UC Irvine
ICS Technical Reports

Title
Memory adaptation techniques : a unified overview across benchmark suites

Permalink
https://escholarship.org/uc/item/3xg3s8zv

Authors
Du, Haitao
D'Alberto, Paolo
Gupta, Rajesh

Publication Date
2001-08-13

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ICS

TECHNICAL REPORT

Memory Adaptation Techniques: A Unified Overview Across Benchmark Suites

Haitao Du, Paolo D'Alberto, Rajesh Gupta

August 13, 2001

Technical Report ICS-01-41

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949)824-1565
{hdu, paolo, rgupta}@ics.uci.edu

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Information and Computer Science
University of California, Irvine

Memory Adaptation Techniques: A Unified Overview Across Benchmark Suites

Haitao Du Paolo D'Alberto Rajesh Gupta
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425
{*hdu, paolo, rgupta*}*@ics.uci.edu*

Abstract

In this paper we present the results from an extensive comparison study of three R-tree packing algorithms, including a new easy to implement algorithm. The algorithms are evaluated using both synthetic and actual data from various application domains including VLSI design, GIS (tiger), and computational fluid dynamics. Our studies also consider the impact that various degrees of buffering have on query performance. Experimental results indicate that none of the algorithms is best for all types of data, but ours is best anyways because it is so cool.

RECEIVED
APR 15 2002
UCI LIBRARY

Contents

1 **Introduction** 1

2 **A Model of Cache Misses** 4
- 2.1 Miss Ratio . . . 5
- 2.2 Miss Discrimination by Type . . . 6
- 2.3 Application of Miss Discrimination . . . 8
- 2.4 Distribution of Misses over Time in MoM . . . 8
- 2.5 Analysis Limitations . . . 11

3 **Background on Adaptations** 11
- 3.1 Stream Buffer (SB) . . . 11
  - 3.1.1 Pre-fetch . . . 12
  - 3.1.2 Hardware pre-fetch . . . 13
  - 3.1.3 Stream Buffer . . . 16
- 3.2 Victim Cache (VC) . . . 18
- 3.3 Adaptive Line Size Cache (ALS) . . . 19
- 3.4 Adaptive Fetch Line Size Cache (AFL) . . . 20

4 **Benchmark Suites** 22
- 4.1 DIS Benchmark Suite . . . 23
- 4.2 DIS Stressmark Suite . . . 25
- 4.3 SPEC95 . . . 28
  - 4.3.1 SPEC CINT95 . . . 29
  - 4.3.2 SPEC CFP95 . . . 30

5 **Experimental Results** 31
- 5.1 Experiment Setup . . . 32
  - 5.1.1 Platform . . . 32
  - 5.1.2 Baseline Simulator . . . 32
  - 5.1.3 Adaptive Cache Modules . . . 33
  - 5.1.4 Compiler . . . 34
  - 5.1.5 Baseline Architecture . . . 34
- 5.2 Example Benchmark Analysis . . . 34
  - 5.2.1 Analysis of Matrix: why Stream Buffer Works . . . 34
  - 5.2.2 Analysis of Wave5: Why Victim Cache Works . . . 36
  - 5.2.3 Analysis of Applu: Why Adaptive Line/Fetch Size Caches Work . . . 37
- 5.3 Results per Benchmark Suite . . . 38
  - 5.3.1 Experimental Results for DIS Benchmark Suite . . . 38
  - 5.3.2 Experimental Results for DIS Stressmark Suite . . . 41
  - 5.3.3 Experimental Results for SPEC95 . . . 43

List of Figures

1 Average Miss Ratios 5
2 Miss Discrimination 7
3 MoM: Plate 5 Miss Distribution over Different Levels & Average Miss Ratio . . . 9
4 MoM:Plate 5 Instruction Issue Efficiency 9
5 MoM: Plate 3 Memory Distribution over Different Levels & Average Miss Ratio . 10
6 MoM: Plate 3 Miss Ratios using Different Adaptations 10
7 Example of When Compulsory and Capacity Misses Happen 12
8 Prefetching Optimization on the Previous Example 12
9 Finite State Machine of PC Based Prefetch 14
10 Example where Partition Based prefetch is effective 15
11 Example where PC based prefetch is more effective than Partition based prefetch 16
12 Example where Partition based prefetch is more effective than PC based prefetch 16
13 Memory Hierarchy with Stream Buffer 17
14 Example of when Single Way Stream Buffer fails and Multi-ways Stream Buffer is effective 18
15 Memory Hierarchy with Victim Cache 18
16 Average Miss Ratios of Adaptations on Matrix with input in2 35
17 Average Miss Ratios of Adaptations on Matrix with input m09 35
18 Average Miss Ratios of Adaptations on wave5 with reference input 36
19 Average Miss Ratios of Adaptations on wave5 with reference input 37
20 Average Miss Ratios of Adaptations on DIS Benchmark Suite 38
21 Average Miss Ratios of adaptations on Raytray 39
22 Average Miss Ratios of adaptations on IM 39
23 Average Miss Ratios of Adaptations on DIS Stressmark Suite 41
24 Avearge Miss Ratios of Adaptations on Pointer and Update 41
25 Kernel of Transitive_Closure 42
26 Average Miss Ratios of Adaptations on SPEC95 44
27 Average Miss Ratio Reduction of Different Adaptations 46

1 Introduction

As the gap between CPU speed and memory speed increases rapidly, a large percentage of application execution time is spent on memory accesses [23][22]. It becomes very challenging to feed a hungry processor with data and instructions from the memory hierarchy. This situation may be negatively exploited by applications with dominant memory activities, such as *Data Intensive Applications* [37]. Data Intensive Applications are characterized by large data sets challenging the capacity of caches, non-contiguous memory accesses challenging the associativity of caches, and frequent load and store instructions ($\frac{1}{2}$ or $\frac{1}{3}$ of the instructions are memory accesses) that set the memory accesses as dominant operations. Even if the miss ratio in these applications is relatively small (as small as 4%), the misses still result from a significant percentage of the overall instructions (i.e. 2%). When the time spent on a miss is two order of magnitude more than CPU operation cycle, the total memory access time is dominant. Thus the memory utilization is the performance bottleneck. There are three important research topics related to memory utilization:*asymptotic analysis, application engineering* and *architecture engineering.* Architecture engineering is the subject of this report.

The asymptotic analysis of application complexity must be revisited: the naive count of instructions is not sufficient to represent the execution time of an application. The topic has been investigated for twenty years. Hong and Kung [24] formalized the problem for the very first time and proposed a general approach to determine the lower bounds to the memory accesses of an algorithm. They propose a two-level-memory model describing the I/O complexity of main memory to and from disks: first level has zero-latency time and finite size; second level has constant-latency time and unlimited size. Algorithms are expressed by *Direct Acyclic Graphs (DAG).* The approach has been generalized in [3] and [44]. Recently, upper bounds as well as lower bounds are investigated and some initial results can be found in [6] [5]. In general, applications cannot be described as DAGs without simplifications[1]. Memory model cannot be implemented without simplification.

Application Engineering is a collection of mechanisms to reorganize the application so that the impact of misses is minimized. Given an application and an architecture, there are different ways where application engineering can be applied:

- The developer is aware of the architecture features and designs the application properly. The approach can achieve high performance but the application so obtained is not portable across different architectures (i.e. through *tiling* and data alignment [40]).

[1] i.e. input-dependent-loop

- The installation of an application is adaptive. The installation consists of the determination of the architecture parameters and then the source code is tailored upon the parameters and compiled. In practice, there may be different source codes for different architecture (see [57][19])

- The compiler is aware of the architecture features and it generates the proper executable. The source code does not change across architectures (i.e.*tiling* in [58][59]).

- The algorithm is memory hierarchy oblivious, the application is designed so that optimal performance is achieved without any a priori knowledge of the memory hierarchy (for a survey see [20] or [53][7] [14]).

The engineering of the memory architecture is the ability to tailor the configuration to the application needs. In fact, most of today's memory architectures are the result of the design tradeoff over a series of variables (such as system performance, cost, capacity, and bandwidth, etc.) in a very large design space. Not surprisingly, no fixed memory system is optimal for every application: different applications can exploit different performance and the same application with different inputs might exploit different performance [32]. Adaptive architecture is a promising solution because it is able to adapt to the application's memory behaviors.

In this report, we present a simple and unified overview of memory adaptations across different benchmark suites: the improvements, the pitfalls and a summary of what our group has learnt in the last two years of investigations (in particular [51][55][25]). We demonstrate that the adaptation approaches are in general very effective over a very large spectrum of applications. In practice, we investigate both the positive cases and negative cases. The positive cases stress out why memory adaptations are effective. The negative cases show what prevents them to be effective. For the negative cases, we claim:

- When the application does not have locality, adaptation approaches cannot be effective (see Section 2 and Section 5).

- Miss ratio is not a proper metric. It expresses the average behavior, but hides the information of the miss distribution (see Section 2.4).

Four memory adaptation approaches are investigated in this report: *Stream Buffer (SB)* [28], *Victim Cache (VC)* [28][50],*Adaptive Cache Line Size (ALS)* [51] and *Adaptive Fetch Size (AFL)*[52]. These adaptations deal with different types of cache misses: SB hides the latency of compulsory and capacity misses; VC removes conflict misses; ALS and AFL intend to remove conflict misses as well as compulsory and capacity misses.

These adaptation approaches were applied on top of a common baseline. In the following we describe briefly the mechanisms related to our implementations.

The SB has four hardware pre-fetch buffers in parallel, with each one being a vector composed of four elements. The element size is equal to the cache line size. During the execution of an application, the strides for memory references, either *PC based* [11] or *Partition based* [2], are monitored. If the stride is predicted as constant k, SB will pre-fetch up to four elements, which are k elements apart, into one vector. SB tends to reduce compulsory misses and capacity misses. In fact, it offers a simple and very effective memory adaptation approach to hide memory latency and thus improve performance. A L1 miss happens when the referenced data is in neither the first level cache nor the SB. From the experimental results, we show that the average miss ratio reduction is 52.3% using SB.

The VC is a vector of 32 elements, placed between the first level cache and the second level cache. It is a very small full associative cache. When an element is evicted from the first level cache, it goes into the VC, and then to the lower memory levels when evicted from VC. VC tends to reduce conflict misses. A miss happens when an element is in neither the first level of cache nor the VC. The approach is simple and effective and, in average, the miss ratio reduction is 14.4% using VC.

SB and VC are the mechanisms with cache structure fixed. The cache structure changes in ALS and AFL: the line size changes dynamically. In ALS, every memory reference can have a customized line size associated with it. In case of a hit, there is no difference between an ALS cache and a standard cache. In case of a miss, the adaptation is activated. The goal is to adapt the line size: spatial locality can be exploited using a larger line size; the interferences can be reduced using a smaller cache line size. Therefore at any time there can be different active line sizes. ALS tends to reduce capacity misses3 and conflict misses. The approach is flexible and is effective: in average the miss ratio reduction is 32.5% using ALS.

The AFL approach is a "simplified" version of the ALS. The cache line size dynamically changes but at any time there is only one cache line size. During an interval, the cache performance is monitored. And at the end of the interval the best cache line size is properly determined and set for the next interval. The approach is simpler than ALS and is in general very effective; using AFL, the miss ratio was reduced by 36.6% in average.

The benchmarks we chose are *Data Intensive Systems (DIS) Benchmark* [37][36], *Stressmark* [38] and *SPEC95* [49]. DIS Benchmark suite is a collection of five applications. They are most scientific computations, i.e. Fast Fourier Transform (FFT) [19]

and Method of Moments (MOM) [36], and one of them is a data-base application. They have very demanding memory space requirements, but they are optimized for cache-based memory hierarchy. Stressmark suite is a set of kernels, taken from larger applications such as DIS benchmarks. The kernels are chosen to exploit memory access behaviors that are not easily measured otherwise. SPEC95 is a well-known benchmark suite; it is a collection of several heterogeneous applications, briefly classified as floating point and integer applications. The benchmark suite is designed to test the different aspects of modern architectures, but the major objective is not to test memory hierarchy. Indeed, modern architectures can obtain good performance on this benchmark suite.

The experimental results collected are simulation-based. The simulator used is *simple-scalar* [8] enhanced with the adaptive memory modules developed by our group (used also in [51][52]). We did not use sampling-based-simulation [33] [12]. We simulated up to 3 billion instructions, which is a good compromise between the precision of a full simulation and the speed of a partial one.

The rest of the report is organized as follows. In Section 2 we propose and analyze a model of cache misses. MoM is used to show the memory behavior over time. The recognition of this behavior helps thethe evaluation of adaptations. In Section 3 we give a detailed background and description of the adaptation approaches. In Section 4 we present the complete set of the benchmark suites. In Section 5 we introduce the experimental environments and show the experimental results. Finally, we gave the conclusion of this report. The Appendix includes the rest of environmental configurations, the complete input sets, and every detail needed to reproduce the results here presented.

2 A Model of Cache Misses

In this section, we look for a method to determine when and why an adaptation is effective. For such a purpose we need a quantitative and concise measure of memory behavior. We observe that the memory adaptations target to reduce the effects of or remove different types of misses. So we measure the miss ratios of conflict, capacity and compulsory miss. By construction, miss ratios do not express any relationship among each other; thus we cannot show when an adaptation is effective, even if the misses of the corresponding type exist. However, we show that adaptations are not effective when there is no certain type of misses, for example Victim Cache for applications with a few number of conflict misses. We catch the effect of the relationship among misses by the measure of the miss distribution over time. We present an example (MoM) where the type of miss are distributed in such a way that in a certain period of time adaptation is very effective, but the average miss ratio does not show any effect. Thus we claim that

the average miss ratio is not always persuasive.

2.1 Miss Ratio

The overall miss ratio is the first metric we observe. Considering the execution time and the size of the benchmarks, we use a fast simulation process to determine the miss ratios. We use *Shade* software package from Sun microsystem [9]. We compile the benchmarks for Ultra 5 architecture and use the predefined cache simulator *cachesim5*. The experimental results are reported in Figure 1.

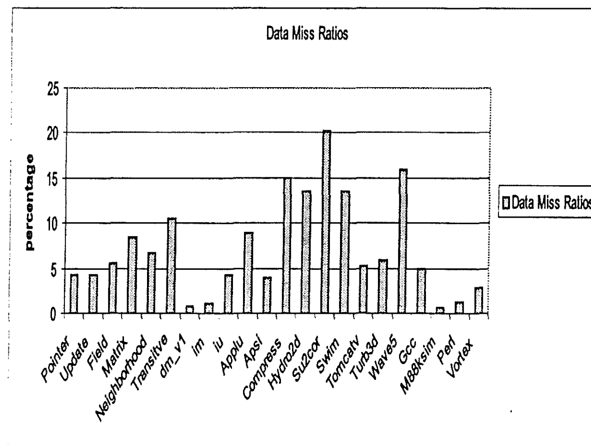


Figure 1: Average Miss Ratios

The miss ratio is defined as the ratio of the number of misses at a certain level of the memory hierarchy, e.g., cache, over the total number of memory references (both loads and stores). This metric is commonly used, and in general is very useful. Reduction of the miss ratio means performance boosting. This can be reflected in the following simple model. This model describes the access time of the memory system with one level of cache:

$$\begin{aligned} t_{MemoryTime} &= |Hits|t_{hitLatency} + |Misses|t_{missLatency} \\ &= |TotalAccess|t_{hitLatency} + MissRatio(t_{missLatency} - t_{hitLatency}) \end{aligned}$$

where $|Hits|=$ number of hits, $|Misses|=$ number of misses, and $|TotalAccess|=$ number of memory accesses.

From this model, we can see that there are several ways to improve application performance. New VLSI technology and organization of memory hierarchy can reduce the

access time. Optimization of applications can reduce the total number of accesses or the miss ratio alone (i.e. tiling in [48]). For example, the application is re-designed to reduce the number of accesses or an architecture-aware-compiler re-works the code to fit the memory system. Finally but most importantly, the miss ratio is the function of a pair of parameters (application, memory-system), and orthogonal improvements can be achieved by adapting the memory system to the application.

However, this simplified model does not express the composition of different types of misses. Since adaptations intend to deal with different types of misses, the model is not powerful enough to show when and why one adaptation is effective. The quantitative measure of the type misses can help to determine what kinds of adaptations are effective. In this section we investigate the quantitative discrimination of misses by type: *Capacity, Compulsory and Conflict misses.*

2.2 Miss Discrimination by Type

Given a memory system as baseline, we determine the types of misses based on these simple observations:

- Observation 1: a compulsory miss cannot be removed (The accesses to the lower memory hierarchy cannot be avoided because the data are not in the cache. Prefetch can only hide miss latency.)
- Observation 2: an ideal cache, fully associative and with optimal replacement policy, removes conflict misses. An optimal replacement policy uses the information from the past accesses as well as the future accesses (non causal) to replace data from the cache.
- Observation 3: only the increase of the cache size removes a capacity miss (a capacity miss is similar to a conflict miss, since the data is evicted from the cache and replaced by another data, but it is due to lack of space).

We determined the types of misses by indirect measure, tuning the memory system parameters for different simulations and measurements. We describe our methodology as follows:

- We simulated and got the number of data misses for the baseline (32KB, 2-way, 32B line), and we indicated it as $M_{BaseLine}$;
- We acquired the number of data misses for the same in size cache but 32-way associative, defined it as $M_{capacity,compulsory}$; we chose a 32-way associative cache because we think it a good approximation of an ideal cache and no conflict misses would be present; only capacity and compulsory misses exist.

