A Quantitative Evaluation of Adaptive Memory Hierarchy

Haitao Du*
hdu@ece.uci.edu

Paolo D'Alberto
Rajesh Gupta
Alexandru Nicolau
Alexander Veidenbaun†
{*paolo,rgupta,nicolau,alexv*} *@ics.uci.edu*

University of California at Irvine ‡

Abstract

As the gap between CPU speed and memory latency time increases, memory access becomes the dominant factor of any application performance. We consider the *memory hierarchy engineering*, i.e. a mechanisms set to adapt the hierarchy memory parameters to the application needs, to overcome the performance bottleneck of memory access. We evaluate four memory adaptations according to their performance across different benchmark suites: Victim Cache, Stream Buffer, Adaptive Line Size cache, Adaptive fetch line size cache. We investigate the approaches by studying their organizations and, quantitatively, by their ability to reduce certain types of misses (conflict, compulsory and capacity misses).

We demonstrate that these approaches are in general very effective over a large spectrum of applications. For 28 benchmarks, Stream Buffer has an average miss ratio reduction of 52.3% while Adaptive line size has 32.5%, adaptive fetch line size cache has 36.6% and Victim Cache has 14.4%.

As a complementary work, we propose a *miss discrimination model* to help the memory adaptations evaluation and we present an application for which there is miss reduction but the miss ratio is not able to represent it.

---

*Department of Electrical and Computer Engineering, The Henry Samueli School of Engineering, University of California at Irvine, 444F Engineering Tower, Irvine, CA 92697-2625 USA; email:*hdu@ece.uci.edu*

†P. D'Alberto, R. Gupta, A. Nicolau and A. Veidenbaun, Department of Information and Computer Science Zot Code 3425, 444CS Irvine CA 92697-3425; email:{*paolo,rgupta,nicolau,alexv*}*@ics.uci.edu*

‡This work has been supported by DARPA/ITO under contract DABT63-98-C-0045.

Categories and Subject Descriptors: B.3.2 Design Styles - Cache memories; B.3.3 Performance Analysis and Design Aids - Simulation; D.4.2 Storage Management - Allocation/deallocation strategies; C.4 Performance of Systems - Measurement techniques - Modeling techniques - Performance attributes - Reliability, availability, and serviceability; A.1 Introductory and survey.
General Term: Caches, performance evaluation.
Additional Keys Words and Phrases: Adaptive memory hierarchy evaluation, metrics, miss reduction and latency hiding.

1 Introduction

As the gap between CPU speed and memory speed rapidly increases, a large percentage of application execution time is spent on memory accesses (e.g. [GUPTA 1998] and [GUPTA 2000]). 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* [MUÑOZ 1999]. 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 determine 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.

There are three important research topics related to memory utilization: 1) *asymptotic analysis*, 2) *application engineering* and 3) *architecture engineering*. For the first two branches the application is the subject of the investigations. Indeed, the intrinsic characteristics of the application and the ability to rework the application code are investigated in asymptotic analysis and application engineering, respectively. In both cases, the memory hierarchy architecture is invariable, at least during the execution of the application. We present the result of our investigation on the *architecture engineering*: the parameters of the hierarchy may vary during the execution of the application, while the application is fixed. The tuning of the interaction between hardware and software reorganization is a very hard problem, and we only takle the problem for small kernels. Indeed, some adaptations may be ineffective on an application but, by tuning a notch in the source code, they can be very effective (See Section 5.2.1 a version of SB).

For sake of explanation, we introduce shortly the ideas, main results and references in the above areas, then we focus on architecture engeneering.

1. Asymptotic analysis is a powerful abstraction to represent performance by a closed form, by a formula, function of few parameters describing the problem. We all are familiar with the O notation (or Ω notation for lower bounds) of application performance. It is a simple upper bound to the number of basic operations (e.g. comparisons to sort an array or multiplications to multiply two matrices) and it offers a sharp, short estimation of the execution time. Nowadays and in modern architectures, the access time to retrieve and store data must be taken in account as well as the number of operations. The asymptotic analysis of application complexity must be revisited/enhanced. This problem has been investigated for the last twenty years and still there are open problems.

   The pioneers on the subject are Hong and Kung: they set the basis of the current theory to investigate lower bounds to memory accesses for an application described as DAGs. In [HONG 1981], they formalized the problem for the very first time, as a game, 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 (RAM); second level has constant-latency time and unlimited size (Hard Disk). Algorithms are expressed by *Direct Acyclic Graphs (DAG)*, where only data dependency among operations is represented (independent from the scheduling). The approach has been generalized in [AGGARWAL *et all* 1987] and [SAVAGE 1993] for more complex memory hierarchy models. Recently, upper bounds as well as lower bounds are investigated and some initial results can be found in [BILARDI 2000] and [BILARDI and PESERICO 2000]. In general, applications cannot be described as DAGs without simplifications[1] and also real memory hierarchy have characteristics that do not fulfill completely the models proposed (e.g. limited associativity).

