as 22 percent between L2 and L1 cache and 32 percent between L2 and the main memory.

Contents

1	Introduction	1
2	Related Work	2
3	AFS Cache	3
4	Locality-based Fetch Size Adaptation	5
4.1	Spatial Locality Detection	5
4.2	Prediction with Fixed Thresholds	5
4.3	Prediction with Adaptive Thresholds	6
4.4	Hardware Cost	8
5	Sampling-based Fetch Size Adaptation	8
5.1	Intuition	8
5.2	Sampling Algorithm	8
5.3	Hardware Cost	9
6	Performance	10
6.1	Experimental Setup	10
6.2	Miss rate	11
6.3	Normalized Traffic	13
6.4	Comparison with the ALS Cache	15
7	Conclusion	16
	References	16

List of Figures

1	Optimal fetch size for SPEC95 benchmarks (32KB cache).	1
2	Optimal fetch size for IJPEG (32KB cache).	2
3	Relationship between Physical Cache Line and Virtual Cache Lines	4
4	Next fetch size prediction with fixed thresholds	6
5	Next fetch size prediction with aging thresholds	7
6	Sampling overview	9
7	Miss rate with adaptation interval of 1M and 100K memory accesses	11
8	Miss rate with different fixed thresholds in the locality algorithm	11
9	Miss rate with fixed threshold and aging threshold in the locality algorithm	12
10	Miss rate reduction in a 32KB L1 AFS cache	12
11	Miss rate reduction in a L2 256KB AFS cache	13
12	Normalized traffic between a L1 32KB cache and a L2 256KB cache	14
13	Normalized traffic between a L2 256KB cache and the main memory	14
14	Miss rate reduction in a L1 32KB cache	15
15	Miss rate reduction in a L2-256KB cache	15

List of Tables

1	SPEC95 Benchmarks	10
---	-----------------------------	----

1 Introduction

In current cache designs, a cache consists of multiple lines of equal size. This size is used to map addresses to cache lines. For a cache with a fixed line size, the determination of the line size is based on the spatial and temporal locality of average benchmarks. Due to variations in locality in different applications and in different regions of an application, fixed line size limits cache’s ability in locality utilization.

In our previous research [12], we have proposed an **adaptive line size (ALS)** cache to take advantage of the changing locality in applications. Large line sizes are used when good spatial locality is detected and small line sizes are used when poor spatial locality is detected. For most applications, the miss rate of an ALS cache is smaller than the optimal miss rate of a **fixed line size (FLS)** cache.

In a traditional cache, the fetch size is equal to the cache line size and one miss-fetch fills one cache line. The fetch size can also be different from the line size and one miss-fetch can fill multiple cache lines. We conjecture that adapting fetch size can achieve the same level of cache performance as adapting the cache line size. A large fetch size can be used in applications with good spatial locality and a small fetch size can be used in applications with poor spatial locality. A cache with variable fetch sizes is called **adaptive fetch size cache**.

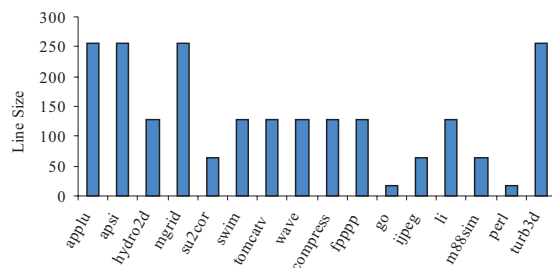


Figure 1: Optimal fetch size for SPEC95 benchmarks (32KB cache).

Figure 1 shows the optimal cache fetch size for SPEC95 benchmarks. The optimal fetch size is the size that results in minimal miss rate. Fetch size 16B, 32B, 64B, 128B and 256B is the optimal for 2, 3, 0, 7 and 4 benchmarks. This figure clearly demonstrates that different benchmarks have different optimal fetch size.

Figure 2 shows the optimal fetch size in time for *IJPEGE*. Each point in the figure is the optimal fetch size during an interval of one million memory accesses. Fetch size 32B, 64B and 128B is the optimal for 25, 47 and 28 percentage of the time respectively. This figure demonstrates the need for fetch size adaptation in different regions of an application.

We have investigated two strategies for fetch size adaptation, sampling-based and locality-based. In sampling-based adaptation, several possible fetch sizes are profiled each for a short period at the start of an interval. Then the fetch size with the minimal miss rate is selected as the fetch size for the rest of the interval. In locality-based adaptation, the fetch size during an interval is based on the aggregate locality utilization of the cache during the previous interval.

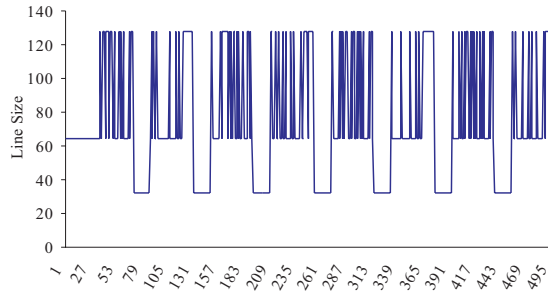


Figure 2: Optimal fetch size for IJPEg (32KB cache).