- We determined the number of misses when the cache size is 256MB, and defined it as $M_{compulsory}$; we considered that the cache of 256MB is big enough so that there are only compulsory misses, Observation 3 and 1.

The number of conflict misses is computed as $M_{conflict} = M_{BaseLine}$ - $M_{capacity,compulsory}$ and the number of capacity misses is computed as $M_{capacity} = M_{capacity,compulsory}$- $M_{compulsory}$. In Figure 2 we show, in percentage, the number of misses by nature.

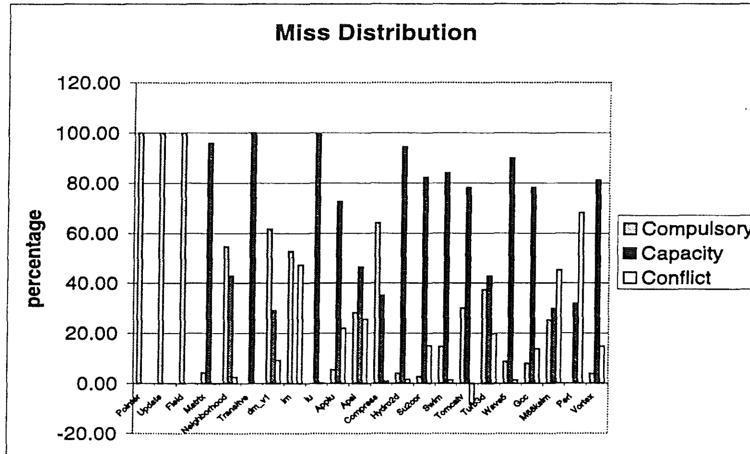


Figure 2: Miss Discrimination

In benchmark *tomcatv*, $M_{capacity,compulsory}$ is larger than $M_{BaseLine}$, which means that increasing the associativity of the cache would increase the number of misses. That is why we computed a negative number for the conflict misses (which is not possible in the real life). The problem is that the measures are just approximations, since an ideal cache is non-causal and cannot be simulated by the software utilized. The replacement policy adopted is the *Least Recently Used (LRU)* [15], and it might not be the optimal policy for an application since it uses just the past behavior. To explain how the replacement policy affects the miss measurement, consider two caches of equal size but with different associativity: k1 and k2, with k1< k2. If the replacement policy is optimal, the cache with associativity k1 must have a larger number of misses than the cache with associativity k2. However, this might not be the case because of non-optimal replacement policy. Since we cannot exactly determine the $M_{capacity,compulsory}$, the percentage values for the conflict misses are just lower bounds, and those for capacity misses are just upper bounds.

This chart quickly points out that different benchmark suites have very different memory behaviors. DIS Stressmarks have no conflicts, mostly capacity and compulsory misses. SPEC95 has mostly capacity misses, except two of them with mostly conflict misses.

2.3 Application of Miss Discrimination

Adaptations selectively reduce or hide certain types of misses. Therefore when there are no targeted misses, the adaptation cannot be effective. We claim that we can use the collected average measure to show where there is no improvement space, and thus when an adaptation is not effective. For example, from the miss distribution collected, the DIS stressmark suite has no improvement space for VC, because it has no conflict misses. In general, the DIS benchmark suites have a small number of conflict misses; we can predict that any memory adaptation that is good at reducing conflicts should not be very effective. Another example is from benchmark *Perl*: it has very small improvement space for SB, because the capacity and compulsory misses are not dominant.

To justify our claim we report a qualitative comparison, obtained from the experimental result in Section 5, among memory adaptations. We have summarized in Table 1 the memory adaptations that are effective for each benchmark. The ranking goes from the best, 1, to the worst, 4. In the last row we report the median as representative of the average behavior. When there is no indication, no memory adaptations achieve any significant improvements. As we can see, the experimental results confirm our expectations. For example VC is not effective for the Stressmarks.

However, the approach to discriminate the types of the misses cannot be used to explain why adaptations are effective. The reason is that any percentage is an average measure over the whole execution of the application. The temporal distribution may vary; conflicts misses may arise all in a very short interval of time or they may be evenly distributed during the execution. next, we present an example where miss distribution over time is important in the evaluation of the miss reduction by adaptation.

2.4 Distribution of Misses over Time in MoM

In this section, we show that memory adaptations may be effective in a short period of time, but the performance is not detectable globally. Using the following example, we show that, in general, miss ratio is not an effective metric to evaluate performance.

The *Method of Moments* is one of the DIS benchmarks. The application has a very small miss ratio percentage even for the very large inputs. To illustrate properly the following experimental results, we need to explain briefly the algorithm. MoM is a divide and conquer algorithm. It has a tree-like decomposition, and an integer number identifies each level in the tree. At level 1 there are the leaves of the tree. The root of the tree is at the top and its level is the function of the input size (ranging from 5 to 7). The algorithm first visits the tree from the leaves to the root (upwards) and then from the

root to the leaves (downwards). Each node is visited twice but the computation is different each time. We can indicate the execution of the application as a sequence of the levels visited, i.e. a 5 levels tree has the execution following the order 1, 2, 3, 4, 5, 4, 3, 2, 1. The leaves have a head and a tail computation. During upward stage the head computation at level 1 is the preparation of the inputs, and the tail computation during downward stage is the formatting of the outputs. We instruct the code so that we can collect statistics by hardware counters on a *R12K* microprocessor [60]. In Figure 3 and Figure 4, we can observe the temporal behavior, level by level, of the cache misses and efficiency to issue instructions (average cycle per instructions: CPI). The input set is Plate 5.

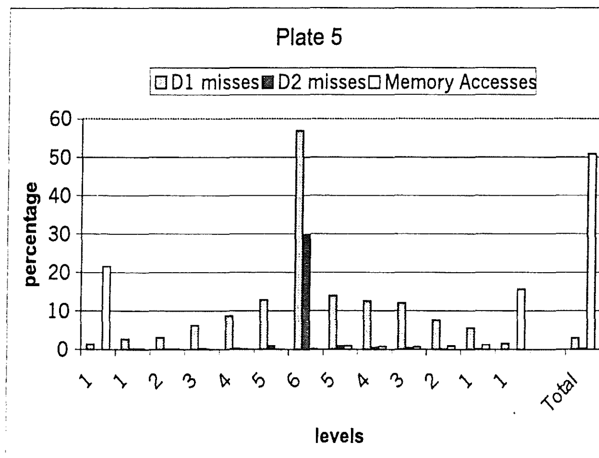
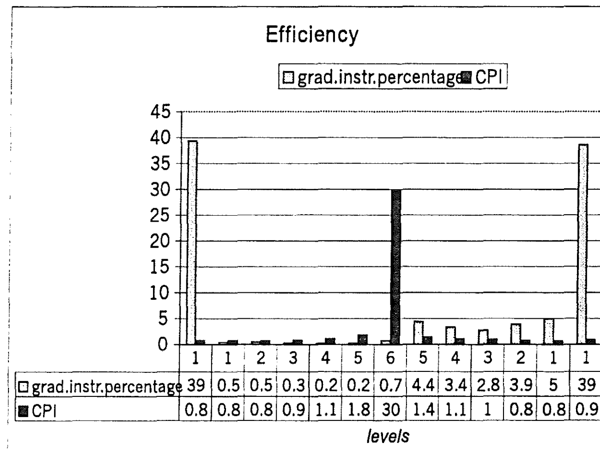


Figure 3: MoM: Plate 5 Miss Distribution over Different Levels & Average Miss Ratio



	1	1	2	3	4	5	6	5	4	3	2	1	1
grad.instr.percentage	39	0.5	0.5	0.3	0.2	0.2	0.7	4.4	3.4	2.8	3.9	5	39
CPI	0.8	0.8	0.8	0.9	1.1	1.8	30	1.4	1.1	1	0.8	0.8	0.9

Figure 4: MoM:Plate 5 Instruction Issue Efficiency

The Figure 3 shows that 50% of instructions of the application are loads and stores and the miss ratio is 4% (last two bars). The miss ratio is level dependent. The higher

is the level the larger is the miss ratio; i.e. at level 6, the root, miss ratio is 57%. In Figure 4, we can see the distribution of the graduated instructions and their average CPI. The average miss ratio is small because memory accesses are mostly at level 1 with high data locality, while the higher levels have very few accesses with poor data locality. In the latter case, the miss rate degradation is associated with CPI degradation, therefore performance degradation.

The input size of MoM for Plate 5 is too large for any simulations. So a smaller input set must be used. We show in Figure 5 that a similar behavior exists for Plate 3, which has smaller input size so that we can obtain experimental result by simulations.

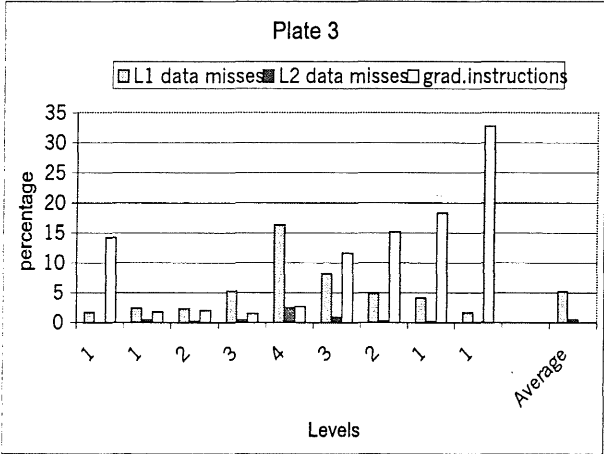


Figure 5: MoM: Plate 3 Memory Distribution over Different Levels & Average Miss Ratio

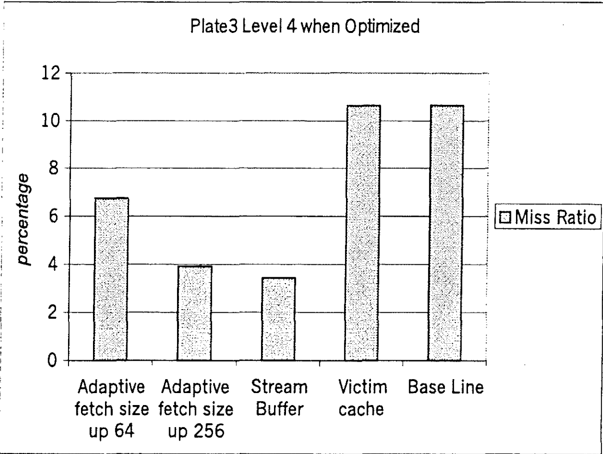


Figure 6: MoM: Plate 3 Miss Ratios using Different Adaptations

We focus on the computation at the top level, in this case the level 4. At this level most adaptations are very effective. The simulations results are in Figure 6.

VC does not improve any performance but the other approaches are effective and achieve improvements between 40% and 60%. The SB is the most effective one.

2.5 Analysis Limitations

In this section we do not propose any closed formula or analytical model describing the cache misses for the following reasons:

- Applications can be really complex, and only the developers may have the chance to determine a simple but effective model (see MoM).
- Small kernels may have extra memory accesses that are introduced artificially by the compiler (such as Pointer) and cannot be taken into account by a static model.
- To obtain a model of cache misses is really challenging sometimes even for a single application and single memory system: one must exploit the relationships among application, compiler optimizations, static instruction scheduling and dynamic instruction scheduling. This is true even though most of the modern architectures preserve the in-order memory accesses proposed by the program, for memory hierarchy coherence.

We can see that to obtain a cache model is becoming impractical for a large set of applications.

3 Background on Adaptations

Four memory adaptation approaches will be described in this section: Stream Buffer (SB), Victim Cache (VC), Adaptive Line Size Cache (ALS) and Adaptive Fetch Size Cache (AFL).

3.1 Stream Buffer (SB)

Stream Buffer was first proposed by Jouppi [28]. The author proposed a very basic prefetch approach, *pre-fetch on a miss*, and later was improved in [39]. In order to give a clear view of the SB we implemented, we need to describe pre-fetch mechanisms thoroughly. Pre-fetch mechanisms can be classified into Software pre-fetch and Hardware pre-fetch. In hardware pre-fetch, we investigate two types of prediction mechanisms: PC Based and Partition Based. In Section 3.1.3 we describe the organization and the prediction mechanisms of SB used to collect experimental results of this paper.

3.1.1 Pre-fetch

Pre-fetch [54] is the mechanism that fetches data from the lower-level caches (or memory) before they are actually used. The advantage of Pre-fetch over the *fetch-on-demand* policy is that it can hide the latency of compulsory and capacity misses. A compulsory miss happens when a cache block is accessed for the first time. A capacity miss happens when a accessed cache block is not in the cache because of insufficient cache space. The code in Figure 7 is an example where compulsory and capacity misses happen.

```
for ( i = 0 ; i < k ; i ++) {
    A [ 0 ] += B [ i ];
    for ( j = 1; j < N ; j ++)
        A [ j ] += A [ j - 1 ];
    A [ 0 ] += A [ N - 1 ];
}
```

Figure 7: Example of When Compulsory and Capacity Misses Happen

In Figure 7, the linear array A has size N larger than the size of the cache C. A is updated k times in the nested loop in a circular fashion. In the first iteration, I=0, the linear array is read for the first time. Compulsory misses happen. Since only part of the array can fit the cache, the number of misses does not change in the successive iterations ($I > 0$). The following example in Figure 8 proposes a software Pre-fetch solution to reduce compulsory and capacity misses (C is the size of the cache):

```
( HW / SW ) Pre-fetch A [ 0 : C - 1]
....
for ( i = 0 ; i < k; i ++) {
    A [ 0 ] += B [ i ] ;
    for ( j = 1 ; j < N ; j ++ ) {
        A [ j ] += A [ j - 1 ] ;
        ( HW / SW ) pre-fetch A [ | j + C | mod N ]
    }
    A [ 0 ] += A [ N ] ;
}
```

Figure 8: Prefetching Optimization on the Previous Example

In Figure 8, in order to remove the latency penalty due to compulsory and capacity misses, pre-fetching is performed in parallel with processor computations. There are two pre-fetch instructions in the example. One is outside the loop and one is inside the loop. The outside one pre-fetches the first C elements of A. The inside one pre-fetches one element a time, which will be used C iterations later. Instead of waiting for data requested and load command issued, pre-fetch mechanism anticipates that a certain (set of) data may be referenced in the near future. If this (set of) data is not in the cache, a fetch

command will be issued, either by hardware or software, to fetch the corresponding data block. By the time when the data is needed, it is already in the cache and ready to use.

As we mentioned earlier, pre-fetch commands can be issued either by hardware or by software. We do not describe Software pre-fetch [4], [34] in this report, because our focus is on Hardware pre-fetch, which is described in the following subsections.

3.1.2 Hardware pre-fetch

Hardware pre-fetch depends on special hardware to track data reference traces, to recognize the references with constant strides, and to fetch in advance an instance of a reference based on the stride detected. The constant stride can be *unit stride* and*non-unit stride.*

Unit-stride pre-fetch is also called *sequential pre-fetch.* The simplest approaches of sequential pre-fetch are based on *one block look ahead* (OBL) [47]. OBL initiates a pre-fetch for block b+1 after block b is accessed.

Depending on when to initiate the pre-fetch of b+1, the implementations of OBL are:

- *Pre-fetch-on-miss*: it initiates a pre-fetch for block b+1 whenever an access to block b is a miss and block b+1 is not in cache.
- *Tagged pre-fetch*: it associates a tag bit with every cache block. This bit is set to zero when a block is pre-fetch. Later, if this block is accessed, the bit is set to one. The zero to one transition trigger the prefetching of the next sequential block b+1.

Generally, pre-fetch-on-miss is less effective than tagged pre-fetch. For example, in a purely sequential stream, pre-fetch-on-miss will result in a miss every other access, while tagged pre-fetch will not.

One shortcoming associated with sequential pre-fetch is that pre-fetch may not be initiated far enough ahead to hide the latency. If the pre-fetching is not completed by the time when block b+1 is needed, the processor would stall. An approach to solve this problem is to pre-fetch up to K successive blocks, or a block that is k references ahead, where K can be statically or dynamically [13] determined. But this might result in high traffic from and to the lower-level cache (or memory) [42].

Non-unit-stride pre-fetch detects the non-unit stride Δ and pre-fetch blocks at Δ units stride away. Notice that if Δ equals to one, this approach would be sequential pre-

fetch. Based on the special logic used to monitor access patterns, there are two types of non-units stride pre-fetches:

- PC based Pre-fetch
- Partition Based Pre-fetch

PC based pre-fetch [18][11] is an approach that predicts the stride by comparing the addresses used by successive load or store instructions. For example, three addresses a1, a2, and a3 are used by a load instruction in three successive iterations. If a2-a1 = a3-a2, a stride Δ =a3-a2 is established and the data at address a4 = a3+ Δ is pre-fetched. If a2-a1 $\neq$ a3-a2, we remove a1, and the same process is repeated using a2, a3, and a4. A table (called reference prediction table: RPT in [54]) is necessary to store the most recently used addresses and the last recently detected stride for a memory instruction. Ideally, each memory instruction should be assigned an entry. In practice, however, the table keeps only the most recently executed memory instructions. Table entries are indexed by PCs. The Finite State Machine (FSM) in Figure 9 is a formal description of how PC based pre-fetch works:

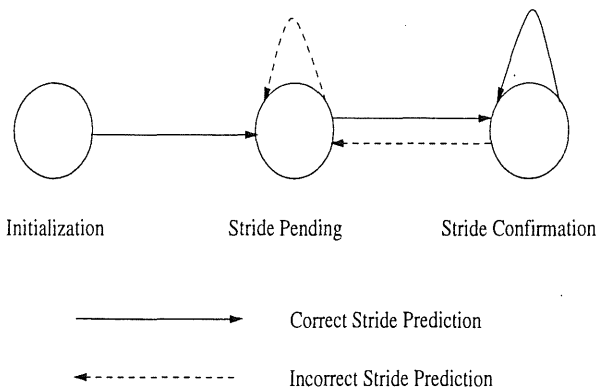


Figure 9: Finite State Machine of PC Based Prefetch

- *Initial state*: An entry is allocated, but no stride is established. The stride is initialized as NULL. The entry contains the operand address a1 of the most recently executed memory instruction. Next time when the successive address a2 is referred, a stride Δ= a2-a1 is detected, and the state enters into *Stride Pending State*. A pre-fetch is issued with the address a2+Δ if the corresponding data block is not in the cache.