2. Application Engineering is a collection of mechanisms to reorganize the application so that the impact of (data) misses is minimized. Given an application and an architecture, there are different ways where application engineering can be applied. Shortly we describe four of them, from the most architecture-aware approach to the most *oblivious*.

   (a) 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 [PANDA 1999]).

   (b) The installation of an application is adaptive. The installation consists of the determination of the architecture parameters, then the source code is tailored upon the parameters and compiled. In practice, there may be different source codes for different architecture (see [WHALEY, FRIGO 1997])

   (c) 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 [WOLFE 1989, WOLFE 1991]).

   (d) 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 [FRIGO 1999] or [TOLEDO 1997, BILARDI 2001, D'ALBERTO 2000]).

   Adapting the application makes sense because software will be software, malleable and interchangeable, and the memory hierarchy has been always considered fixed. 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. But not surprisingly, no fixed memory system is optimal for every application: different applications can exploit different performance and a single application, with different inputs, might exploit different performance [KOGGE 1996].

[1] i.e. input-dependent-loop, merge sort

The engineering of the memory architecture is the ability to tailor the hardware configuration to the application needs. 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 learned in the last two years of investigations (in particular [TANG 1999, VEIDENBAUM 1999, JI 2000]). 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.

In this paper we adopt the terminology used in papers describing hardware solutions (e.g. [JOUPPI 1998, JOUPPI 1998, STILIADIS 1997]). This means that the first level of cache is the upper level, and any other level are indicated as lower level of caches (memory), laying down below. Other papers consider the memory hierarchy built on the other way around, starting from the first level of cache and on top a second level of cache or memory (e.g [AGGARWAL *et all* 1987]).

Four memory adaptation approaches are investigated in this report: *Stream Buffer (SB)* [JOUPPI 1998], *Victim Cache (VC)* [JOUPPI 1998, STILIADIS 1997], *Adaptive Cache Line Size (ALS)* [TANG 1999] and *Adaptive Fetch Size (AFL)* [TANG 2000]. 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 prefetch vectors, 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 of four memory references, either *PC based* [CHEN 1995] or *Partition based* [ALEXANDER 1996], are monitored. If the stride is predicted as constant, k, SB will prefetch 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 improves performance. A miss happens when the referenced data is in neither the first level of cache nor 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 or memory. It is a very small full associative cache. When an element is evicted from the higher level of cache, it goes into the VC, and then to the lower 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 mechanisms for which the hierarchy structure is fixed. As their names intuitively suggest, the line size changes dynamically in ALS and AFL systems. 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 of the line size is activated. Indeed, spatial locality can be exploited using a larger line size; interferences can be reduced using a smaller cache line size. As result of this policy, at any time there can be different active line sizes. ALS tends to reduce capacity misses 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 is reduced by 36.6% in average.

To validate the performance improvements we use simulation tools to collect statistics on several benchmarks. The benchmarks we choose are *Data Intensive Systems (DIS) Benchmark* [MUÑOZ 1999, MUÑOZ 1998], *Stressmark* [MUÑOZ 2000] and *SPEC95* [SPEC 1995]. DIS Benchmark suite is a collection of five applications. They are mostly scientific computations, e.g. Fast Fourier Transform (FFT) [FRIGO 1997], 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 different aspects of modern architectures, one of them is memory hierarchy. Indeed, modern architectures can obtain good performance on this benchmark suite.

We do not modify any of the applications to improve performance. Even though we could for DIS Stressmarks, we implemented them as close to the specifications as possible.

The experimental results collected are simulation-based. The simulator used is *simple-scalar* [BURGER 1997] enhanced with the adaptive memory modules developed by our group (used also in [TANG 1999, TANG 2000]). We did not use sampling-based-simulation [MARTONOSI 1993] [CROWLEY 1999]. 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. We investigate the capability of the metrics to capture the performance improvements due to adaptations. Indeed, in Section 2.5 we present one of the DIS benchmark (MoM), for which current metrics are not able to report any adaptation effects. In Section 3, we give a detailed background and description of adaptation approaches. In Section 4, we present the complete set of the benchmark suites. In Section 5, we introduce the experimental environment and experimental results. Finally, we give our conclusions in Section 6. For sake of explanation, we report some charts in the Appendix.

2 A Model of Cache Misses

The goal of this paper is not to present a collection of data, telling how good architecture adaptation is. The goal is to interpret the experimental results and have a clear understanding of the performance obtained. We seek also a validation of performance metrics, to build up confidence on the experimental results collected in this work. In this Section, we look for a method/metric to answer very simple questions: when and why an adaptation is effective.