The rest of this report is organized as follows. Related work is discussed in Section 2. AFS cache is described in Section 3, followed by two strategies for fetch size adaptation in Section 4 and Section 5. Cache performance and system performance are shown in Section 6. This report is concluded with future work in Section 7.

2 Related Work

Previous research has investigated the exploitation of changing application locality for better cache performance and low traffic. In [16, 12], we have proposed to adapt cache line size based on the spatial locality where different cache lines can have different sizes. If two neighboring cache lines have neighboring tags, there is good spatial locality between these two lines. These lines can merge into a line of double size so that one miss-fetch will bring both lines into the cache and the number of cache misses is reduced. When a cache line is to be replaced, if half of the line is not used, then either there is not enough spatial locality in this line or this line conflicts with another line. This line is partitioned into two lines to prevent cache pollution and reduce traffic between the cache and the next level memory hierarchy. The small line size can increase cache utilization and improve temporal locality.

In spatial footprint [7], several neighboring caches lines are fetched at the same time if there are spatial locality among them. This approach is similar to adaptive line size cache in that both predict spatial locality among cache lines. But the predictors are different. The predictor in spatial footprint is complicated and instruction address is used. Thus it may not be used for off-chip caches because the instruction address may not be available.

[17] also investigates the use of different cache line sizes for different load instructions. Loads are classified into normal loads and superloads. A superload is used for data with good spatial locality. The line size for a superload is four times the line size for a normal load. Two approaches for loads classification have been proposed, the offline profiling approach and the online prediction approach. The online approach is similar to our approach used in [16, 12] because the spatial locality among neighboring lines is predicted. However, our approach is simpler and can support multiple line sizes; while the online approach in [17] only supports two line sizes.

In a sector cache [9, 13], a cache sector consists of several contiguous cache lines. All the lines in a sector share a tag, but each line has its own coherency and valid bits. The tag array size is

significantly lower than that in a traditional cache. The transfer granularity between the cache and the next level memory hierarchy is one cache line. One drawback of the sector cache is cache underutilized. If the optimal line size is smaller than the sector size, then only a fraction of the cache will hold valid data.

Adaptivity has also been applied in other 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]. [14] shows that "optimal" limit varies and suggests 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, 8].
- *Adapting branch history length* in branch predictors was proposed in [6] because the optimal history length was shown to vary significantly among programs.
- *Adaptive page size* has been proposed in [11] to improve the page management overhead and it is used in to reduce the TLB and memory overhead in [10].
- *Adaptive adjustment of data prefetch length* in hardware was shown to be advantageous [2], while in [4] the prefetch lookahead distance was adjusted dynamically either purely in hardware or with compiler assistance.

3 AFS Cache

In an AFS cache, a fetch size is a multiple of cache line size and is a power of two. Changing a FLS cache to an AFS cache only needs small hardware support. Instead of hardwiring the fetch size as line size, the fetch size is stored in a register, which can be set based on application locality. Some logic is also added so that several cache lines can be filled on a miss-fetch.

Similar to a FLS cache, an AFS cache consists of several lines (called physical cache lines) of equal size. A physical cache line (PCL) is an atom element in cache operations and the PCL size is used in the tag generation, and cache indexing. During a cache access, an address is partitioned into three fields:

$$(tag, index, offset).$$

Suppose the address width is A bits. For a 2^W -way set-associative cache with cache size 2^C bytes and line size 2^L bytes, the size of field tag , $index$ and $offset$ is $(A + W - C - 3)$, $(C - L - W)$ and $(L + 3)$ bits respectively. A PCL can be uniquely identified by a pair:

$$(way_num, line_num).$$

On a miss-fetch, multiple continuous PCLs are filled. These PCLs form a virtual cache line (VCL). The VCL size is equal to the fetch size and one miss-fetch will fill one VCL. When the fetch size is 2^V times the PCL size, a VCL can be uniquely identified by a triplet :

$$(way_num, i * 2^V, (i + 1) * 2^V - 1)$$

where $(i * 2^V)$ is the line number of the first PCL in this VCL and $((i + 1) * 2^V - 1)$ is the line number of the last PCL in this VCL.