- *Stride Pending state*: a stride is newly detected, and it needs to be confirmed by the next reference address a3. When a3 comes, we get Δ_{new} = a3−a2. If Δ_{new}= Δ, the

stride is confirmed. The state enters into *Stride Confirmed State.* However, if Δ_{new} $\neq\Delta$, the entry is updated by the new address a3, along with the newly detected stride Δ_{new}. The state would stay in *Stride Pending State* .

- it Stride Confirmed state: A stride Δ_{new} is established. A pre-fetch for a set of data blocks with distance Δ_{new} is issued. When new address a4 comes and the new stride a4−a3=a3−a2, the state remains at this state. Otherwise, the state goes back to *Stride Pending State* .

Partition based pre-fetch can be implemented as consecutive-address based scheme [39], which is used in in our experiments. Consecutive-address based scheme is similar to PC based pre-fetch [18][11] described in the previous section. It is stride-based. One table is used to track the references. Each entry in the table is allocated to a memory "chunk", which is a contiguous memory area. The higher bits of the reference addresses index the entry of the table. If two memory addresses belong to the same "big chunk", they index the same entry. In practice, the division of the "chunk" can be done statically. Usually, the elements in two different vectors are allocated to different "chunks". For example, in Figure 10, the memory addresses for A[I] and A[I+1] belong to the same chunk and index the same entry. Similarly, B[I] and B[I+1] index the same entry. The strides for prediction are calculated between two successive memory references. If a stride Δ is detected to be constant, the data that lies Δ apart is pre-fetched. The mechanism for detecting strides is the same as PC based pre-fetch, so FSM in Figure 9 can be employed here. For example, in Figure 10, the strides of 1 are detected for vector A and B, so A[I+2] is pre-fetched after A[I+1], and B[I+2] is pre-fetched after B[I+1].

```
for ( i = 0 ; i < k ; i += 3) {
    A [ i ] = B [ i ] + 1;
    A [ i + 1] = B [ i + 1 ] + 2 ;
    A [ i + 2 ] = B [ i + 2 ] + 3 ;
}
```

Figure 10: Example where Partition Based prefetch is effective

PC based pre-fetch can capture the locality in one reference instruction of different iterations. Partition based pre-fetch can catch the locality in a group of references. From the mechanism point of view, we cannot say that one is better than the other. The example Figure 11 is a case where PC based pre-fetch is better. In this example, PC based pre-fetch can detect the strides for the four different memory references, V1[I], V1[2I], V1[4I], and V2[I]. Thus it issues pre-fetch commands with appropriate stride for individual reference. Partition based pre-fetch, however, cannot distinguish the first three

memory references V1[I], V1[2I] and V1[4I]. They fall into the same table entry and no constant strides can be detected. For V2[I], partition based pre-fetch will distinguish it from the reference to V1, and pre-fetch for V2 will succeed.

In another example in Figure 12, Partition based pre-fetch outperforms PC based pre-fetch. In this example, the index computations in vector V1 are randomized. PC based pre-fetch cannot detected any strides. However, Partition Based SB can detect the stride 32 between V1[I*RANDQ], V1[I*RANDQ + 32], and V1[I*RANDQ+64], thus pre-fetching is issued for V1[I*RANDQ+96].

In our experiments, both PC based pre-fetch and Partition Based prefetch are employed. The experimental results show that there are cases where one mechanism outperforms the other.

```
for ( i = 0 ; i < N ; i ++ ) {
    ip += V1 [ i ];
    ip += V1 [ 2 * i ];
    ip += V1 [ 4 * i ];
    ip += V2 [ i ];
}
```

Figure 11: Example where PC based prefetch is more effective than Partition based prefetch

```
for ( i = 0 ; i < N ; i ++ ) {
    randq = V1 [ i ];
    ip1 += V1 [ i * randq ];
    ip2 += V1 [ i * randq + 32 ];
    V1 [ i * randq + 64 ] = ip1;
    V1 [ i * randq + 96 ] = ip2 ;
}
```

Figure 12: Example where Partition based prefetch is more effective than PC based prefetch

3.1.3 Stream Buffer

Stream Buffer [28] has a small FIFO buffer and a hardware pre-fetch mechanism. The pre-fetched data are fetched into Stream Buffer to avoid cache pollution. On a cache miss, the data are sent to cache. At the same time, they are removed from Stream Buffer. The prediction mechanism used in Stream Buffer can be either PC or Partition based.

Therefore, it can pre-fetch data in unit stride or non-unit stride. The experimental results from both prediction mechanisms will be presented in this report.

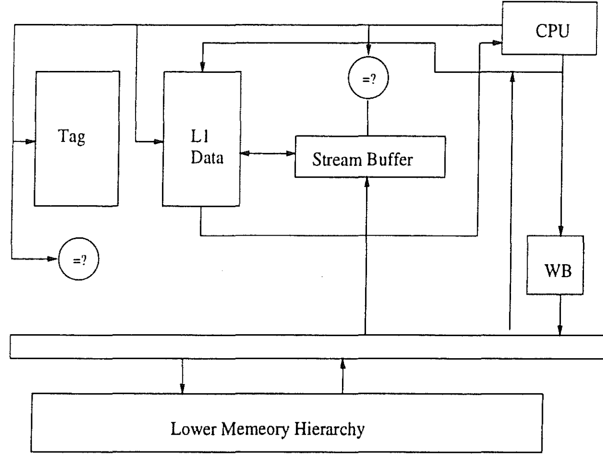


Figure 13: Memory Hierarchy with Stream Buffer

Figure 13 illustrates the memory hierarchy with the Stream Buffer between the Level one and lower level cache (or memory). The data blocks are pre-fetched into Stream Buffer in order. They will be evicted in the same order. More specific, once a pre-fetch is started, up to K cache blocks with stride Δ are loaded, where K is the degree or depth of Stream Buffer, and Δ is the established stride. Each time data are required, Stream Buffer is accessed in parallel with data cache. Only the tag of the first line in Stream Buffer is compared with the tag of the reference address. If there is a miss in cache and a hit in Stream Buffer, Stream buffer provides the data to the cache. The first line is evicted and the following lines shifts up. In case that data are not in the first line, Stream Buffer is flushed.

One of the key factors in Stream Buffer is the depth of Stream Buffer K. The larger K in a regular stream, the more future references will be pre-fetched ahead. However, large K increases traffic. In our experiments, we choose K as 4.

A single-way Stream Buffer is not effective when there are multi-way streams, as in example in Figure 14. Every stream flushes the pre-fetched data for the previous one.

To solve this problem, multi-way Stream Buffer is propose in [28]. It is used to remove compulsory and capacity misses in multiple concurrent streams. Obviously, the larger the number of Stream Buffers, the more concurrent streams can be captured. The optimal number of parallel Stream Buffers is application-dependent. In our experiments, we use a 4-way Stream Buffer. In general it is a good trade off between space and performance.

```
for ( k = 0 ; k < n ; k ++ )
   for ( i = 0 ; i < n ; i ++ )
      for ( j = 0 ; j < n ; j ++ ) {
         old = DIN ( j , i );
         new1 = DIN ( j , k ) + DIN ( k , i );
         DOUT ( j , i ) = ( new1 < old ? new1 : old );
}
```

Figure 14: Example of when Single Way Stream Buffer fails and Multi-ways Stream Buffer is effective

3.2 Victim Cache (VC)

In a memory hierarchy composed of several levels of caches, the Victim Cache [50][46] is placed between two levels as assistant for the upper level. VC is a very small fully associative cache. It has the same line size of the level it assists. It applies a write back policy. It holds the data evicted from the upper level, and only when evicted from it, the data are sent to the lower levels. In a memory sustem, if some data is repeatedly evicted from the upper level, and then requested after a short period of time, a large percentage of application execution time would be spent waiting for the data from the lower levels. However, when VC is used and the data is found in the VC, the waiting time is shortened a lot. For its small size and associativity, VC intends to hold data with temporal locality. These data are evicted from upper level cache due to conflict misses.

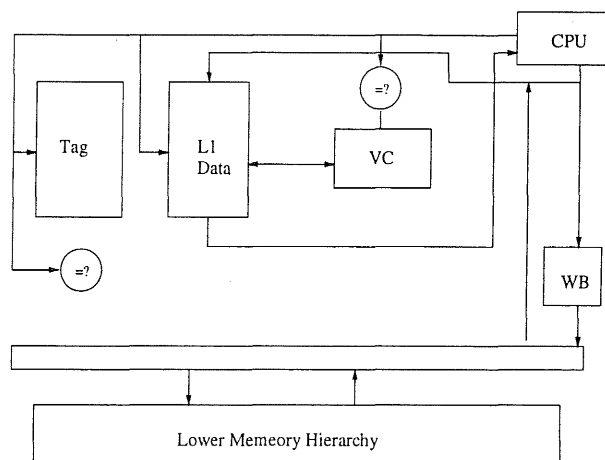


Figure 15: Memory Hierarchy with Victim Cache

Figure 15 illustrates the memory hierarchy with VC between the Level one and lower

level cache (or memory).Each time a data is requested, both VC and the assisted level are accessed in parallel. The lookup in VC is the same as lookup in general data cache. When data are missing in the upper level but can be found in the VC, the VC supplies the data as a usual cache. There is no need to access the other level cache (or memory). If the data are missing in VC, the lower level cache (or memory) has to be accessed. The fetched data will bypass VC and go directly to the upper level cache. The access time of the VC is not longer than access time of the assisted level, because the small size of the cache assistant compensates the complexity of the associativity.

Depending on the reference stream, the performance of VC can be either significant or negligible. In a sequential reference stream where there is seldom data reuse, VC can remove few misses. While if two vectors are accessed concurrently using a direct-mapped cache, $(n-1)$ out of n miss will be removed for each line, where n is the number of elements per line. The performance of VC depends also on its capacity. The larger the cache is, the more temporal locality can be captured, and the more the conflict misses will be removed. However, the lookup time would be longer.

Right now, we are using VC of 32-way associative with 32B of each line. Its performance will be described in Section 5.

3.3 Adaptive Line Size Cache (ALS)

In [21] [43], the authors present experiments results showing that different applications have different spatial and temporal locality, and also different parts of an application may have similar characteristics [55]. In this scenario, applications may be unable to take full advantage of the spatial locality offered by a cache with fixed line size. ALS [51] is a cache design that uses cache lines of different sizes concurrently. The size of each line changes dynamically on demand of application needs.

There are some definitions in the memory hierarchy with ALS cache:

- **Definition.** ALS cache is composed of same-sized lines, *physical cache line (PCL).* Each of them is a power of two, for hereafter is 16B.
- Definition. The composition of contiguous PCLs is a virtual cache line (VCL), whose size is a power of two.
- **Definition.** Every VCL is associated with a *virtual line (VL)* in the lower level cache. On a cache miss, one VL is fetched into ALS and fill one VCL.
- **Definition.** Two VLs are said to be *neighboring* if they have the same size and the starting addresses of both of them divided by twice the VL size are the same [51].

For a certain VCL or VL, its line size can be changed dynamically during its lifetime. Line size adjustment is based on the algorithm we will describe shortly. No matter how it changes, line size is always a power of two. The lookup in ALS is the same as in a fixed line size cache. A cache hit will not result in any line size change. In case of a cache miss, the missing VL is fetched from lower-level cache (or memory) to a buffer close to the ALS cache. For every VCL to be evicted, the prediction algorithm determines the size of its associated virtual line, and makes it increase, decrease, or stay the same, depending on the locality detected. Then the VCL is evicted from the cache.

When a VCL is replaced from cache, the line size prediction algorithm works as follows:

1. Get the VL that is mapped to the VCL and the neighboring virtual line of the VL.
2. If the VCL of the neighboring virtual line is also in the cache, the line size is increased.
3. In case that there is no neighboring VCL in the cache and at most one half of the VCL has been accessed before, the line size is decreased.
4. Otherwise, no change is made to the line size.

ALS exploits spatial locality by increasing the line size. In cases where spatial locality is small but temporal locality is demonstrated, the line size is decreased to avoid cache pollution. Ideally, line size can be arbitrary large or small so that either spatial or temporal locality can be exploited as much as possible. However, the minimum line size is limited by the physical line size, and the upper bound of the line size is limited by the fixed bandwidth between ALS and lower cache (or memory). Another issue associated with ALS is the initial line size. Experiments have shown that the selection of initial line size is not important [55]. In our experiments, the candidate line sizes are 16B, 32B, 64B, 128B, 256B, and the initial line size is 32B.

In ALS, the line size is predicted for a specific virtual line, while in Adaptive fetch line cache (AFL), which will be described in the following section, line size is predicted for all the lines in the cache.

3.4 Adaptive Fetch Line Size Cache (AFL)

The AFL [52] has the same motivation as ALS: to allow cache to follow the change of locality, both across applications and in an application. As in ALS, the fetch line size must be a power of two. The difference is that, at any time, the fetch line size is one

for all lines, and its prediction is based on the global information, thus is not biased to a particular virtual line.

Depending on observations used for prediction, there are two types of fetch size prediction algorithms:

- *Sampling-based* fetch size prediction: it predicts the optimal fetch size for a long interval by finding the optimal fetch size over several small intervals. The optimal fetch size is the one that will result in minimum miss ratio among a set of candidate fetch sizes. Each candidate is applied for a short interval and its miss ratio is calculated. After all the candidates are tested, the corresponding miss ratios are compared. The candidate with the minimum miss ratio is selected, and is going to be used throughout the next interval. In the next interval, the fetch size for the coming interval will be predicted in the same way. It has been observed that the optimal fetch line size for a small interval is very likely to be optimal for a long interval [52].

- *Locality-based* fetch size prediction: As in the sampling-based fetch size prediction, memory access trace is divided into separate time intervals. The fetch size for the coming interval is predicted based on the spatial locality observed in the current interval. The spatial locality information, which is reflected in the tedency of the size adjustment trace of individual lines, is accumulated and used as the basis for fetch size prediction. Two counters, $IncCounter$ and $DecCounter$, are used to accumulate the number of times that fetch line size is requested to increase and decrease by individual lines, respectively. $IncCounter\%$ and $DecCounter\%$ are used to characterize locality. At the end of each interval, $IncCounter\%$ and $DecCounter\%$ are compared with the thresholds $threshold_{inc}$, and $threshold_{dec}$, respectively. If $IncCounter\%$ is bigger than $threshold_{inc}$, fetch size doubles. If $DecCounter\%$ is bigger than $threshold_{dec}$, fetch size decreases by half. In the next interval, the same process is repeated and the fetch size is predicted.

Experiments [52] have been done to explore the parameters in the above two fetch size prediction algorithms. The fist parameter is the sampling interval in Sampling-based AFL. It has been pointed out that smaller intervals are better because AFL can adapt to the changing locality faster. However, if the intervals are too small, the locality information used for prediction will be insufficient. In our experiments, the sampling interval is set to be 100K instructions.

The possible fetch sizes that can be used in the next time interval are explored for Sampling based AFL. In this algorithm, the fetch size can be:

- All possible fetch sizes, and
- The set of current fetch size, the immediately larger fetch size and the immediately smaller fetch size.

The first approach takes one interval time to determine the optimal fetch size because all possible fetch sizes are tested, but it results in more computations. The second approach may take more than one interval to determine the optimal fetch size, but results in fewer computations.

In Locality-based AFL, thresholds that will result in the smallest miss rates are different for different applications [52]. High $threshold_{dec}$ and low $threshold_{dec}$ will cause a small fetch size, while low $threshold_{dec}$ and high $threshold_{inc}$ will cause a large fetch size. If both thresholds are too high, the adaptation is insensitive to the change of locality. The adaptive threshold algorithm, called aging threshold algorithm, is thus implemented in [52]. In this algorithm, both $threshold_{inc}$, and $threshold_{dec}$ are decreased by an aging ratio if there is no fetch size change prediction.

In our experiment, we chose the locality-based prediction. The possible fetch sizes that can be used in the next time interval are: current fetch size, the immediately larger, and the immediately smaller. The possible line sizes are 16B, 32B, 64B, 128B, and 256B. The *aging threshold algorithm* was applied, with aging ratio 0.01, lower bound of threshold 0.4, middle threshold 0.55, and upper bound is 0.7 (Please refer to [52] for the more detailed explanation of these parameters).

4 Benchmark Suites

In order to evaluate memory adaptation approaches under all kinds of situations, a set of representative and commonly used benchmark suites were chosen. From the most general ones to the most specific ones, they are *Data Intensive Systems (DIS) Benchmark* [37][36], *DIS Stressmark* [38] and *SPEC95* [49].

- The term Data Intensive is used to reference problems characterized by large data sets, non-contiguous memory accesses, and frequent load/store instructions. DIS Benchmark suite was created to qualify the performance gains likely to be achieved for data intensive problems. DIS benchmarks include the processes of data movement and preparation, and the interactions between program components, as in general applications.