First of all, we observe the most common metric: the data miss ratio. We observe the *power* of this metric in Section 2.2. Then, we observe that the memory adaptations target the reduction of the effects or, complete remotion, of certain type of cache misses. For example prefetching in SB is used to reduce the effect of compulsory and capacity misses. So as first step to validate the set of benchmarks, we measure the distribution of the conflict, capacity and compulsory misses, Section 2.3 and 2.4. By construction, the total number of misses, even of different types, does not say when an adaptation will be effective. However, it does say when an adaptation will not be effective.

Thus we claim that the average miss ratio is not always persuasive as metric. We present an example (MoM) in Section 2.5 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 it. MoM is very representative of the problems we faced during the interpretation of the experimental results.

2.1 Analysis Limitations

As disclaimer, in this Section we do not propose any closed formula or analytical model describing the cache misses for the following reasons. 1) Applications can be really complex, and only the developers may have the chance to determine a simple but effective model (see Section 2.5). 2) Small kernels may have extra memory accesses that are introduced artificially by the compiler (see Section 4.2) and cannot be taken into account by a static model. 3) To obtain a model of cache misses is really challenging sometimes even for a single application and single memory system. In fact, one must exploit the relationships among application, compiler optimizations, static instruction scheduling and dynamic instruction scheduling.

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

2.2 Miss Ratio

The overall miss ratio is the first metric we observe. It 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.

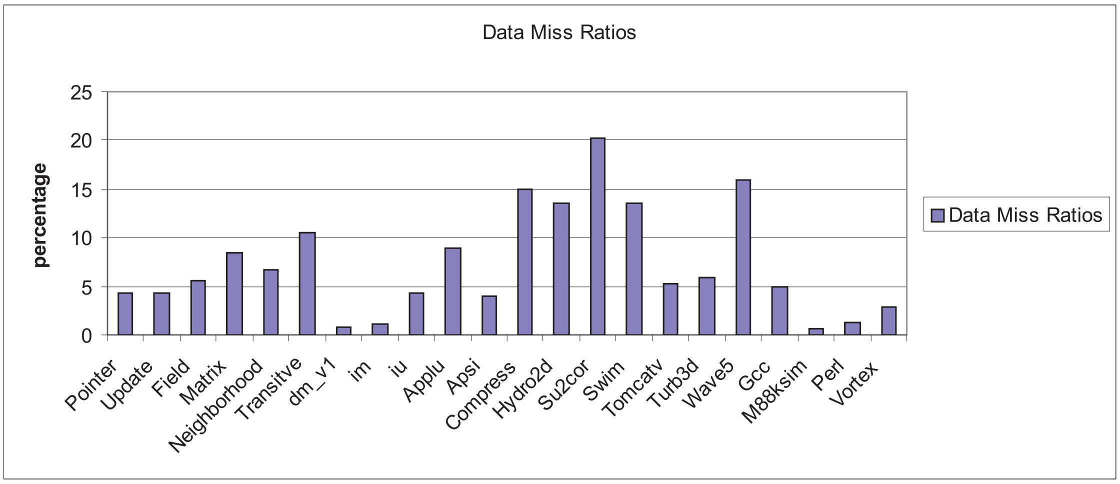


Figure 1: Average Miss Ratios for the following benchmarks: pointer, update, field, matrix, neighborhood, transitive, dm, im (part of raytray), iu, applu, apsi, compress, hydro2d, su2cor, swim, tomcatv, turbo3d, wave5, gcc, m88ksim, perl and vortex.

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 microsystems [CABEZA 1999]. We compile the benchmarks for Ultra 5 architecture and use the predefined cache simulator *cachesim5*. The experimental results are reported in Figure 1.

Reduction of the miss ratio means performance boosting. This can be reflected in the following simple model. It describes the access time of the memory system with one level of cache:

$$\begin{aligned} t_{MemoryTime} &= |Hits| t_{hitLtnc} + |Misses| t_{missLtnc} \\ &= |TotalAccess| t_{hitLtnc} + MissRatio(t_{missLtnc} - t_{hitLtnc}) \end{aligned} \tag{1}$$

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 (such as $t_{hitLtnc}$ and $t_{missLtnc}$). Optimization of applications can reduce the total number of accesses or the miss ratio alone (e.g. tiling in [SONG 1999]). 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.