Two VCLs $(way_num1, i * 2^v, (i + 1) * 2^v - 1)$ and $(way_num2, j * 2^v, (j + 1) * 2^v - 1)$ are called “neighboring” VCL if they are part of a large VCL with double line size. That is, they are “neighboring” if there exists an integer k such that one of the following conditions is satisfied:

- $i == 2 * k$ AND $j == (2 * k + 1)$
- $i == (2 * k + 1)$ AND $j == 2 * k$

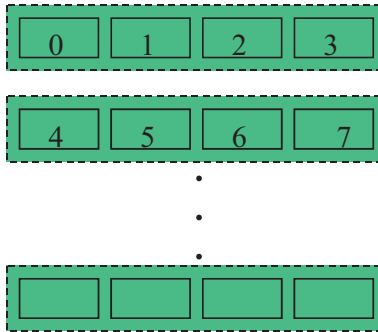


Figure 3: Relationship between Physical Cache Line and Virtual Cache Lines

Figure 3 shows the relationship between PCLs and VCLs in a direct-mapped cache. In this example, the fetch size is four times the cache line size. VCL (0,0,3) consists of PCLs (0,0), (0,1), (0,2) and (0,3). VCL (0,4,7) consists of PCLs (0,4), (0,5), (0,6), and (0,7). VCL (0,0,3) and VCL (0,4,7) are “neighboring” VCLs because they are part of a VCL (0,0,7) whose size is eight times the cache line size.

Suppose there are two caches A and B. Cache A is a FLS cache and cache B is an AFS cache. The line size of cache A is equal to the fetch size of cache B. There is an one-to-one mapping between one PCL in cache A and one VCL in cache B. It can be proven by induction that:

If both caches are initially empty and the sequence of memory accesses to both caches are same, then both caches will always have the same data.

As both caches always have the same data, a memory access will always hit or miss in both caches. Thus the performance of an AFS cache with fetch size lB is the same as the performance of a FLS cache with line size lB . By changing the fetch size dynamically, we are effectively having a cache with multiple physical line sizes.

4 Locality-based Fetch Size Adaptation

The spatial locality in a program has a direct impact on the cache performance. For a program with good spatial locality, large cache fetch size is preferred. Similarly, for a program with poor spatial locality, small fetch size is preferred.

Memory accesses are divided into intervals in time. The fetch size remains unchanged during an interval and the aggregate spatial locality during this interval is monitored. Then at the end of an interval, fetch size for the next interval is predicted using the detected spatial locality. To simplify the prediction, the candidates for next fetch size are: current fetch size, the immediately larger fetch size and the immediately smaller fetch size.

4.1 Spatial Locality Detection

Spatial locality can be detected as follows:

- good spatial locality

For two “neighboring” VCLs, if their tags are equal, then there is good spatial locality between them. A larger fetch size can fill all the PCLs in these two VCLs and can potentially eliminate one cache miss.

- poor spatial locality

When a VCL is to be replaced, if either the first half or the second half of the PCLs in this VCL are unused, then this VCL doesn’t have good spatial locality and unused PCLs are polluting the cache. A smaller fetch size can reduce cache pollution and increase cache utilization.

Spatial locality detection can be done in parallel with miss fetching. Cache hit time, which is on one of the critical paths in a processor design, is not affected.

4.2 Prediction with Fixed Thresholds

At the end of an interval, the fetch size for the next interval is predicted based on the aggregate spatial locality for all VCLs during this interval. The aggregate spatial locality can be measured using the following two parameters:

$inc\%$ the percentage of VCLs that show good spatial locality.

$dec\%$ the percentage of VCLs that show poor spatial locality.

Figure 4 shows an algorithm for next fetch size prediction. When $inc\%$ is larger than a threshold, the fetch size will double for the next interval. When $dec\%$ is larger than a threshold, the fetch size will decrease to half for the next interval. The parameters inc_{thresh} and dec_{thresh} are fixed in this algorithm. Thresholds range from 0.5 to 0.7. The larger a threshold, the more difficult it is to increase/decrease the fetch size. In contrast, the smaller the threshold, the easier it is to increase/decrease the fetch size.

Given:

```
incthresh, decthresh, maxfetch_size, minfetch_size
fixed_fetch_line_predict(fetch_size, inc%, dec%)
BEGIN
1. If inc% > incthresh Then
2.     If fetch_size < maxfetch_size Then
3.         fetch_size = fetch_size * 2
4.     Endif
5. Else if dec% > decthresh Then
6.     If fetch_size > minfetch_size Then
7.         fetch_size = fetch_size / 2
8.     Endif
9. Endif
END
```

Figure 4: Next fetch size prediction with fixed thresholds

4.3 Prediction with Adaptive Thresholds

We conjecture that different applications favor different thresholds. It would be better if thresholds could also be adaptable. Therefore, an algorithm based on an aging mechanism is described in Figure 5 to adaptively change the thresholds.

In algorithm *aging_fetch_line_predict*, *inc_{thresh}* and *dec_{thresh}* are initialized to *mid_{thresh}* and can change from *max_{thresh}* to *min_{thresh}*. There are three stages in this algorithm: validation stage from line 1 to line 10, fetch size prediction stage from line 11 to line 23, and threshold aging stage from line 24 to line 29. At the fetch size prediction stage, the next fetch size is predicted the same way as in algorithm *fixed_fetch_line_predict*. At the threshold aging stage, if the fetch size for the next interval is predicted to be same as current fetch size, then *inc_{thresh}* and *dec_{thresh}* will decrease by *aging_{rate}*. Because of aging, *inc_{thresh}* and *dec_{thresh}* can have values as small as *min_{thresh}*. This can potentially increase locality utilization by allowing more number of fetch size adaptation.