- DIS Stressmark suite was developed as complementary of DIS Benchmark suite. It intends to illustrate more directly particular elements of the DIS problems. It requires less energy to implement but often at the expense of reduced realism. The focus is not the on number of accesses but the way memory is accessed. Thus, DIS stressmark suite evaluates the performance of memory hierarchies; and in some cases (i.e. Pointer) optimized memory architectures (i.e. SB) do not outperform general-purpose memory architectures (with simple cache).

- SPEC95 is intended to provide a common set of programs to measure computing-intensive performance of processor, memory hierarchy and other features of a computer system. This common set is used to compare performance of different architectures. It was built to be more resistant to compiler optimizations, with longer run times and larger problems, and having more application diversity.

The complete view of each of these benchmark suites will be given in the following sections.

4.1 DIS Benchmark Suite

DIS benchmark suite [37][36] was developed as representatives of Data-Intensive (DIS) applications so that new architectures and approaches explored for these applications can be effectively evaluated. Usually, these applications have large data sets that are accessed non-contiguously. They cannot take full advantage of typical memory optimizations.

DIS Benchmark suite intends to represent DIS applications in a simplified but realistic way. Instead of focus on specific, isolated tasks, DIS Benchmark suite includes the processes of data movement and preparation, as well as the interactions between program components. People should not assume that, as data sets grow large, these "overhead" functions diminish in proportional resource consumption.

There are five benchmarks in DIS Benchmark suite:

- *Method of Moment* (MoM). MoM algorithm is applied in the frequency domain to compute electromagnetic scattering from complex objects. It requires the solution of large dense linear systems of equations. The currently used solver is Boeing's fast solver, based on the preconditioned GMRES iteration method and fast multi-pole method (FMM) for fast matrix-vector multiplies. The key FMM kernels represented in the benchmark are the translation operations and spherical harmonic filtering.

Indeed, the benchmark is missing of the pre-processor phase, the iterative solver and the post-processor phase typical for MoM. The computational complexity of these FMM methods is O(NlogN) and memory requirement is O(N). There is one memory-related bottlenecks that contribute to MoM's advantage as DIS problems: non-unit stride accesses. The filter of spherical harmonic filtering in FMM is on rectangular arrays of data in three stages. The arrays are accessed first by rows, then by columns, and finally, by rows again. In the second stage, it is necessary to access memory locations that are not consecutive. So the speed of the fast MoM algorithm is limited by the speed of accessing memory hierarchy with non-unit stride.

- *Simulated SAR Ray Tracing* . The algorithm in Simulated SAR Ray Tracing is to simulate the performance of hypothetical sensors systems and to predict the signature of targets from a large number of viewing angles as well as target signatures that are inaccessible. The method is based on the image domain approach that uses a generalization of the physical optics approximation to compute target scattering. The simulated SAR technique can be divided into three steps. First step is the ray-tracing portion, the process of sampling a scene database made of polygons, splines, and Constructive Solid Geometry. The second step is the process of converting the ray-traced information, the ray history, into the electromagnetic (EM) response of the sampled scene data. This portion is trivial and can be negligible. The final step is the process of converting the 2-D array of EM responses into complex images. This involves large data passing and different layout of memory and should pose some problems on memory performance.

- *Image Understanding* (IU). It belongs to the class of target detection and classification problems. The application is composed of three parts: (1) morphological filter, (2) region of interest (ROI) selection and (3) feature extraction. The morphological filter component generates images. It has address to operation ratio of around 2-to-1 (implementation dependent). Thus data starvation may be frequently encountered. The operational and addressing cost of ROI is associated with the internal implementations and the data involved, so no accurate estimation can be given. In the feature extraction step, a gray-level co-occurrence matrix is processed for statistics in image. The cost depends on the number of features or targets presented in the input image. The ratio of computations to addressing can be either high or low, so no estimation can be given here.

- *Multidimensional Fourier Transform.* It is widely utilized in a diverse set of technical fields. The algorithm represented in DIS benchmark is multidimensional Discrete

Fourier Transform (DFT). DFT can accomplish the task in O(NlogN) operations if Fast Fourier Transform is applied. Associated with DFT is the memory bottleneck that results from non-unit-stride memory accesses. No matter what arrangement is made and what memory accesses the inner loop attempt, the outer loop is always opposite or irregular, which prevents a unit-stride access. The implementation tested is the Fastest Fourier Transform in the West (FFTW) and it is a divide and conquer algorithm and it exploits data time locality.

- *Data management* (DM): it is chosen from the area of Data Base Management System (DBMS), which is dominated by archival storage and retrieval of large volumes of essential static data. The focus of this benchmark is on the two weaknesses of conventional DMBS implementations: index algorithms (search by index) and ad hoc query (non-index) processing (search by key). Since both index searching and non-index searching require index query and index management, the bottlenecks associated with index query and index management are of more interests. The indexing method chosen within this benchmark is an R-Tree structure. The R-Tree index is a height-balanced tree containment structure, that is, nodes of the tree contain lower nodes and leaves. Three kinds of operations are associated with R-Tree: query, insert, and delete. The bottleneck associated with query operations is the maximum number of node accesses, which is N, or a complete search over all possible paths, where N is the number of paths of the tree. The maximum cost associated with insert is N+2h, where h is the height of the tree, and is N+h associated with delete operation. The performance improvement of the benchmark depends on the improvement over index maintenance and non-index search.

4.2 DIS Stressmark Suite

DIS Stressmark suite [38] was developeded as complementary to DIS Benchmark suite. Since it was difficult to measure specific elements of interest from large applications, smaller procedures were written and gathered as DIS Stressmark suite.

DIS Stressmarks are small and focus only on particular elements of a problem. Usually they will lose realistic when representing applications. Thus the architectures optimized for DIS Stressmarks may perform worse for general applications. So DIS Stressmarks should be used in support of DIS Benchmarks, not replacing them.

DIS Stressmarks have a large overhead in data initialization. People should exclude this part from tests. It is not our research interests. And also, it is much larger than the

kernel. The performance of the kernel would be hidden if this part is included.

There are seven kernels in DIS Stressmark suite:

- *Pointer.* It repeatedly follows the input-dependent pointers ("hop") to locations in memory. The procedure consists of fetching a small number of words at a given address, finding the median of the values, and using the result and an additional offset to determine the address for the next fetch. The process is repeated until a "magic number" is found, or until a fixed number of fetches have been done. No temporal locality exists if the input is randomized. The number of words fetched at a given address is called size of a "window". Since fetching in a window is contiguous, the larger the window size, the more spatial locality. The kernel exploits no spatial locality if the window size is set to 1.

- *Update.* It is a variation of Pointer. The difference is the following: when a small number of words at a given address is fetched, the first element in the window is updated with the total sum of the window's elements, and then the median is found and used to determine the address for the next fetch. Update has the same "pointer jumping" behaviors as Pointer, thus spatial locality is difficult to exploit.

- *Matrix.* It characterizes operations dealing with data stored in a compact form. In this stressmark, the *iterative conjugate gradient* method is used to solve a linear system, which is represented by the equation $A \bullet x = b$, where A is a sparse nxn matrix, and x and b are vectors with n elements each. As the required method is iterative, the steps are performed until x is found to be within a specified error tolerance, or for a specified maximum number of iterations, whichever occurs first. Different matrix storage schemes may generate different memory behaviors and performance. In our implementation *Compact Row Storage scheme* is used [17]: a sparse matrix is stored in two vectors. The row elements are contiguous stored in a vector, and associated with a vector are the original column indexes. Row-wise accesses are faster than column-wise accesses, as spatial locality is exploited, e.g., in computations such as matrix-by-vector multiplication. In Matrix-vector-multiplications, vectors are indexed non-contiguously and the "forward jumping" behaviors may happen.

- *Neighborhood.* It deals with data that is organized in a two-dimensional grid, and computed by neighborhood operators. The operator can be described as follows: given a ray and a distance along the direction, two points in the grid are determined as neighbors, and a computation is performed on them. The operation is

performed on each valid pair of points chosen in a row-wise fashion. Memory accesses are contiguous along rows, and spread along the direction of the operator. The texture measurements are obtained by estimating a gray-level co-occurrence matrix (GLCM). The matrix contains information about the spatial relationships between pixels within an image. Statistical descriptors of the co-occurrence matrix have been used as a practical method for utilizing these spatial relationships. Two statistical descriptors, GLCM entropy and GLCM energy, are calculated for each valid direction. The descriptors can be estimated by a neighborhood computation using sum-histogram (i.e. a vector element indexed by the value sum of the neighborhood operands is incremented by one) and the difference-histogram (i.e. a vector element indexed by the value difference of the neighborhood operands is incremented by one) as neighborhood operators. The accesses to image have high spatial locality. The operator result has no particular stride depending on the values of two pixels compared. Some temporal locality may exist in difference-histogram when the values of two pixels change in the same scale.

- *Field.* It emphasizes regular access to large quantities of data. It involves scanning for strings using ad hoc. In this way, it tests a system's ability to perform searching when indices are unavailable or inadequate. The procedure consists of searching an array (field) of random words for token strings, which are used as delimiters. All words between instances of the delimiter form a sample set, from which simple statistics are collected. The delimiters themselves are updated in memory. When all instances of a token are found, the process is repeated for a new one. The memory behavior depends largely on the token used for searching matching instances. If a token is not matched in the data field, the searching would be offset by only one position each time. So the sample sets would be contiguous and almost repeatedly scanned. The statistics lists, which are forwarded by one position only when a matching instance is found, would stick to one position repeatedly. So there is high temporal locality when the token matching is a rare case. On the other hand, if the token is matched, the sample sets would forward by the length of the token.

- *Corner-Turn.* It emphasizes effective memory bandwidth without stressing functional units. It involves the matrix transposition ("corner-turn") operation useful in signal processing applications. Although matrix transposition is a required element in other applications within this suite, it involves practically no computation, so memory bandwidth issues are not readily masked behind processing latency. The procedure consists of transposing a matrix of random words repeatedly. It has both in-place and out-of-place modes, referring to whether or not the transposed matrix

must overwrite the original.

- *Transitive Closure.* It emphasizes semi-regular access to elements in multiple matrices concurrently. It requires the solution of the all-pairs shortest path problem, which is fundamental to a variety of computational problems. The procedure utilizes the Floyd-Warshall all-pairs shortest path algorithm []. It accepts as input an adjacency matrix of a directed graph, which is stored in row major format. It then uses a re-current relationship to produce the adjacency matrix of the shortest-path transitive closure. If the dimension of the matrix is n, Floyd-Warshall algorithm takes $O(n^3)$ steps, which asymptotically is no better than n calls to Dijkstra's single-source shortest-paths algorithm ($O(n^2)$). However, this approach is generally considered to operate better in practice than Dijkstra's, especially when adjacency matrices (as opposed to lists) are employed. The program suggested and implemented in Transitive Closure is not the standard Floyd-Warshall's algorithm: the two inner loops are interchanged so most of the memory accesses are in column major, loosing the inherently spatial locality of the original algorithm.

4.3 SPEC95

SPEC95 [49] was the latest version of the world-wide standard for measuring and comparing computer's processors, memory architecture and compiler. It is an improvement over its predecessors for computing-intensive benchmarking. The benchmarks are developed to be as resistant as possible to compiler optimizations, which might not translate into real world performance gains. The benchmarks now run with long times. No small changes or fluctuations in the measurements should have a significant impact on the performance improvement being seen. Some benchmarks have large problems requiring a great amount of resources, while others have smaller ones. So diverse applications are represented. However, SPEC95 is not intended to evaluate the graphics, network or I/O.

SPEC95 is composed of two suites of benchmarks: SPEC CINT95, which includes a set of eight computing-intensive integer and non-floating point benchmarks SPEC CFP95, which includes a set of computing-intensive floating-point benchmarks.

In the following two sections, each of benchmarks will be described in more details covering the application areas represented, tasks accomplished, and characteristics of the routines.

although potentially any format supported by jpeg) is both compressed and decompressed at multiple settings. The difference between the original and decompressed image is evaluated, and simple statistics are taken. The trivial provided the routines jpeg requires too expensive I/O to be acceptable. In order to remedy the situation, this version reads an image into a memory buffer, and processes it repeatedly with different compression settings.

- *134.perl* accomplishes the task of a Shell interpreter. It performs text and numeric manipulations (anagrams and prime number factoring).

- *147.vortex* is a single-user object-oriented database transaction benchmark that exercises a system kernel. It is a subset of a full database program called vortex (vortex stands for "Virtual Object Runtime Expository"). Transactions to and from the database are translated through a schema; a schema provides the necessary information to generate the mapping of the internally stored data block to a model viewable in the context of the application. Vortex has been modified to not commit transactions to memory in order to remove input-output activity.

4.3.2 SPEC CFP95

SPEC CFP95 includes a set of 10 computing-intensive floating-point benchmarks

- *101.tomcatv* is a highly vectorizable, double precision, floating point FORTRAN benchmark (computation on a stream of data). It represents fluid dynamics and geometric translation applications. It is a vectorized mesh generation program. It is part of Prof. W. Gentzsch's benchmark suite. It does little I/O and is described by Prof. Gentzsch as 90 - 98 % vectorizable.

- *102.swim* is a single precision, floating point FORTRAN benchmark. It is used in whether prediction. Swim stands for Shallow Water Model with 1024 x 1024 grid (grid size controlled by parameters N1, N2). The program solves the system of shallow water equations using finite difference approximations on a N1 x N2 grid.

- *103.su2cor* is a double precision, floating point FORTRAN program that is vectorizable. It is used in quantum physics. In this application, program from the area of quantum physics, masses of elementary particles are computed in the framework of the Quark-Gluon theory. The data are computed with a Monte Carlo method taken over from statistical mechanics.

4.3.1 SPEC CINT95

SPEC CINT95 includes a set of eight computing-intensive integer/non-floating point benchmarks:

- *099.go* is an example that utilizes artificial intelligence in game playing. Go plays the game of "go against itself". The benchmark is a stripped down version of a successful go-playing computer program. There is a great deal of pattern matching and look-ahead logic. It is commonly that up to a third of the run-time can be spent in the data-management routines.

- *124.m88ksim* is a simulator for the 88100-microprocessor. It can measure the number of clocks, which an 88100 microprocessor would take to execute a program. It is essentially an integer program, although the exact instruction mix of the simulator depends on the program being simulated. The simulator can pass the system calls from the simulated program through to the host system running the simulator.

- *126.gcc* is a CPU intensive integer benchmark. It is based on the version 2.5.3 GNU C compiler, which is distributed by the Free Software Foundation. The benchmark measures the time the GNU C compiler takes to convert a number of preprocessed source files into the optimized Sparc assembly language (.s files).

- *129.compress* reduces the size of the named files using adaptive Lempel-Ziv coding. Whenever possible, each files is replaced by the one with the extension (.Z). If no files are specified, standard input is compressed to standard output. Compressed files can be restored to their original form using Uncompress. The amount of compression obtained depends upon the size of the input, the number of bits per character, and the distribution of common sub-strings. Compression and decompression programs are used in a wide variety of applications, which require storage and/or transmission of large text files.

- *130.li* is a CPU intensive integer benchmark, which performs minimal I/O. Li is a Lisp interpreter written in C. The workload used is a translation of the Gabriel benchmarks by John Shakshober of DEC.

- *132.ijpeg* represents an image processing application. It performs jpeg image compression with various parameters. First, an original bitmap image (usually GIF,

- *104.hydro2d* is a double precision, floating point arithmetic program that is vectorizable. In this application, program from the area of astrophysics, hydro-dynamical Navier-Stokes equations are solved to compute galactic jets.

- *107.mgrid* is a FORTRAN benchmark used in the area of electromagnetism. It demonstrates the capabilities of a very simple multi-grid solver in computing a three dimensional potential field.

- *110.applu* is a FORTRAN benchmark used in fluid dynamics/math. It solves matrix system with pivoting.

- *125.turb3d* is used for simulating isotropic, homogeneous turbulence in a cube, which has periodic boundary conditions in x, y, z coordinate directions. It solves the Navier-Stokes equations using a pseudo spectral method: Leapfrog-Crank-Nicolson scheme, which is used for time stepping.

- *141.apsi* is a double precision, floating point arithmetic FORTRAN scientific benchmark. It is used to solve for potential temperature, wind, velocity and pressure of pollutants. The synoptic scale components are in quasi-steady state balance, while the mesoscale pressure and velocity are found diagnostically.

- *145.fppp* is a double precision, floating point FORTRAN scientific benchmark. It is a quantum chemistry benchmark. It measures performance on one style of computation (two electron integral derivative), which occurs in the GaussianXX series of programs. It does very little I/O. The input contains as the first entry the number of atoms. The computational time should be proportional to the 4th power of the number of atoms. In order to get this dependence, the atoms are placed in a relatively compact region of space, and are positioned in a graphite-like lattice (as the atoms in fpppp appear to be carbons).

- *146.wave* is a double precision, floating point FORTRAN scientific benchmark. It solves Maxwell's equations and particle equations of motion on a Cartesian mesh, which has a variety of field and particle boundary conditions. The benchmark problem involves 750,000 particles on 75,000 grid points for 40 time steps; about 11 M words (32-bit) of memory are required. Considerable indirect addressing dominates the code's runtime.

5 Experimental Results

In this section, we present a complete view of how the memory adaptation approaches work on SPEC95, DIS Benchmark suite, and DIS Stressmark Suite. We first describe the experimental environments. Then three examples are used to describe how adaptation

approaches work for different memory behaviors. Finally, we show the complete experimental results by benchmark suites. We analyze and demonstrate in detail how different adaptation approaches can have different memory performance. We compare and correlate the performance of these adaptation approaches in a systematic way so that readers can understand and benefit from our analysis mechanism.