A disadvantage of threshold aging is that it is easier to make wrong fetch size prediction, which may degrade the cache performance. Therefore, when the fetch size in current interval is different from the fetch size in the previous interval, the cache performance of these intervals are compared at the validation stage. If fetch size adaptation results in poorer cache performance in current interval, then fetch size in the previous interval will be used in the next interval. The corresponding *inc_{thresh}*/*inc_{thresh}* will also be set to *max_{thresh}* so that it will be more difficult (take more time) to make future fetch size change in the same direction.

Given:

max_{thresh} , min_{thresh} , mid_{thresh} , $aging_{rate}$, max_{fetch_size} , min_{fetch_size}

$aging_fetch_line_predict(inc_{thresh}, dec_{thresh}, cur_{miss_rate}, prev_{miss_rate})$

BEGIN

1. If ($fetch_size$ increased last time) AND ($cur_{miss_rate} > prev_{miss_rate}$) Then
2. $fetch_size = fetch_size/2$
3. $inc_{thresh} = max_{thresh}$
4. return
5. Endif
6. If ($fetch_size$ decreased last time) AND ($cur_{miss_rate} > prev_{miss_rate}$) Then
7. $fetch_size = fetch_size * 2$
8. $dec_{thresh} = max_{thresh}$
9. return
10. Endif
11. If $inc\% > inc_{thresh}$ Then
12. If $fetch_size < max_{fetch_size}$ Then
13. $fetch_size = fetch_size * 2$
14. $inc_{thresh} = mid_{thresh}$
15. return
16. Endif
17. Else if $dec\% > dec_{thresh}$ Then
18. If $fetch_size > min_{fetch_size}$ Then
19. $fetch_size = fetch_size/2$
20. $dec_{thresh} = mid_{thresh}$
21. return
22. Endif
23. Endif
24. If $inc_{thresh} > min_{thresh}$ Then
25. $inc_{thresh} = inc_{thresh} - aging_{rate}$
26. Endif
27. If $dec_{thresh} > min_{thresh}$ Then
28. $dec_{thresh} = dec_{thresh} - aging_{rate}$
29. Endif

END

Figure 5: Next fetch size prediction with aging thresholds

4.4 Hardware Cost

The following additional is needed to support locality detection:

- a register to store interval length;
- three registers to store threshold values;
- one register to store the aging rate;
- two counters to store the number of VCLs that show good/poor spatial locality;
- a comparator to determine whether two VCLs have equal tags.

5 Sampling-based Fetch Size Adaptation

Sampling-based fetch size adaptation predicts the fetch size for a long interval based on the optimal fetch size over several small intervals.

5.1 Intuition

Although the optimal fetch size changes over time, a fetch size may stay optimal for an extended period. As can be seen in Figure 2, 64B line size is optimal for the first 60 million memory accesses. We can use the optimal fetch size in a short interval to predict the optimal fetch size in a long interval.

In this report, we are concentrating on improving cache performance and a fetch size is considered optimal if it results in minimal miss rate. It is impossible to measure multiple miss rates for multiple fetch sizes simultaneously. The following assumption can be used to approximate the optimal fetch size— *the locality in a program will not change dramatically during a short period.*

Hence the optimal fetch size can be obtained in the following way:

1. select a list of fetch sizes
2. use each fetch size for a short period and obtain the miss rate
3. select the fetch size with the minimal miss rate as the optimal fetch size

5.2 Sampling Algorithm

Figure 6 shows how to adapt fetch size over time. Memory accesses in time are divided into adaptation intervals. Each adaptation interval has two phases, a sampling phase and a stable phase. The sampling phase consists of several sampling intervals and different fetch sizes are used in each sampling interval. The optimal fetch size during the sampling phase is used as the fetch size for the stable phase.

As there are two types of intervals used in the algorithm design, the lengths of the intervals and their relationship should be determined carefully. Sampling interval should not be very short

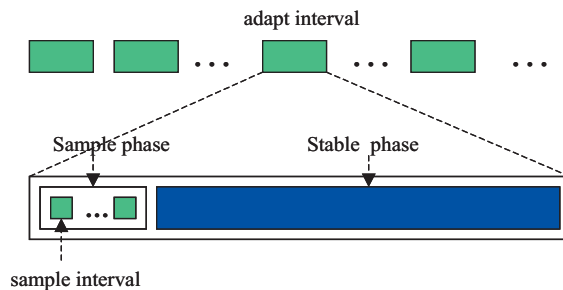


Figure 6: Sampling overview

so that the effect of the fetch size on the cache miss rate can be observed. On the other hand, the sampling interval should not be very long, otherwise the locality may change between different sampling intervals and the selected optimal fetch size may not be optimal. In our experiments, the sampling interval is an order of magnitude larger than the number of cache lines.

There are performance penalties in the sampling phase. Several fetch sizes are used and only one of them is the optimal. In a sampling interval with a non-optimal fetch size, the miss rate is higher than that in the interval with the optimal fetch size. The adaptation interval should be much larger than the sampling interval to amortize the performance penalties during the sampling phase. In our experiments, the adaptation interval is two orders of magnitude larger than the sample interval.

We propose the following two sampling algorithms:

- *all sampling* (*samp-a*): all the possible fetch sizes are used in the sampling phase
- *neighbor sampling* (*samp-n*): current fetch size, the immediately larger fetch size and the immediately smaller fetch size are used in the sampling phase

There are tradeoffs in these algorithms. For example, *samp-a* can adapt to the optimal fetch size faster because all the possible fetch sizes are sampled. But the penalty is also high by using a larger number of non-optimal fetch sizes. In contrast, *samp-n* adapts to the optimal fetch size slower because at most three fetch sizes are sampled. But fewer number of non-optimal fetch sizes can result in low penalty. The tradeoffs will be further investigated in Section 6.

5.3 Hardware Cost

The additional hardware to support sampling-based algorithms is:

- two registers to store interval lengths: one for the sampling interval, the other for the adaptation interval;
- two counters, one for the sampling interval and the other for the adaptation interval;
- several registers to record the performance (miss rate) statistics for each fetch size.

Program Name	Input	Instr (M)	Memory (M)
APPLU	applu.in	1609	500
APSI	apsi	1472	500
HYDRO2D	hydro2d.in	2109	500
MGRID	mgrid.in	1542	500
SU2COR	su2cor.in	4233	500
SWIM	swim.in	1324	500
TOMCATV	tomcatv.in	1384	500
WAVE	wave5.in.ref	1686	500
COMPRESS	bigtest.in	1255	500
FPPPP	natoms.in	1472	500
GO	null.in	2109	500
IJPEG	specmun.ppm	1542	500
LI	au.lsp, boyer.lsp,...	1094	500
M88KSIM	ctl.in	1449	500
PERL	scrabbl.pl	1384	500
TURB3D	turb3d.in	1255	500

Table 1: SPEC95 Benchmarks

6 Performance

6.1 Experimental Setup

The system architecture used in this study consists of a processor, L1 cache, L2 cache and memory. Both caches block on a cache miss. The cache uses write-back policy. The L1 cache size is 32KB and is direct-mapped. Possible fetch sizes are: 16B, 32B, 64B, 128B and 256B. The L2 cache size is 256KB and is direct-mapped. Possible fetch sizes are: 64B, 128B, 256B and 512B.

A set of SPEC95 benchmarks are simulated. The benchmark statistics are shown in Table 1. These benchmarks exhibit a sufficiently varied memory behavior to thoroughly check our architecture. All the benchmarks are used in the study of L1 cache. Only the first 8 benchmarks have meaningful L2 miss rate and they are used in the study of L2 cache and 2-level cache architecture.

The benchmarks are compiled for an R3000 processor using MIPS and MIPSPro compilers with the following flags: -n32 (MIPS-III instruction set, 32b executable) and -O2. We use MINT-3 [15] to model a single-issue, statically-scheduled processor. We have designed an execution-driven memory hierarchy simulator.

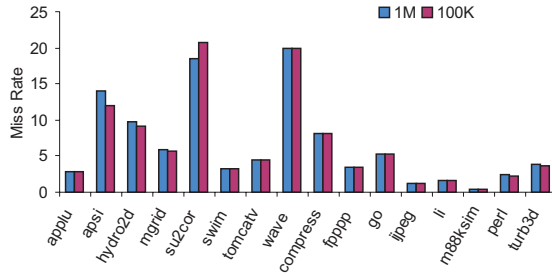


Figure 7: Miss rate with adaptation interval of 1M and 100K memory accesses

6.2 Miss rate

Figure 7 shows the miss rate with different adaptation interval length with the locality based algorithm. The cache size is 32KB. Both inc_{thresh} and dec_{thresh} are set to 0.7. For eight benchmarks—APPLU, SWIM, WAVE, COMPRESS, FPPPP, GO, IJPEG, LI, the miss rate is almost same regardless of interval length. SU2COR achieves slightly better miss rate with the 1M-interval. For the other six benchmarks, the miss rate obtained by 100K interval is slightly better. We conclude that the small 100K interval is better as it can quickly adapt to locality variation in applications.

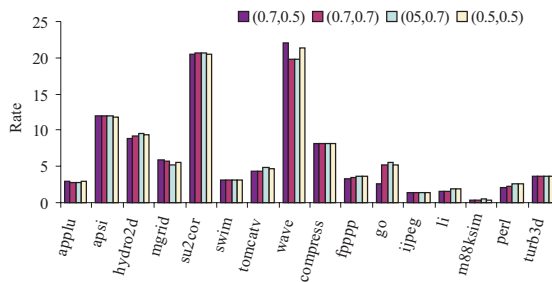


Figure 8: Miss rate with different fixed thresholds in the locality algorithm

Figure 8 compares the miss rate with different fixed thresholds in the locality algorithm. The label on each bar stands for $(inc_{thresh}$ and dec_{thresh} . For APSI, SU2COR, SWIM, COMPRESS, M88KSIM and TURB3D, the miss rate is almost same regardless of thresholds. For the other ten benchmarks, we can see noticeable differences in miss rate.