5.1 Experiment Setup

The platform, simulator, compiler and baseline architecture we used are introduced in this Section.

5.1.1 Platform

Our experiments are performed on Sun Ultrasparc 5 with the speed 333MHz and 128M RAM.

5.1.2 Baseline Simulator

The simulator used for baseline simulation is a processor simulator *Sim-Outorder*, which is from a processor simulator suite SimpleScalar[8]. The SimpleScalar architecture is derived from the MIPS-IV ISA [41] with small modifications to the semantics. It supports non-blocking cache and speculative execution. Configuration interface is provided. Users can statically configure the architecture before simulation. The relevant configuration flags include:

- max:inst <uint>, which specifies the maximum number of instructions simulated. In our experiments, it is 3 billion, which is large enough to fully execute most applications.

- fastfwd <int>, which specifies the number of instructions skipped before statistics is collected. It is used when we want to skip the data initialization and go directly to the kernel.

- cache:dl1 dl1 : number of sets : block size : associative : replacement policy. It is the level 1 data cache specification. For example, dl1:512:32:2:l defines a level 1 data cache with size 32KB, block size 32B, 2-way associative and least-recently-used replacement policy. This is the data cache specification of the baseline.

- cache:dl2 dl2 : number of sets : block size : associative : replacement policy. It is the level 2 data cache specification. For example, dl2:8192:64:2:l defines a level 2 data cache with size 1MB, block size 64B, 2-way associative and least-recently-used replacement policy. Users can also specify level 2 data cache as a unified cache by

combining level 2 data cache and level 2-instruction cache together. This is the data cache specification of the baseline.

The other related configurations include:

- cache:dl1lat 1: level 1 data cache hit latency. It is one instruction cycle
- cache:dl2lat 8: level 2 data cache hit latency. It is eight instruction cycles.

Please see the Appendix A for more details on configuration flags and their default values.

5.1.3 Adaptive Cache Modules

Two modules are introduced on top of the baseline simulator:

- Victim Cache
- Stream Buffers

Two modules are introduced in place of the baseline simulator:

- Adaptive Line Size cache model, replacing the original cache model,
- Adaptive Fetch Size cache model, replacing the original cache model

Victim Cache model lies between level 1 data cache and level 2 data cache. It is a 32-way fully associative cache with block size 32 Bytes. The replacement policy used is least-recently-used. The access latency is one instruction cycle.

Stream Buffer model is used to support level 1 data cache. It is a four-way pre-fetch buffer, and each way has four elements. Each way is used to keep track of a reference stream with constant stride. The access latency to each buffer is one instruction cycle.

Adaptive line cache model consists of adaptive line size cache controller and adaptive line size cache. Cache lookup in the adaptive line size cache is the same as the cache lookup in a fixed size cache. So is the access latency. The major difference is in the case of miss. For sake of explanation we consider just one level of cache (minor modifications must be applied when there are more than one level). On a miss, the controller predicts the line size for the evicted virtual line in memory (virtual line is the sequence of memory locations that will fit in an adaptive line in the cache) so that any modification of the line size has effect next time the virtual line will be fetched. Adaptive cache consists of different sizes of lines adjusted according to the spatial locality in the past stream. User can specify possible line sizes statistically. The possible line sizes are 16B, 32B, 64B, and

256B in our experiments.

Adaptive fetch size cache model consists of adaptive fetch size cache controller and adaptive fetch size cache. Periodically, the controller changes the cache line size as a function of the measured performance in the previous sample interval. In our experiments, the prediction mechanism is locality-based. The sample interval is 100K instructions. The initial fetch line size is 32 B. The possible fetch line size can be 16B, 32B, 64B, 128B, and 256B. The aging threshold algorithm is used with aging ratio 0.01, lower bound of threshold 0.4, middle threshold 0.55.

5.1.4 Compiler

Benchmarks written in C are compiled using SimpleScalar version of GCC. Those written in FORTRAN are compiled using SimpleScalar version of f77. The optimization flag used is -O3, which performs nearly all the supported optimizations. Function inlining is performed.

5.1.5 Baseline Architecture

The baseline architecture for level one data cache is:

- Cache size: 32KB
- Block size: 32B
- Associative: 2-way
- Replacement policy: least-recently-used

5.2 Example Benchmark Analysis

An analysis of benchmark improvement is important to our evaluation. It helps us to understand how the adaptations exploit the spatial and temporal locality.

Three benchmarks are chosen for evaluation in this section. Each one illustrates one adaptation approach. *Matrix* illustrates Partition based Stream Buffer (SB-PA) and PC based Stream Buffer (SB-PC). *wave5* illustrates VC, and *applu* illustrates ALS and AFL.

5.2.1 Analysis of Matrix: why Stream Buffer Works

As shown in Figure 16, SB-PA and SB-PC reduce the miss ratio of *Matrix* by 80% and 82%, respectively. ALS and AFL reduce it by 72% and 79%, respectively. VC is not

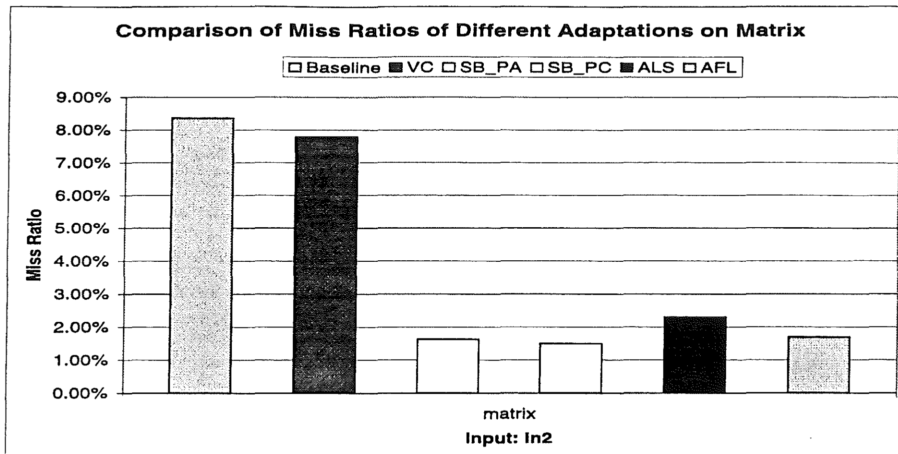


Figure 16: Average Miss Ratios of Adaptations on Matrix with input in2

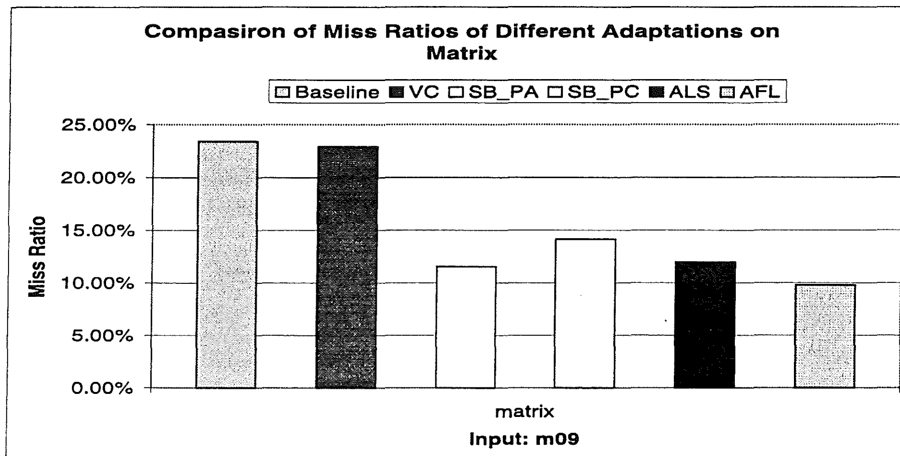


Figure 17: Average Miss Ratios of Adaptations on Matrix with input m09

effective. These experimental results are based on a particular input: in2 (see Appendix B for detail). Performance depends on the size and density of the vectors. For example, using another input m09.in in in Figure 17, the miss ratio reduction using SB-PA and SB-PC is 50% and 39%, respectively. It is 48% and 58% using ALS and AFL, respectively, and 2% using VC. We have noticed that SB is always the most effective even if the input is changed. Without the loss of generality, we choose in2 as representative input.

The algorithm of *Matrix* is the *iterative conjugate gradient* method. The algorithm seeks the solution of an equation, $A \bullet x = b$, where A is a sparse N x N matrix, and x and b are vectors. The matrix storage scheme used is the *Compact Row Storage* [17]: Each row is represented by two vectors. Value vector contains the values of the non-zero elements (at) in the row. And index vector stores the column indexes of the non-zero elements. The kernel includes two kinds of operations:*matrix-by-vector multiplication* and

Basic Linear Algebra Subprograms Level 1 (BLAS 1) operations. A quantitative analysis shows that the number of memory accesses generated by matrix-by-vector multiplication is dominant. And the unit-stride accesses to value vector and index vector are dominant. Conflicts may happen when the index vector conflicts and evicts elements of b. But this happens very rarely, as is shown in Figure 16.

SB captures the unit-stride accesses. Therefore every time when the data in value vector and index vector are needed, they are in the cache or SB. Due to the unit-stride accesses, all the elements in each cache line are used. Such characteristics are also be exploited by ALS and AFL.

5.2.2 Analysis of Wave5: Why Victim Cache Works

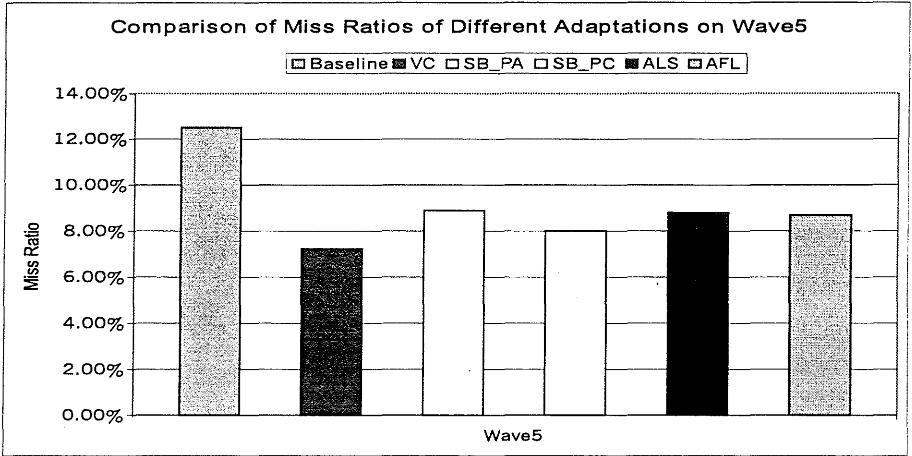


Figure 18: Average Miss Ratios of Adaptations on wave5 with reference input

The miss ratio of *wave5* has been reduced by 42% using VC, 29% and 36% using SB-PA and SB-PC respectively, and around 30% and 30% using ALS and AFL respectively. The Figure 18 shows the performance of the memory adaptation approaches. It can be figured out that at least 42% misses are conflict misses. This is counter-intuitive with respect to the miss distribution achieved in Section 2, where conflict misses are a small fraction of the total misses. There might be the following reasons:

- Adaptations have been simulated for 3 billion instructions, which is about half of the entire execution. A partial behavior is exploited, not an average behavior. Indeed, the simulated instructions have a large portion of conflict misses and therefore VC is the most effective one.
- The conflict misses measured in Section 2 are underestimated due to the non-optimal replacement policy.

In more detail, we did profiling to generate the procedure relationships in *wave5*. We assume that the procedure with longer execution the time causes the larger portion of the misses. In *Wave5*, the dominating procedure is "PARMVR", which accounts for 65% of the total execution time. In this procedure, there are two kinds of memory access patterns:

- Four vectors with high temporal locality have unitary stride accesses.
- Some vectors have the "Pointer Jumping" characteristic, i.e. A[B[j]].

For most of time, more than two vectors are accessed concurrently. So conflicts happen frequently in a 2-way associative cache. "Pointer Jumping" characteristics may cause more conflicts. VC can capture conflict misses from vector accesses with high temporal locality.

5.2.3 Analysis of Applu: Why Adaptive Line/Fetch Size Caches Work

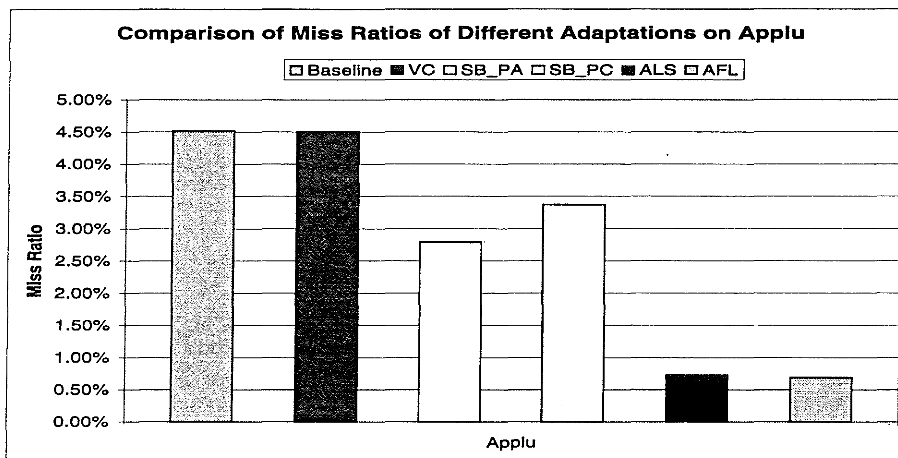


Figure 19: Average Miss Ratios of Adaptations on wave5 with reference input

The miss ratio of *applu* has been reduced by 84% and 85% using ALS and AFL respectively, by 25% and 40% using SB-PA and SB-PC respectively, and no reduction using VC. The experimental results show that most of the locality in *applu* is spatial locality. There are too many concurrent streams and SB is not effective, but ALS and AFS are.

The miss distribution shows that around 80% of misses are compulsory and capacity misses. So there is improvements space for ALS, AFL and SB. In the kernel, there are two kinds of memory access patterns:

- There are unitary stride accesses.

- There are seven streams in the same loop. Each of them has a constant stride. Four of them refer to the same vector. The other three refer to different vectors.

ALS and AFL are able to exploit the spatial locality in both unitary stride accesses and stream accesses. SB-PC has only 4 ways of pre-fetch buffers, which is insufficient for seven streams. SB-PA is a little bit more effective than SB-PC. It allocates buffers only for those three reference streams for different vectors, and the number of pre-fetch buffers is sufficient.

5.3 Results per Benchmark Suite

This section shows the effectiveness of memory adaptation approaches by benchmark suites. As summary for all benchmarks, the average miss ratio reduction is 40% and 52% using SB-PA and SB-PC respectively; it is 32% and 36% using ALS and AFL respectively; it is 14% using VC.

5.3.1 Experimental Results for DIS Benchmark Suite

In this section we present the experimental results of DIS benchmarks except the Method of Moment, which is presented separately in Section 2.4. We use MoM as a motivating example to show that adaptation is very effective in short periods of application execution time, but is undetectable by the average miss ratio.

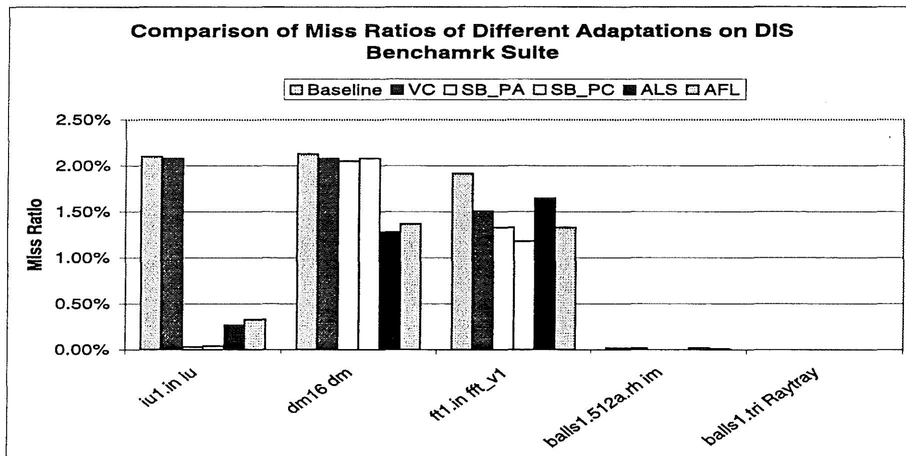


Figure 20: Average Miss Ratios of Adaptations on DIS Benchmark Suite

DIS Benchmarks have small miss ratios because they have been developed to fit contemporary architectures. Three of the DIS Benchmarks in Figure 20, *IU, DM* and *FFT* have relatively high miss ratios (around 2%), while the other two, *IM* and *Raytray*, have

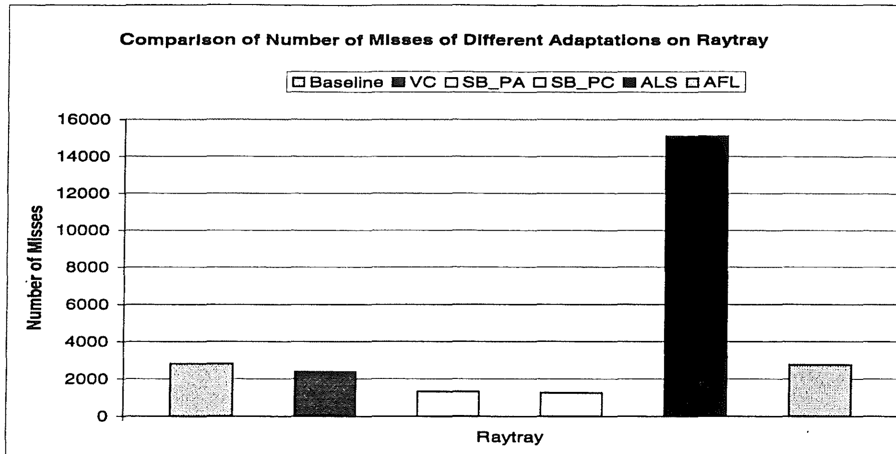


Figure 21: Average Miss Ratios of adaptations on Raytray

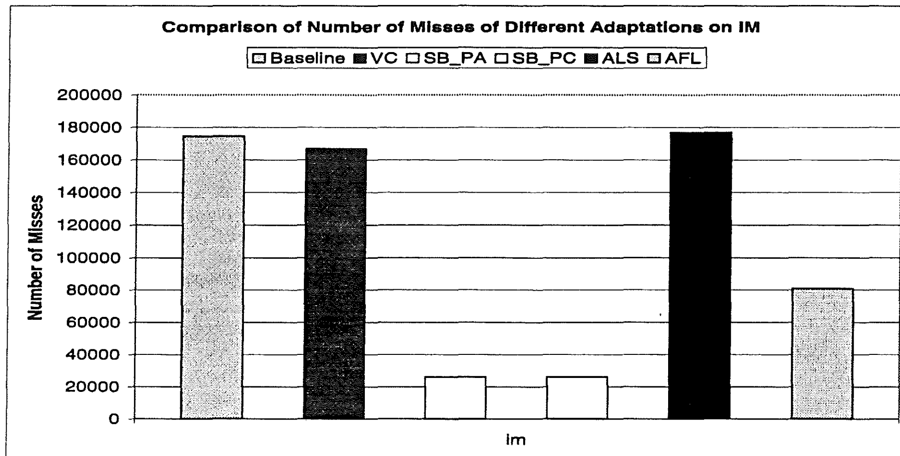


Figure 22: Average Miss Ratios of adaptations on IM

very small miss ratios (less then 0.1%). Due to the different order of magnitudes, we show the last two benchmarks in separate charts in Figure 21 and Figure 22, where the metric used is the number of total misses.