For example in GO, the miss rate with threshold (07, 05) is half that with other thresholds. It is easier to adapt to a small fetch size with large inc_{thresh} and small dec_{thresh} . Adaptation to small fetch sizes results in small miss rate because the optimal fetch size of GO is 16B as shown in Figure 1.

In case of WAVE, the second and third bars, which have low dec_{thresh} , are shorter than other bars. Low dec_{thresh} makes it difficult to adapt to a small fetch size and fetch size is large most

of the time. Large fetch size results in small miss rate because the optimal fetch size of WAVE is 128B.

In conclusion, high dec_{thresh} and low inc_{thresh} make it easy to adapt to a small fetch size. Low inc_{thresh} and high dec_{thresh} make it easy to adapt to a large fetch size.

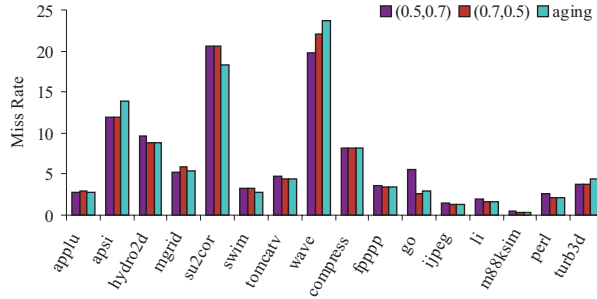


Figure 9: Miss rate with fixed threshold and aging threshold in the locality algorithm

Figure 9 shows the miss rate with fixed and aging threshold in the locality algorithm. The first two bars use fixed thresholds. The third bar uses aging threshold, where max_{thresh} , mid_{thresh} , min_{thresh} and $aging_{rate}$ is 0.7, 0.55, 0.4 and 0.01 respectively. Except for APSI, WAVE and TURB3d, the miss rate with aging threshold is close to or better than the optimal miss rate with fixed thresholds. This demonstrates that aging threshold is less biased toward either large line sizes or small line sizes.

Miss rate reduction, which is calculated using the following formula, is used to evaluate the effect of adaptivity:

$$reduction_{adapt}^{alg} = \frac{rate_{fixed}^{baseline} - rate_{adapt}^{alg}}{rate_{fixed}^{baseline}} * 100$$

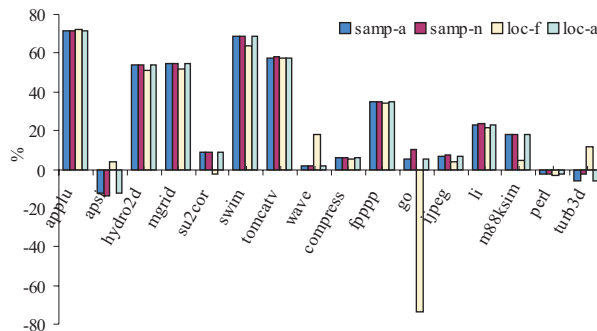


Figure 10: Miss rate reduction in a 32KB L1 AFS cache

Figure 10 shows the miss rate reduction in a 32KB L1 AFS cache. The baseline for comparison is a FLS cache with 32B line size. There are four bars for each benchmark. The first bar

corresponds to *all sampling* algorithm. The second bar corresponds to *neighbor sampling* algorithm. For sampling-based algorithms, the sample interval is 10000 memory accesses and the adaptation interval is 1,000,000 memory accesses. The third bar corresponds to *fixed threshold* algorithm. The inc_{thresh} and dec_{thresh} are both 0.7. The fourth bar corresponds to *adaptive threshold* algorithm. The max_{thresh} , mid_{thresh} , min_{thresh} and $aging_{rate}$ is 07, 0.55, 0.4 and 0.01 respectively.

For five benchmarks, APPLU, HYDRO2D, MGRID, SWIM and TOMCATV, all algorithms can reduce miss rate by more than 50 percent. For WAVE, COMPRESS, FPPPP, IJPEG, LI, and M88KSIM, the miss reduction ranges from 1 to 30 percent. For APSI, PERL and TURB3D, algorithms *samp-a*, *samp-n* and *loc-a* result in an increase in miss rate. We can see from Figure 1 that the optimal line size for APSI and TURB3D is 256B and the optimal for PERL is 16B. So all these benchmarks either have very good spatial locality or very poor spatial locality. Adapting fetch size is not necessary and adds overhead. *loc-f* still improves miss rates for APSI and TURB3D. In *loc-f*, because of high thresholds, it is difficult to make wrong fetch size decisions. Thus the overhead due to adapting fetch size is smaller.

The average miss rate reduction for *samp-a*, *samp-n*, *loc-f* and *loc-a* is 24, 25, 20 and 24 percent respectively. *loc-f* is the worst.