SB is effective for *IU, FFT,* and *IM.* SB-PA and SB-PC have similar performance. It suggests that the references access the separate address "chunks" and have constant strides. (We do not discriminate SB-PA and SB-PC in our discussion below. We use SB-PC as representative and use SB to refer both of them). Almost all the miss latency in *IU* is hidden. Its memory access pattern is regular and predictable. The number of concurrent reference streams is not greater than four. For*IM*, SB hides the latency of 85% of total misses. Only 30% of miss latency in *FFT* is hidden, which implies that capacity and cold misses are not dominant for this input set. There is no latency hiding for *DM* using SB (2% improvement). We infer from the experimental results (comparing with

ALS and AFL) that two possible situations exist in *DM*:

- there is no regular reference streams,
- there are too many concurrent reference streams.

For Raytray, SB hides the latency of around 50% of misses.

ALS and AFL have comparable performance as SB in *IU*, which is 85% miss ratio reduction. The memory access patterns in *IU* are regular and have good spatial locality. ALS and AFL have 40% of miss reduction in *DM*, which is much better than SB (2% latency hiding) and VC (2% reduction). In *IM*, AFL removes 50% of misses. ALS increases the number of misses because of unsuccessful line size prediction. ALS and AFL are effective for *FFT*. We think the performance would be better if the upper bound of the line size could be larger than 256B. ALS and AFL are not effective for *Raytray*. ALS introduces misses (12 times that of the baseline).

VC is effective for *FFT* and reduces the miss ratio by 21%. The number of conflict misses in this benchmark is significant so that SB is not very effective. The kind of conflict in this benchmark is independent from the line size. Therefore ALS and AFL is not very effective either. In *IU* and *DM*, VC reduces the miss ratios by 0.95% and 2% respectively.

In DIS Benchmark suite, we have observed:

- SB is the most effective one among the three kinds of adaptations. It is effective on *IU*, *FFT*, *Raytray*, and *IM*. The average improvement is 55%.

- The average performance improvements using ALS and AFL are 35% and 51%, respectively (we did not take *Raytray* into calculation because we think it is an exception). They are effective for four of the DIS Benchmark suite: *IU*, *DM*, *FFT*, and *IM*. For *DM*, ALS and AFL are more effective than SB because they can exploit some kind of spatial locality that SB cannot, i.e., large number of reference streams and irregular memory access pattern.

- In general, VC is not effective in DIS Benchmark suite. The average miss ratio reduction is 9%. Only *FFT* is significantly improved (21%).

5.3.2 Experimental Results for DIS Stressmark Suite

Memory adaptations improve four of the six DIS Stressmarks. They are not effective for *Pointer* and *Update* because these benchmarks have neither spatial nor temporal locality. Thus we illustrate the miss ratios in two figures: Figure 23 shows *Field, Matrix, Neighborhood,* and *Transitive Closure*, Figure 24 contains *Pointer* and *Update.*

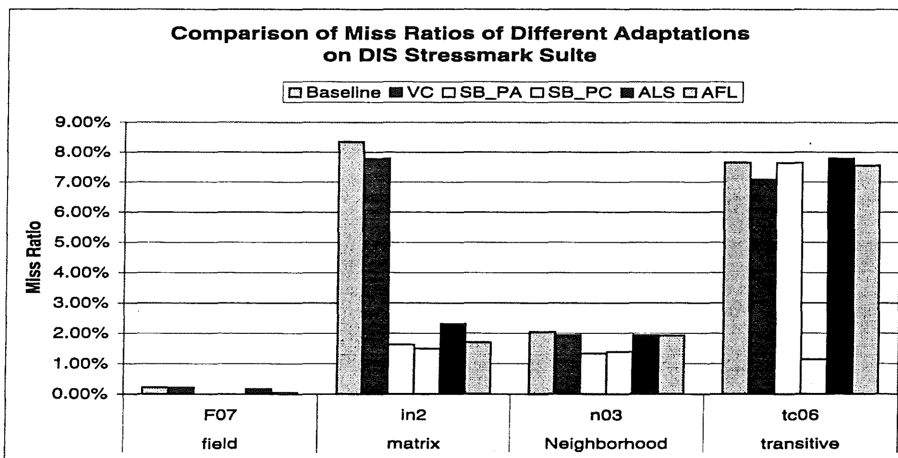


Figure 23: Average Miss Ratios of Adaptations on DIS Stressmark Suite

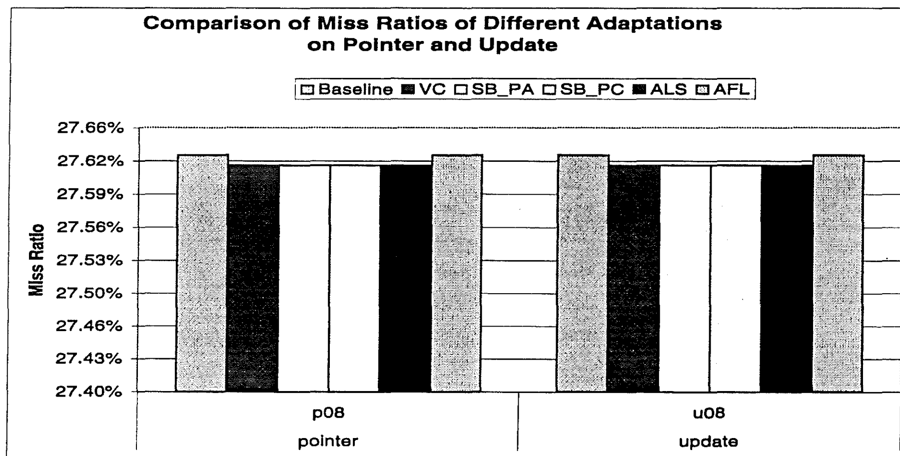


Figure 24: Avearge Miss Ratios of Adaptations on Pointer and Update

The kernels of *Pointer* and *Update* have very high miss ratios (27%) because of the random memory accesses. Every memory access through a pointer might result in a miss (So the miss ratio should be 100%!). In practice, the optimizing compiler generates an executable that has "spills" in memory. Since these "spills" are generally hit in cache, the miss rate of the kernel is dropped. Because the random memory accesses cannot be optimized, we will not discuss *Pointer* and *Update* in our analysis. In this section, our

focus is on the Stressmarks that can be improved.

Note that data initializations of the DIS Stressmarks are not negligible. We tested the stressmark kernels separately.

SB-PA has similar performance of SB-PC. It means that most of the reference streams have separate address space and regular strides. There is an exception: SB-PC improves *Transitive Closure* (85%), while there are almost no improvements by SB-PA (0.13%). We report the kernel here in Figure 25: DIN is a matrix with N rows and N columns stored in row major, so is DOUT.

```
for ( k = 0 ; k < n ; k ++ )
   for ( i = 0 ; i < n ; i ++ )
      for ( j = 0 ; j < n ; j ++ ) {
         old = DIN ( j , i );
         new1 = DIN ( j , k ) + DIN ( k , i );
         DOUT ( j , i ) = ( new1 < old ? new1 : old );
}
```

Figure 25: Kernel of Transitive_Closure

Using SB-PA, the first two references: DIN(j,i) and DIN(j,k), specify a stride. The third one, DIN(k,i), which is a constant reference in the inner loop, does not confirm the stride. No stride can be established. A simple optimization, i.e. move the loop invariant outside the inner loop, would circumvent the problem. In contrast, SB-PC recognizes the column major accesses, DIN(j,i) and DIN(j,k), and detects the large strides. As previously adopted, we use the notation "SB" to refer both pre-fetch schemes. SB hides the latency of 80% of the total misses for *Matrix* and around 30% for *Neighborhood*. It hides nearly all the miss latencies in *Field* (100%). SB is effective because most memory accesses have constant strides, and more than 99% of the misses in *Matrix* and *Neighborhood* are cold and capacity misses. In *Field*, almost all the misses are compulsory misses. *Field* has a very big working set and it has to fetch new data constantly.

ALS and AFL are effective for *Matrix*, in which they have comparable performance with SB. They are also effective for *Field* (50% improvement) because of its spatial locality. They are not effective for *Neighborhood*: by an elimination process and by the average line size from the experimental results (larger than 32B), we think that the absence of improvement is due to the increase of interferences that offsets the benefit of a larger line size. ALS and AFL are not effective for *Transitive Closure* (less than 1% improvement). In fact, there is no spatial locality because the row major matrix is accessed by column.

VC is not effective for all the stressmarks. For *Matrix*, The improvement is %. For *Neighborhood*, we achieve 5% improvement. For *Transitive Closure*, we use input tc06, for which eight columns of the adjacent matrix can fit in the cache and conflict misses do not happen. So VC is not effective, even if the input sizes grow larger such that less than eight columns can fit in the cache and capacity misses arise. VC cannot remove these capacity misses. *Field* has very small baseline miss rate(almost 0%) such that VC is not effective.

In DIS Stressmark suite, we observe:

- SB is effective for all the DIS Stressmarks except *Pointer* and *Update*. The average miss ratio reduction is 74% (we didn't take *Pointer* and *Update* into calculation because they are not optimizable.)

- ALS and AFL only improve *matrix* and*Field*. The average miss ratio reduction is 25 % and 40% respectively, excluding *Pointer* and*Update*. The performance of ALS and AFL may be restricted by randomized accesses, which is common in DIS Stressmark suite.

- VC is not effective for DIS Stressmark suite.

- We found that SB-PA is not effective for *Transitive Closure* because the executable does not have such memory access pattern that can be exploited. We manually moved the loop invariant DIN(k, i) out of the inner loop, and we executed it again. Now SB-PA is very effective and Transitive Closure gets 50% improvement. Using this example, we show that just using benchmarks to evaluate architectural mechanisms is not accurate. Architectural mechanisms may have the potential to optimize the applications but the executables generated do not give such chance.

5.3.3 Experimental Results for SPEC95

We simulated 17 out of 18 SPEC95 benchmarks with the reference inputs.The average miss ratio is shown in Figure 26. Each benchmark is improved by at least one adaptation approach. The average miss ratio reduction is 40% and 52% using SB-PA and SB-PC respectively, 31% and 22% using ALS and AFL respectively, and 19% using VC.

SB is effective for all the tested SPEC95 benchmarks. For some benchmarks, (i.e., *compress95, su2cor, apsi, turb3d, wave5, swim*), SB-PC is more effective than SB-PA.

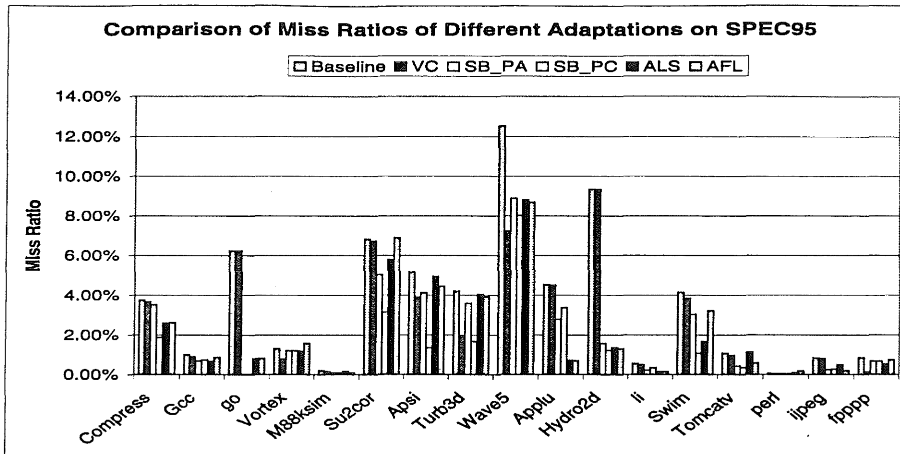


Figure 26: Average Miss Ratios of Adaptations on SPEC95

Swim is a good example to explain this: there are references that belong to different memory chunks but the sequence of references inside a chunk are not in constant stride.

For*applu* and *li*, SB-PA works better than SB-PC. We use *applu* as an example to explain this. There are more than four reference streams in *applu.* SB-PC detects the strides for all of them and allocates the space in Stream Buffer. However, these streams would evict each other because there are only four ways in Stream Buffer. SB-PA instead considers several of these streams to be in the same chunk, and monitors the strides among them, just as for one reference stream. Finally the number of streams is reduced to be less than four.

As usual, we use SB-PC as representative because we observe that SB-PC is in general more effective than SB-PA, and we use SB to refer to both of them. We are going to introduce the improvements from the largest to the smallest. Some benchmarks have relatively high baseline miss rates and SB is able to optimize them significantly, such as *go* (99%), *apsi* (73%), *turb3d* (60%), *swim* (74%), and *hydro2d* (87%). It can be inferred that the reference streams in these benchmarks are purely sequential, and almost all the spatial locality can be exploited. The miss distribution from the experimental results supports this inference, i.e., nearly 100% misses in*hydro2d* are capacity and cold misses, which provides space for SB to work in.

The improvement on *Compress* (49%), *Vortex* (60%), *Su2cor* (54%),*wave5* (36%), and *applu* (25%), is significant. There is still high spatial locality in them, e.g., the portion of cold and capacity misses are 85% in *su2cor*, 97% in *wave5.* Some spatial locality is not exploited. The reason is that there are too many concurrent reference streams, i.e., *wave5* and *applu.*

Some benchmarks have very small baseline miss ratios (i.e. ranging from 0.05% for *perl* up to 1.07% for *tomcatv*), and SB still performs very well on them: *m88ksim* (60%),*li* (40%), *tomcatv* (66%), and *perl* (40%). From the performance point of view, the improvement is negligible. But from the architectural point of view, SB is effective.

SB does not significantly improve *vortex*, (only 7% improvement) because *Vortext* has high percentage of conflict misses (15% for *vortex*).

ALS and AFL are effective for most SPEC95 benchmarks. They have very good performance on some of the benchmarks, such as *hydro2d* (85%/86%), *applu* (84%/85%), *swim* (59%/22%), and *compress* (31%/30%). *Hydro2d* has very regular and short strides with high spatial locality.*Applu* has many sequential reference streams. For *swim*, AFL is less effective than ALS because AFL takes more time to reach the "optimal" line size than ALS. For benchmarks such as *wave5* (29%/31%), *apsi* (4%/13%) and *turb3d* (4%/6%), ALS and AFL are not very effective because the benchmarks have no negligible percentage of conflict misses (25% and 19% respectively and see characterization of wave 5 in Section 5.2.2).

ALS and AFL have a drawback that other approaches do not have: misses can be introduced and memory performance can be degraded. In other words, the overall number of misses is larger with adaptation than without. For example, *vortex* has a degradation of 21.54% due to AFL, and *tomcatv* has a degradation of 4.67% due to ALS.

VC is effective for *apsi* (24%), *turb3d* (53%), and *wave5* (42%). It has an average improvement of 20%. Eight of SPEC95 benchmarks have been improved at least by 12%.

We observe that ALS and AFL are effective and they have an average improvement of 31.01% and 22.25%, respectively. Both can improve nine benchmarks by at least 30% and 30%, respectively. ALS introduces more misses than the ones it reduces for tomcatv and perl. AFL has the same problem as ALS but for vortex, su2cor and perl. If we compute the average miss ratio reduction without the above benchmarks, it is 40% for ALS and 45% for AFL. Furthermore, those cases have very small baseline miss ratios, up to 1%. The performance of these benchmarks would not vary significantly even if degraded.