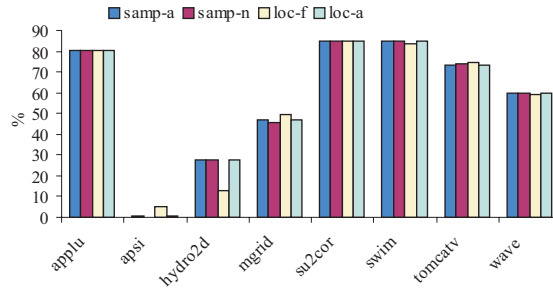


Figure 11: Miss rate reduction in a L2 256KB AFS cache

Figure 11 shows the miss rate reduction in a L2 256KB AFS cache. The baseline is FLS cache with 64B line size. All algorithms improve miss rates for every benchmark. For APPLU, SU2COR and SWIM, the miss rate reduction is as high as 80 percent. The miss reduction for APSI is small because there is not much variations in spatial locality in this benchmark and the largest fetch size is always the best. For all other benchmarks, the miss rate reduction ranges from 10 to 70 percent. The average miss reduction for all benchmarks is 57 percent. All the algorithms are effective in reducing L2 cache miss rate.

6.3 Normalized Traffic

The normalized traffic is calculated using the following formula:

$$traffic_{adapt}^{alg} = \frac{traffic_{fixed}^{baseline}}{traffic_{fixed}^{baseline}}$$

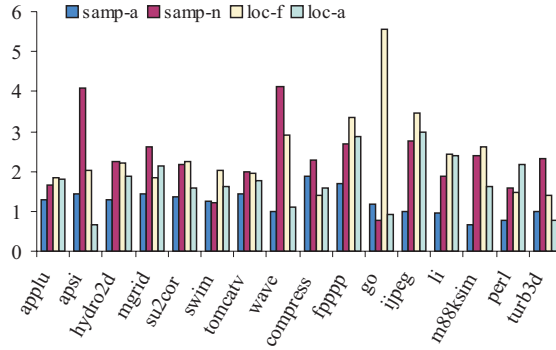


Figure 12: Normalized traffic between a L1 32KB cache and a L2 256KB cache

Figure 12 shows normalized traffic between a L1 AFS cache and a L2 FLS cache. The baseline is the traffic between a L1 FLS cache with line size of 32B and a L2 FLS cache. For some benchmarks, there are big differences in the normalized traffic when different algorithms are used. For example in GO, the normalized traffic is 0.76 for algorithm *samp-n*, but the normalized traffic is 5.5 for algorithm *loc-f*.

The average normalized traffic for all benchmarks is 1.22, 2.3, 2.42 and 1.74 for algorithm *samp-a*, *samp-n*, *loc-f* and *loc-a* respectively. Algorithm *samp-a* is effective in traffic control. On the contrary, algorithm *loc-f* is the worst.

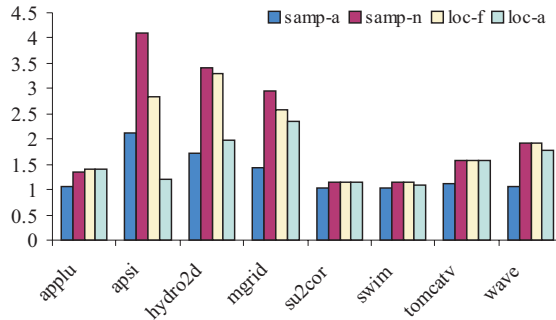


Figure 13: Normalized traffic between a L2 256KB cache and the main memory

Figure 13 shows normalized traffic between the main memory and a L2 AFS cache. The baseline is the traffic between the main memory and a L2 FLS cache with line size of 64B. Comparing to the normalized traffic between a L2 FLS cache and a L1 AFS cache, the differences in traffic for different algorithms are much smaller.

The average normalized traffic for all benchmarks are 1.32, 2.2, 1.98 and 1.56 for algorithm *samp-a*, *samp-n*, *loc-f* and *loc-a* respectively. Algorithm *samp-a* is again the best in traffic and algorithm *samp-n* is the worst here.

6.4 Comparison with the ALS Cache

We compare the effectiveness in miss rate reduction by an AFS cache and by an ALS cache. For the AFS cache, *samp-a* algorithm is used in fetch size adaptation. For the ALS cache, *NE-PL-IF-DF* algorithm is used in line size adaptation. The optimal miss rate for the FLS cache is used as baseline to compute the miss rate reduction.