6 Conclusion

In this report, we have evaluated Victim Cache (VC), Partition based and PC based Stream Buffer (SB-PA and SB-PC), Adaptive Line cache (ALS) and Adaptive fetch size

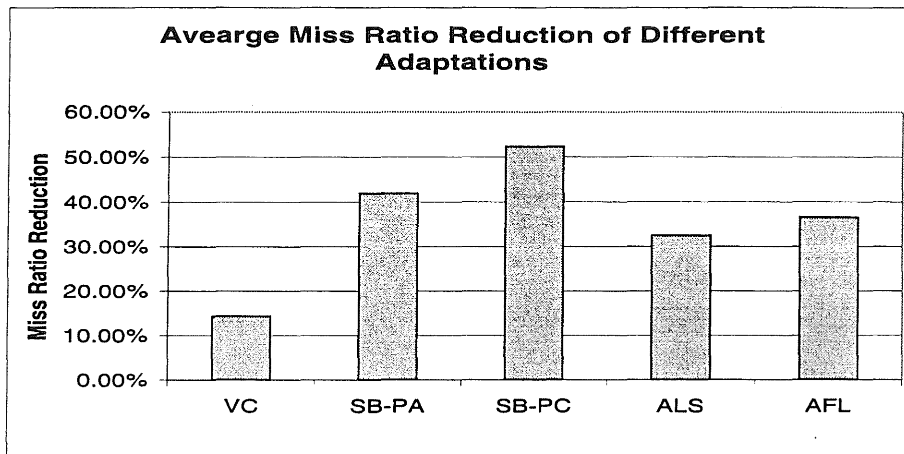


Figure 27: Average Miss Ratio Reduction of Different Adaptations

cache (AFL).The averag emiss ratio reduction is shown is Figure 27 Based on the experiments coming from a total of 28 benchmarks, the following conclusions can be drawn:

- Among architectural assists, SB is effective in most cases. It can improve the performance of all the 28 benchmarks. ALS and AFL are effective, but less than SB. ALS can improve 23 benchmarks. For the other 5 benchmarks, the miss ratios with ALS are higher than baseline because of unsuccessful prediction. AFL can improve 25 benchmarks. For the other 3 benchmarks, the miss ratios with ALS are higher than baseline. VC is effective only in a small set of benchmarks when there is locality but conflicts arise. The miss ratio reduction is less than 10% for 18 benchmarks.

- ALS and AFL are "potentially" the most versatile one. SB has a fixed number of buffers. It is not well utilized unless the number of concurrent streams is equal to the number of buffers. ALS and AFL do not have such a constraint. However, ALS and AFL have the problem to fully exploit spatial locality due to line size misprediction. The prediction is based on the information collected in the past, and the assumption that the similar behavior would occur in the future. However, this assumption is not always true. (see also in [56])

- For some benchmarks (i.e., DIS Benchmarks), we have found that the average miss ratio was not capable in efficiency evaluation. The adaptation may be effective in part of the application, but the effectiveness may not be seen "globally". For example, we show the temporal behavior of MoM. And we have found that ALS, AFL and SB can reduce the miss rates from 40% to 60% in the crucial part of the execution (level 4), but no improvement in the average behavior.

Besides these, we observe that VC is efficient in removing conflict misses. However, this efficiency may be limited by it capacity. The optimal size of VC is application dependent.

SB can be optimized if the number of ways is adaptive. There should be a good way to determine the trade-off between capacity and cost. And it is still an open problem.

References

[1] A. Aggarwal, A.K. Chandra and M. Snir, "Hierarchical memory with block transfer". *1987 IEEE.*

[2] T. Alexander and G. Kedem, "Distributed Prefetch-buffer/cache design for high performance memory systems". In *Proceeedings of 2nd IEEE Symposium on High-performance Computer Architecture*, 1996. IEEE Press, Piscataway, NJ, 254-263.

[3] A. Aggarwal, B. Alpern, A.K. Chandra and M. Snir, "A model for hierarchical memory". *Proc. of 19th Annual ACM Symposium on the Theory of Computing*, New York,1987,305-314.

[4] D. Bernstein, D. Cohen, and A. Freund, "Compiler techniques for data prefetching on the PowerPC", In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques (PCAT'95)*, Limassol, Cyprus, June 27-29, 1995.

[5] G. Bilardi, E. Peserico, "A Characterization of Temporal Locality and its Portability Across Memory Hierarchies". *ICALP 2001, International Colloquium on Automata, Languages, and Programming*, Crete, July 2001.

[6] G. Bilardi, A. Pietracaprina, and P. D'Alberto, "On the space and access complexity of computation dags", *26th Workshop on Graph-Theoretic Concepts in Computer Science,* Konstanz, Germany, June 2000.

[7] G.Bilardi, P.D'Alberto and A.Nicolau, "Fractal Matrix Multiplication: a Case Study on Portability of Cache Performance". *Workshop on Algorithm Engineering 2001*, Aarhus, Denmark.

[8] D.C. Burger and T. M. Austin. "The SimpleScalar Tool set, Version 2.0". *Computer Architecture News*, 25 (3), pp. 13-25, June 1997. Extended version appears as UW Computer Sciences Technical Report No.1342, June 1997.

[9] M. L. C. Cabeza, M. I. G. Clemente, and M. L. Rubio, "Cachesim: a cache simulator for teching memory hierarchy behavior", *Proceedings of the 4th annual Sigcse/Sigue on Innovation and Technology in Computer Science education,* 1999, p. 181.

[10] D. Chiou, P. Jain, L. Rudolph, Devadas, "Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches". *Proceedings 2000. Design Automation Conference.Proceedings of ACM/IEEE-CAS/EDAC Design Automation Conference,* Los Angeles, CA, USA, 5-9 June 2000.) New York, NY, USA: ACM, 2000. p.416-19..

[11] T.F. Chen, J.L. Baer, "Effective Hardware-based data prefetching for high-performance processors", *IEEE Trans. Comput.* 44, 5 May 1995, 609-623.

[12] P. Crowley and J.L. Baer, "On the use of trace sampling for architectural studies of desktop applications ", *Proceedings of the international conference on Measurement and modeling of computer systems,* 1999,

[13] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared-memory multiprocessors". In *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, 1993, P. 56-63.

[14] P.D'Alberto, G.Bilardi and A.Nicolau, "Fractal LU-decomposition with partial pivoting". *ICS Technical Report TR# 00-22.*

[15] A. Dan and D. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes". *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems,* 1990, P. 143-152.

[16] J.J. Dongarra, I.S. Duff, D.C. Soransen and H.A. van der Vorst, "Numerical Linear Algebra for Performance Computers", *ed. SIAM Asymptotic Analysis.*

[17] S.C. Eisenstat, M.C. Gursky, M.H. Schultz, and A. H. Sherman, "Yale Sparse Matrix Package", *Yale University Department of Computer Science Technical Report,* volumes 112 and 114, 1977.

[18] K. Farkas, P.Chow, N.Jouppi, and Z.Vranesic. Memory-System Design Considerations for Dynamically-Schedulled Processors. In *24th Annual International Symposium on Computer Architecture (ISCA)*, June 1997.

[19] M. Frigo and S.G. Johnson, "The fastest Fourier transform in the west". *MIT-LCS-TR-728 Massachusetts Institute of technology*, Sep. 11 1997.

[20] M. Frigo, C.E. Leiserson, H. Prokop and S. Ramachandran, "Cache- oblivious algorithms", *Proc. 40th Annual Symposium on Foundations of Computer Science*, (1999)

[21] A. Gonzalez, C.Aliagas, and M. Valero, "A data Cache with Multiple caching strategies tuned to different types of locality". *Intl. Conference on Supercomputing,* pp. 338-347, 7 April 1995.

[22] R. Gupta, "Achitectural adaptation in AMRM machines", *Proceedings of IEEE Computer Society Workshop on VLSI 2000* , Los Alamitos, CA, USA, p75-79. *System Design for a System-on-Chip Era* , Orlando, FL, USA, 27-28 April, 2000.

[23] R. Gupta, "AMRM: Project Technical Approach A Technology and Architectural View of Adaptation", *Project Kickoff Meeting* , November 5, 1998, *http://www.ics.uci.edu/ amrm/slides/amrm_structure/pta/sld001.htm*

[24] Hong Jia-Wei and T.H. Kung: I/O complexity, "The Red-Blue pebble game". *Proc.of the 13th Ann. ACM Symposium on Theory of Computing* Oct.1981,326-333.

[25] X.Ji, D.Nicolaescu, A.Veidembaum, A.Nicolau and R.Gupta "Compiler-Directed Cache Assist Adaptivity". *ICS Techincal Report #00 17*, May 2000.

[26] T.L. Johnson, D. A. Connors, W. W. Hwu, "Run-time adaptive cache management", *Proceedings of the Thirty-First Hawaii International Conference on System Sciences,* (Cat. No.98TB100216), (vol.7), Kohala Coast, HI, USA 6-9 Jan. 1998; IEEE Comput. Soc, 1998. p.774-775, vol.7.

[27] T. L. Johnson, W. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis", *24th Annual International Symposium on Computer 24th Annual International Symposium on Computer Architecture.* ISCA'97 . ACM, May 1997. p.315-326.

[28] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache pre-fetch buffer", *25 years of the international symposia on Computer architecture (selected papers)*, 1998, Pages 388 - 397.

[29] B.Kagstrom, P.Ling and C.Van Loan,"GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark". *ACM transactions on Mathematical Software*, Vol24, No.3, Sept.1998, pages 268-302.

[30] A. Ki, A. E. Knowles, "Adaptive data pre-fetching using cache information", *24th Annual International Symposium on Computer Architecture.* ISCA '97, Denver, CO, USA, 2-4 June 1997. ACM, May 1997. p.315-326.

[31] D. Kim, E. Kim, J. Lee, "A virtual cache scheme for improving cache-affinity on multiprogrammed shared memory multiprocessors", *High Performance Computing Systems and Applications*, Kluwer Academic Publishers, 1998. p.249.

[32] P.M.Kogge, "Summary of the architecture group finding", In *PetaFlops Architecture Workshop (PAWS)*, April 1996.

[33] M. Martonosi, A. Gupta, and T. Anderson. "Effectiveness of Trace Sampling for Performance Debugging Tools." *Proc. 1993 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems,* 1993.

[34] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors", *Parallel Distrib. Comput*, 12, 2 June, 1991, P.87-106.

[35] "MIPS R10000 Microprocessor", User Manual Version 2

[36] J. Mu•oz, "Presentation at Data-Intensive Systems Principal Investigators Meeting", 1 October, 1998, (on line at *http://www.dapar.mil/ito/research/pdf_files/dis_approved.pdf/*).

[37] J. Mu•oz, "Data-Intensive Systems Benchmark Suite Analysis and Specification", Submitted by Atlantic Aerospace Electronic Corp to DARPA/ITO, 30 June 1999. (On line at the *http://www.aaec.com/projectweb/dis/DIS_Benchmarks_v1.pdf)*

[38] J. Mu•oz, "DIS Stressmark Suite: Specification for the Stressmarks of the DIS Benchamrk Project Version 1.0", Submitted by Atlantic Aerospace Division, Titan System Corp. to DARPA/ITO, 24 August, 2000. (on line at *http://www.aaec.com/projectweb/dis/DIS_Stressmarks_v1_0.pdf)*

[39] S. Palacharla, R. E. Kessler, "Evaluating stream buffer as a secondary cache replacement", *Intl. Symposium on Computer Architecture*, pp.24-33, May 1994.

[40] Panda, P.R.,Nakamura, H., Dutt, N.D., Nicolau,A. "Augmenting Loop Tiling With Data Alignment For Improved Cache Performance". *IEEE Transactions on Computers,* vol.48, (no.2), IEEE, Feb. 1999. p.142-9.

[41] C. Price, "MIPS IV Instruction Set, revision 3.1", *MIPS Technologies, INC.*, Mountain View, CA, January 1995.

[42] S. Przybylski, "The performance impact of block sizes and fetch strategies", In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA 1990, P160-169.

[43] S. Saulsbury, F. Pong, and A.Nowatzyk, "Missing the memory Wall: The case for Processor/Memory Integration", *Intl. Symposium on Computer Architecture*, pp. 90-101, 1996.

[44] John E.Savage, "Space-Time tradeoff in memory hierarchies". *Technical report* Oct 19, 1993.

[45] W. W. Schilling, Jr; M. Alam, "The impact of pre-fetching and victim caching on computer systems performance", *Proceedings of the ISCA 12th International Conference (ISCA)*, 1999. p.271-276.

[46] W. W. Schilling, Jr, M. Alam, "Performance simulation of the combination of prefetching and victim caching". *Proceedings of the ISCA 15th International Conference Computers and Their Application (ISCA)*, 2000. p.284-287.

[47] A.J. Smith, "cache memories", *ACM Computing Surveys*, 14, 3(Sept.), 1982, P. 473-530.

[48] Y Song and Z. Li, "New Tiling techniques to improve cache temporal locality", *Proceedings of the ACM SIGPLAN Conference on Programming language Design and implementation,* 1999, p. 215-228.

[49] http://www.spec.org/osg/cpu95/

[50] D. Stiliadis, A. Varma, "Selective victim caching: a method to improve the performance of direct-mapped caches", *IEEE Transactions on Computers*, vol.46, (no.5), IEEE, May 1997. p.603-610.

[51] W. Tang, A. Veidenbaum, A. Nicolau, R. Gupta, "Adaptive Line Size Cache",*UC, Irvine, Technical Report ICS-TR-99-56*, Nov. 1999.

[52] W. Tang, A. Veidenbaum, A. Nicolau, R. Gupta, "Cache with Adaptive Fetch Size", *UC, Irvine, Technical Report ICS-TR-00-16*, April 2000.

[53] S.Toledo, "Locality of reference in LU decomposition with partial pivoting". *SIAM J.Matrix Anal. Appl.* Vol.18, No. 4, pp.1065-1081, Oct.1997

[54] S.P. Vanderwiel and D.J.Lilja, "Data Prefetch Mechansims", ACM Computing Surveys, Vol. 32, No2, June 2000.

[55] A.V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau and X. Ji, "Adaptive Cache Line Size to Application Behavior", In *Proceedings of International Conference on Supercomputing (ICS).* June 1999, pp.145-154.

[56] P.V.Vleet, E.Anderson, L.Brown, J.Baer, A. Karlin."Pursuing the Performance Potential of Dynamic Cache Line Sizes", *Proceedings of the International Conference on Computer Design (ICCS'99)*, October 1999.

[57] R.C.Whaley and J.J.Dongarra, "Automatically Tuned Linear Algebra Software". http://www.netlib.org/atlas/index.html

[58] M.Wolfe, "More iteration space tiling", *Proceedings of Supercomputing*, Nov.1989, pg. 655-665.

[59] M.Wolfe and M.Lam, "A Data locality optimizing algorithm", *Proceedings of the ACM SIGPLAN'91 conference on programming Language Design and Implementation*, Toronto, Ontario,Canada,June 26-28, 1991.

[60] M. Zagha, B. Larson, S. Turner and M. Itzkowitz "Performance Analysis Using the MIPS R10000 Performance Counters" *Proc. Supercomputing 1996*, Nov. 96, Pittsburgh, PA. http://www.sgi.com/processors/r10k/timing/perfcount.html

[61] D. F. Zucker, R. B. Lee, M.J. Flynn, "Hardware and software cache pre-fetching techniques for MPEG benchmarks", *IEEE Transactions on Circuits and Systems for Video Technology,* vol.10, (no.5), IEEE, Aug. 2000. p.782-796.

Appendix A: SimpleScalar Configurations

Besides the configurations we described in section 5.1.2, there are some other configurations. These configurations are common to all the cache modules (VC, SB, ALS/AFL). These configurations include:

- fetch:ifqsize instruction fetch queue size. For example, -fetch:ifqsize 4 in our experiments specifies that four instructions can be fetched at one time.

- fetch:mplat extra branch mis-prediction latency. For example, -fetch:mplat 3 in our experiments, specifies that the penalty of the branch mis-prediction is 3 instruction cycles.

- fetch:speed speed of front-end of machine relative to execution core. For example, -fetch:speed 1 in our experiments specifies that .

- bpred branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}. In our experiments, it is -bpred bimod, specifying that the branch predictor is bimodal.

- bpred:bimod bimodal predictor table size. For example, -bpred:bimod 2048 in our experiments specifies that there are 2048 entries in the table.

- bpred:2lev <l1size><l2size><hist_size><xor>. For example, -bpred:2lev 1 1024 8 0 will specify that there is one entry at the 1st level branch history table, 1024 entries at 2nd level branch pattern table, the history table has 8 bits shift register. Dont xor the history and addess for the index of the 2nd level pattern table.

- bpred:comb combining predictor meta_table_size. For example, -bpred:comb 1024 specifies that the meta_table has 1024 entries.

- bpred:ras return address stack size. For example, -bpred:ras 8 specifies that return address stack size is 8 entries.

- bpred:btb <num_sets><associativity>. For example, -bpred:btb 512 4 specifies that BTB has 512 sets with each set 4 ways. So the history of 2048 branch instructions can be recoreded.

- bpred:spec_update speculative predictors update in {ID(Instruction decoding) | WB(Write Back)}. The default specification is bpred:spec_update NULL, which means non-speculation.

- decode:width instruction decode bandwidth. For example, -decode:width 4 in our experiments specifies that four instructions can be decoded in one instruction cycle.

- issue:inorder {TRUE|FALSE} specifies whether to run pipeline with in-order issue or not. In our experiments, it is -issue:inorder FLASE.

- issue:wrongpath {TRUE|FALSE} specifies whether to issue instructions down to the wrong execution paths. The default is issue:wrongpath TRUE.

- commit:width number of instruction committed per cycle. In our experiments, -commit:width 4 specifies that four instructions can be committed per cycle.

- ruu:size size of register update unit (RUU). In our experiments, -ruu:size 16 specifies that the of register update unit is 16.

- lsq:size load/store queue size. In our experiments, -lsq:size 16 specifies that the load/store queue can store 16 load/store instructions.

- cache:il1 L1 instruction cache configuration, {<config>|dl1|dl2|none}. It specifies that level 1 instruction cache can be separate instruction cache, or use the same cache as level 1 or 2 data cache, or there is no instruction cache. In our experiments, it was specified as cache:il1 il1:512:32:2:l, which defined a level 1 instruction cache with 512 sets, 32B line size, direct-mapped, and least recently used replacement policy.

- cache:il1lat level 1 instruction cache access latency. In our experiments, it is specified as cache:il1 1.

- cache:il2 level 2 instruction cache configuration, {<config>|dl2|none}. It has similar meaning as level 1 instruction cache configuration. In our experiments, it was cache:il2 dl2, which defines a unified l2 cache.

- cache:il2lat level 2 instruction cache access latency. In our experiments, it was cache:il2lat 6.

- cache:icompress whether or not to convert 64-bit addresses to 32-bit inst equivalents {TRUE|FALSE}. In our experiments, it was specified as cache:icompress FLASE.

- mem:lat memory access latency <first_chunk><inter_chunk>. In our experiments, it is mem:lat 18 2.

- mem:width memory access bus width (in bytes). In our experiments, it is mem:width 64.

- tlb:itlb instruction TLB configuration. In our experiments, it is defined as tlb:itlb itlb:16:4096:4:l. It has 64 entries.

- tlb:dtlb data TLB configuration. In our experiments, it is defined as: -tlb:dtlb dtlb:32:4096:4:l. It has 128 entries.

- tlb:lat inst/data TLB miss latency (in cycles). In our experiments, it was specified as tlb:lat 30.

- res:ialu total number of integer ALUs available. In our experiments, it is defined as res:ialu 4.

- res:imult total number of integer multiplier/dividers available (to CPU). In our experiments, it is defined as res:imult 1.

- res:memport total number of memory system ports available (to CPU). In our experiments, it was: -res:memport 2.

- res:fpalu total number of floating point ALUs available. In our experiments, it was res:fpalu 4.

- res:fpmult total number of floating point multiplier/dividers available. In our experiments, it was res:fpmult 1.

- Write buffer: this configuration is not defined in the original simulator: sim-outorder. It is defined in our enhanced simulator: sim-r10k. In our experiments, dl0 write buffer is defined as: set size 8, line size 32, direct-mapped, least recently used. Dl1 write buffer is defined as: set size 16, line size 32, direct-mapped, least recently used.

Appendix B: Benchmark Input Sets and Execution Scripts

The complete input sets and the corresponding execution script will be given in this section.

For SPEC95 and DIS Benchmark suite, the standard inputs were provided in the product packages, so the inputs were selected from these standard inputs. For example, most of the SPEC95 inputs are the reference inputs. The selection of inputs for DIS Benchmarks is based on the following rules:

- the size of inputs should be large enough so that the miss rate is not too small (larger than 1
- the size of inputs should not be too large such that the simulation can be finished in a reasonable period of time.

In carrying out these two rules, we found that for some DIS benchmarks, such as Raytray, the miss rate is always very small (almost 0%) for inputs of different sizes. For these DIS Benchmarks, we chose the inputs that result in longer (but not too long) simulation time.

For DIS Stressmarks, We tried to choose the inputs from DARPA/ITO. Finally, the inputs for Pointer, Update, Field, Neighborhood, and Transitive_closure are from the inputs provided by DAPAR/ITO. For Matrix, the input in2 is chosen from the inputs provided by us, and m09 is from DAPAR/ITO.

Please note that the input parameters for DIS Stressmarks will be described in detail, while those of SPEC95 and DIS Benchmarks will not. The reason behind this is that the inputs for DIS Stressmarks are not standardized, and readers needs to know the meaning of each parameter. While for SPEC95 and DIS Benchmarks, the inputs have been standardized.Users are expected to use these inputs without any modification.

The configurations used in the following description is along with the simulator sim-r10k and the users' environments. Please contact Weiyu Tang (wtang@ics.uci.edu) for the corresponding configurations.

- Inputs and Scripts for SPEC95

  SPEC95 CINT:

  - 099.go Input: 9stone21.in (ref) Script: sim-r10k [configurations] go.ss < 9stone21.in

- 124.m88ksim Input: ctl.raw (ref) Script: sim-r10k [configurations] m88ksim.ss < ctl.raw

  Note: dcrand.big, dcrand.lit, dhry.big, and dhry.lit should be put into the working directory (where simulator runs). These files can be found in the same directory as ctl.raw.

- 126.gcc Input: gcc.in Script: sim-r10k [configurations] cc1.ss < gcc.in

- 129.compress Input: test.in (tarin) Script: sim-r10k [configurations] compress95.ss < test.in

- 130.li Input: test.lsp (train) Script: sim-r10k [configurations] li.ss < test.lsp

- 132.ijpeg Input: penguin.ppm (ref) Script: sim-r10k [configurations] ijpeg.ss image_file penguin.ppm compression.quality 90 compression.optimize_coding 0 compression.smoothing_factor 90 difference.image 1 difference.x_stride 10 difference.y_stride 10 verbose 1 GO.findoptcomp

- 134.perl Input: primes.in (ref) Script: sim-r10k [configurations] perl.ss primes.pl < primes.in

  Note: dictionary and primes.pl should be put into the working directory (where simulator runs). These files can be found in the same directory as primes.in.

- 147.vortex Input: vortex.in (ref) Script: sim-r10k [configurations] vortex.ss vortex.in

  Note: bendian.rnv (or lendian.rnv), bendian.wnv (or lendian.wnz), persona.1k, vortex.msg, and vortex.raw should be put into the working directory (where simulator runs). These files can be found in the same directory as vortex.in.

SPEC CFP95

- 101.tomcatv Input: tomcatv.in (ref) Script: sim-r10k [configurations] tomcatv.ss < tomcatv.in

  Note: TOMCATV.MODEL should be put into the working directory (where simulator runs). These files can be found in the same directory as tomcatv.in.

- 102.swim Input: swim.in (ref) Script: sim-r10k [configurations] swim.ss < swim.in

- 103.su2cor Input: su2cor.in (ref) Script: sim-r10k [configurations] su2cor.ss < su2cor.in

  Note: SU2COR.MODEL should be put into the working directory (where simulator runs). These files can be found in the same directory as su2cor.in.

200000 1200000 1200010
210000 1300000 1300005
220000 1400000 1400150
230000 1500000 1500200
240000 1600000 1600400
130000 500000 500100
140000 600000 600100
150000 700000 700100
250000 1700000 1703000

Input Description:

Input consists of a single ASCII file containing all the parameters required for a single run. The parameters are listed below in the order they appear in the input file.

* Size of field of values
  This item defines the range of the memory space that can be accessed throughout the whole process. The larger the size of field, the more capacity misses there would be. That is why the miss rates on the second level data cache would be doubled when the size of field of values exceeds 1MB. For the same reason, in the stressmarks, when the size of field of values exceed 1 MB, the miss rates on the second level cache would become very high.
* Size of sample window
  This defines how many elements are to be scanned to search for the median of the values in the sample window. The bigger the size of this sample window, the more capacity misses there would be, and the more comparisons have to be done to search for the median. However, since this sample window size is relatively small compared with the size of field, its influence on the memory performance is small.
* Maximum number of hops to be allowed for each starting value
  It is the number of the new indexes calculated before the current thread stops and the next thread starts. The more the number of hops, the more the computations and memory accesses have to be done.
* Seed for random number generator
* Number of threads
  This defines how many tests are to be done totally. The more the number of threads, the more computations and memory accesses there would be.

The following three items are the same for all the threads.

- 104.hydro2d Input: hydro2d.in (ref) Script: sim-r10k [configurations] hydro2d.ss < hydro2d.in

  Note: put HYDRO2D.MODEL into the current working directory. This file is located in the same directory as hyfro2d.in.

- 110.applu Input: applu.in (ref) Script: sim-r10k [configurations] applu.ss < applu.in
- 125.turb3d Input: turb3d (ref) Script: sim-r10k [configurations] turb3d.ss < turb3d.in
- 141.apsi Input: apsi.in (ref) Script: sim-r10k [configurations] apsi.ss < apsi.in
- 145.fpppp

  Input: natoms.in (ref) Script: sim-r10k [configurations] fpppp.ss < natoms.in

- 146.wave Input: wave5.in (ref) Script: sim-r10k [configurations] wave5.ss < wave5.in

• Inputs and Scripts for DIS Benchmarks

- FFT Input: ft1.in Script: sim-r10k [configurations] FFT.ss < ft1.in
- IU Input: iu1.in Script: sim-r10k [configurations] IU.ss < iu1.in
- DM Input: dm16.in Script: sim-r10k [configurations] DM.ss -i dm16.in
- IM input: balls3.512a.rh (example) Script: sim-r10k [configurations] Imgform.ss 4 6 3.517576e-03 im_balls3_512.rh < balls3.512a.rh
- Raytray Input: balls1.tri (data) Script: sim-r10k [configurations] Raytray.ss ifile balls1.tri itype t nsamp 4096 lookloc 3.05 8 4.8 lookdir -.25 -.707 -.350 ¿ out.rh

• Inputs and Scripts for DIS Stressmarks

- Pointer Stressmark

  Input: p07.in Content:
  4194304 1 330000 -59817 16
  160000 800000 800100
  170000 900000 900100
  180000 1000050 1000100
  100000 200000 200100
  110000 300000 300100
  120000 400000 400100
  190000 1100000 1100100

    * Item number: 3i+6,i =[0,n] The starting index for the ith thread, where 0 $\leq$ i < 16
      This determines for the ith thread where to start sampling. Its influence is small.
    * Item Number: 3i+7, i =[0,n] The minimum ending index for the ith thread. This one, combined with the maximum ending index, defines the memory space where the indexing could not fall into. That is, when the index is greater than or equal to the minimum ending index, and less than the maximum ending index, we exit this thread. Since we try to magnify the number of the memory accesses for each thread, their values are set to the same as the size of the field. In this way, their influence is small.
    * Item Number: 3i+8, i =[0,n] The maximum ending index for the ith thread.

- Update Stressmark

  Input: U08.in Content: 4194304 1 1000000 -59817 190000 1100000 1100100

  Input Description:

  The input of Update Stressmark is similar to Pointer Stressmark except that there is only one thread.

    * Size of field of values
    * Size of sample window
    * Maximum number of hops to be allowed for each starting value
    * Seed for random number generator
    * The starting index
    * The minimum ending index
    * The maximum ending index

- Field Stressmark

  Input: f07.in Content:

  4194304 -59817 17373 32 59 DB 16 00
  B3 F5 86 00
  72 F2 EF 00
  8F 27 50 00
  F7 15 59 00
  3F 70 29 00
  FD B3 C0 00
  F7 7A F5 00
  1B 78 AF 00
  A2 43 03 00

B4 DC D4 00
70 88 CC 00
E6 B8 4D 00
04 55 D8 00
B7 8B DC 00
3C C9 D0 00
D9 7D 71 00
F3 3B 45 00
65 2D A7 00
10 42 22 00
36 8F 6A 00
23 90 1B 00
23 90 1B 00
47 BB 92 00
86 A2 19 00
1C 0A 4E 00
56 31 44 00
2E 98 88 00
01 BD 72 00
76 E5 23 00
08 39 E7 00
E0 AB 69 00
B6 45 A1 00

Input Description:

Input consists of a single ASCII file containing all the parameters required for a single run. These parameters are listed below in the order they appear in the input file.

* Size of field
  This defines the range of the memory space that will be scanned during the whole process of searching the matching patterns of a certain token. So the larger it is, the memory accesses there would be, and the more capacity misses there would be. However, the miss rate can still be low if a lot of scalars are spilled out to memory. Generally, these scalars have high temporary locality.
* Seed for random number generator
* Offset value for token modifier, mod_offset
  This value is the number of words between a found token word and the word that should be used to modify it. Its influence is small when its value

is small. It may incur some misses when its value becomes large.

  * Number of tokens
    The more the number of tokens, the more iterations are to be performed, so the more computations and memory accesses.
  * Item Number: 5+i, The ith token, where $0 \leq i < n$,
    It is given as a zero-terminated string of hexadecimal values. Its pattern and length has less influence on the memory performance because the values of the elements in the field are the randomly generated numbers.

- Matrix Stressmark Input: in2 Content:
  * in2: -2 2000 40000 65535 0.000031
  * m09: -51525 10000 100000 10000 0.0001

  Input Description:

  Input consists of a single ASCII file containing all the parameters required for a single run. These parameters are listed below in the order they appear in the input file.

  * Seed for random number generator
  * The dimension of matrix A and vectors X and b
    The bigger the dimension, the more memory would be required to store the elements in the matrix and vectors. This would result in more memory accesses and thus more data cache misses each time the matrix is multiplied by a vector.
  * The number of nonzero elements to be inserted within matrix A
    In the actual implementation, this number is only the number of non-diagonal nonzero elements. So the total number of nonzero elements is Numbernonzero + dim since the matrix is positive-definite and symmetric.
  * The maximum number of iterations to be performed
    The bigger this number is, the more memory accesses and computations there would be. Note that the actual number of iterations required may be less if the calculated error is lower than the tolerance specified by the next field.
  * The tolerance of error for the solution vector
    The smaller the error tolerance, the more computations and memory accesses it requires approaching this tolerance.

- Neighborhood Stressmark
  Input: n03 Content: -12789 15 500 1000 1 10 1 2
  Input description:

Input consists of a single ASCII file containing all the parameters required for a single run. These parameters are listed below in the order they appear in the input file.

* Seed for random number generator
* The bit-depth of the image
  This value determines how many possible values one pixel can be. The bigger the bit-depth, the more computations are needed to calculate the total of the sum histogram and the total of the difference histogram. It is less dominant when compared with the dimension of the image, the distanceShort and the distanceLong.
* The dimension of the input image
  This is dominant because the lager the dimension, the more memory accesses and computation are to be done.
* The number of line segments to be inserted into the image
  This is non-dominant because the image is accessed independent of the number of the lines.
* The minimum thickness, in pixels, of the line segments, minThickness
* The maximum thickness, in pixels, of the line segments
  These two determines the thickness of a pixel. They don't affect the memory behavior.
* The shorter of the distances between pixels to be measured.
* The longerer of the distances between pixels to be measured.

The distanceShort and distanceLong are dominant. the reason is that each time along one direction, the pixels within the distance of (dimension - distanceShort) or (dimension - distanceLong) are accessed. So the smaller of distanceShort or distanceLong, the more memory accesses and computations are to be done.

– Transitive_closure Stressmark

  Input: tc06.in Content: 512 65536 1550

  Input Description:

  Input consists of a single ASCII file containing all the parameters required for a single run. These parameters are listed below in the order they appear in the input file.

  * Number of vertices
    The effect of the number of vertices on the computations and memory accesses is $O(n^2)$ because n*n matrix is created to keep the record of the shortest paths between two vertices. By using Floyd-Warshall all-pairs

shortest path algorithm, the time complexity is $O(n^3)$. So its influence is significant.

* Number of edges
  its influence is small because no matter how many edges there are, all the paths between all the vertices are still be calculated.
* Seed for random number generator

Appendix C: Number of Instructions of Data Initialization in DIS Stressmarks

As we mentioned in the report, the data initialization parts of DIS Stressmarks occupy a large percentage of memory activities. Sometimes, the data initialization part is two times more than the kernel in the number of memory accesses (i.e., Neighborhood). In order to get the accurate memory behavior of kernels, the activities of data initialization parts should not be taken into calculation. This requires that we fast-forward the instructions in the data initialization parts. For different inputs, the number of instructions that need to be fast-forwarded is different. Here we just give the number of fast-forwarded instructions for the inputs we used in our experiments.

- Pointer: p07.in Fast-forwarded: 2017880238
- Update: u07.in Fast-forwarded: 51186019
- Matrix: No need to fast-forward because the kernel is dominant.
- Neighborhood: n03.in Fast-forwarded: 55599623
- Field: f07.in Fast-forwarded: 492570882
- Transitive_closure: tc03 No need to fast-forward because the kernel is dominant.

	Benchmark	SB	VC	ALS	AFL
Stressmark suite	Pointer				
	Update				
	Field	1	4	3	2
	Matrix	1	4	3	2
	Neighbor	1	3	3	2
	Transitive	1	2	4	3
	Median	1	4	3	2
DIS suite	Benchmark	SB	VC	ALS	AFL
	DM	3	4	1	2
	FFT	1	3	4	2
	IU	1	4	2	3
	IM	1	3	3	2
	Median	1	4/3		2
SPEC95 suite	Benchmark	SB	VC	ALS	AFL
	Applu	3	4	2	1
	Apsi	1	2	3	4
	Compress	1	4	2	3
	Hydro2d	1	4	3	2
	Li	3	4	2	1
	Su2cor	1	4	2	3
	Swim	1	4	2	3
	Tomcatv	1	3	4	3
	Wave5	2	1	3	4
	M88ksim	1	3	4	2
	Perl	1	2	3	4
	Vortex	3	1	2	4
	Median	1	4	2	3
Median		SB	VC	ALS	AFL
		1	4	3	2

Table 1: Qualitative comparison of Adaptation Performance