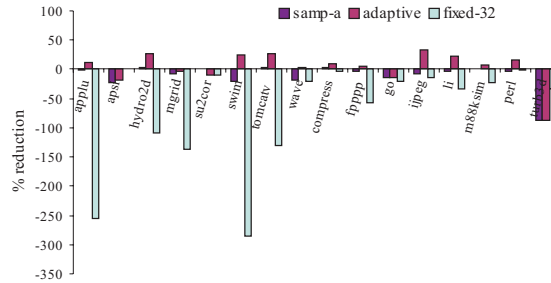


Figure 14: Miss rate reduction in a L1 32KB cache

Figure 14 shows the miss rate reduction for 32KB L1 caches. Both the AFS cache and the ALS cache have no improvement over the optimal miss rate of the FLS cache for four benchmarks. The ALS cache achieves a better miss rate than the optimal FLS cache in eleven benchmarks. The AFS cache achieves a better miss rate than the optimal FLS cache in only three benchmarks. However, both the ALS cache and the AFS cache are much better than the FLS cache with 32B line size, which on average increases 73 percent miss rate. (??FIGURE add fixed-32B miss rate)

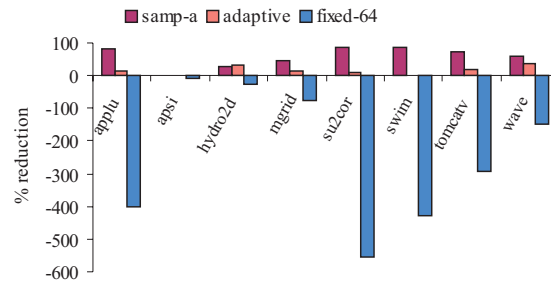


Figure 15: Miss rate reduction in a L2-256KB cache

Figure 15 shows the miss rate reduction for 256KB L2 cache. The ALS cache is better than the AFS cache in seven out of eight benchmarks. On average, the ALS cache and the AFS cache result in 15.6 and 3.18 percent miss reduction over the FLS cache with optimal fetch size. Both caches perform much better than the FLS cache with line size 64B, which on average increases miss rate by 242 percent.

Overall, the ALS cache performs better than the AFS cache. But the AFS cache can be implemented with much less complexity. Both caches perform much better than the FLS cache.

7 Conclusion

In this study, we have proposed and investigated a cache design with adaptive fetch size. With few modifications to conventional cache designs, this cache can achieve the same level of benefits as a cache with multiple cache line sizes to utilize the changing application locality.

We have proposed four algorithms to adapt the fetch size dynamically. The sampling-based algorithm *samp-all* is the best among them. It achieves on average 24 percent miss rate reduction and only 22 percent traffic increase over the FLS L1 cache with 32B line. The algorithms are more effective for the L2 cache than for the L1 cache. The *samp-all* algorithm in L2 on average results in 57 percent miss rate reduction and only 32 percent traffic increase over the FLS L2 cache with 64B line.

In this research, our focus was to optimize cache performance by fetch size adaptation. Fetch size adaptation can also be used to optimize traffic and to reduce power consumption. Different algorithms for fetch size adaptation are needed in these scenarios and this is the focus of our future research.

References

- [1] A. Chien and J. Kim. Planar adaptive routing: low-cost adaptive networks for multiprocessors. In *Int'l Symp. Computer Architecture*, pages 268–277, 1992.
- [2] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Int'l Conf. Parallel Processing*, 1993.
- [3] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Trans. Parallel and Distributed Systems*, 4:466–475, 1993.
- [4] E. H. Gornish and A. Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. In *Int'l Conf. Parallel Processing*, 1994.
- [5] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Int'l Symp. Computer Architecture*, pages 302–313, 1994.
- [6] T. Juan, S. Sanjeevan, and J. Navaro. Dynamic history-length fitting: a third level of adaptivity for branch prediction. In *Int'l Symp. Computer Architecture*, pages 155–166, 1998.
- [7] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Int'l Symp. on Computer Architecture*, pages 357–368, 1998.
- [8] T. Matsumoto, K. Nishimura, T. Kudoh, K. Hiraki, H. Amano, and H. Tanaka. Distributed shared memory architecture for JUMP-1. In *Int'l Symp. on Parallel Architectures, Algorithms, and Networks*, pages 131–137, 1996.
- [9] Motorola. *PowerPC 601, RISC Microprocessor User's Manual*, 1993.
- [10] T. Romer, W. Ohlich, A. Karlin, and B. Bershad. Reducing TLB and memory overhead using on-line superpage promotion. In *Int'l Symp. on Computer Architecture*, pages 176–87, 1995.
- [11] M. Talluri and M. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Int'l Conf. on Architecture Support for Programming Languages and Operating Systems*, pages 171–182, 1994.
- [12] W. Tang, A. Veidenbaum, A. Nicolau, and R. Gupta. Adapting line size cache. Technical Report ICS-99-56, University of California, Irvine, 1999.
- [13] Texas Instrument. *TMS390Z55 Cache Controller, Data Sheet*, 1992.

- [14] S. Turner and A. Veidenbaum. Scalability of the cedar system. In *Int'l Conf. on Supercomputing*, pages 247–254, 1994.
- [15] J. Veenstra and R. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Int'l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 201–207, 1994.
- [16] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Int'l Conf. on Supercomputing*, pages 145–154, 1999.
- [17] P. Vleet, E. Anderson, L. Brown, J. Baer, and A. Karlin. Pursuing the performance potential of dynamic cache line sizes. In *Int'l Conf. on Computer Design*, 1999.