2.3 Miss Discrimination by Type

However, the performance model presented by Equation 1 does not express the composition of 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 kind of adaptations is effective. In this Section we investigate the quantitative discrimination of misses by type: *Capacity, Compulsory* and *Conflict* misses. 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 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 determine the types of misses by indirect measure, tuning the memory system parameters for different simulations and measurements. We describe our methodology as follows. We measure the number of data misses for the baseline (32KB, 2-way, 32B line), and we indicate it as $M_{BaseLine}$. We measure the number of data misses for the same cache but 32-way associative, and we indicate it as $M_{capacity,compulsory}$; we choose a 32-way associative cache because we think it will be a good approximation of an ideal cache and no conflict misses should be present: only capacity and compulsory. We determine the number of misses when the cache size is 256MB, and we define it as $M_{compulsory}$; we consider that the cache of 256MB is big enough so that there are only compulsory misses, Observation 1 and 3. 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.

Note 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. We compute a negative number for the conflict misses, which is not possible. The problem is that the measures are just approximations: an ideal cache is non-causal and cannot be simulated by the software utilized. The replacement policy adopted is the *Least Recently Used (LRU)* [DAN 1990], and it is not in general the optimal policy for an application since it uses just the information of past accesses. Since we cannot exactly determine the $M_{capacity,compulsory}$, the percentage values for the conflict misses are lower bounds, and, respectively, those for capacity misses are upper bounds.

Figure 2 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.

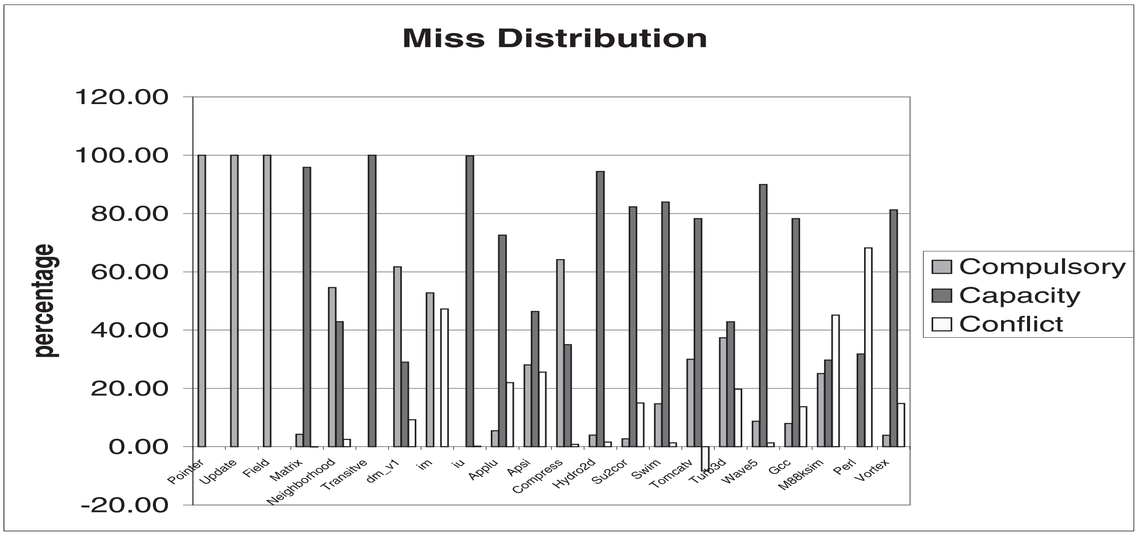


Figure 2: Miss Discrimination in capacity, compulsory and conflict misses in percentage w.r.t. the overall misses of the following benchmarks: pointer, update, field, matrix, neighborhood, transitive, dm, im (part of raytray), iu, applu, apsi, compress, hydro2d, su2cor, swim, tomcatv, turbo3d, wave5, gcc, m88ksim, perl and vortex. From left to right we can find the following bars: compulsory data misses, capacity data misses and conflict data misses

2.4 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 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 the positive cases (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. We present an example where miss distribution

Suite	Benchmark	SB	VC	ALS	AFL
Stressmark	Pointer				
	Update				
	Field	3	4	2	1
	Matrix	2	4	3	1
	Neighbor	3	4	2	1
	Transitive	1	2	4	3
	Median	2	4	3	1
DIS	DM	3	4	1	2
	FFT	1	3	4	2
	IU	1	4	2	3
	IM	1	3	3	2
	Median	1	4/3	1/2/3/4	2
SPEC95	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		1	4	3	2

Table 1: Qualitative comparison of Adaptation Performance: 1 the best performance, 4 the worst performance

over time is important for the evaluation of the miss reduction by adaptation.

2.5 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.

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 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 with an integer number we identify each level in the tree. At level 1 there are the leaves of the tree. The root is at the top and its level is 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 execution following the order 1, 2, 3, 4, 5, 4, 3, 2, 1. The leaves have a head and a tail computation. During the upward phase the head computation at level 1 is the preparation of the inputs, and the tail computation during the downward phase is the formatting of the outputs. We instruct the code so that we can collect statistics by hardware counters on a *R12K* microprocessor [ZAGHA 1996]. 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.

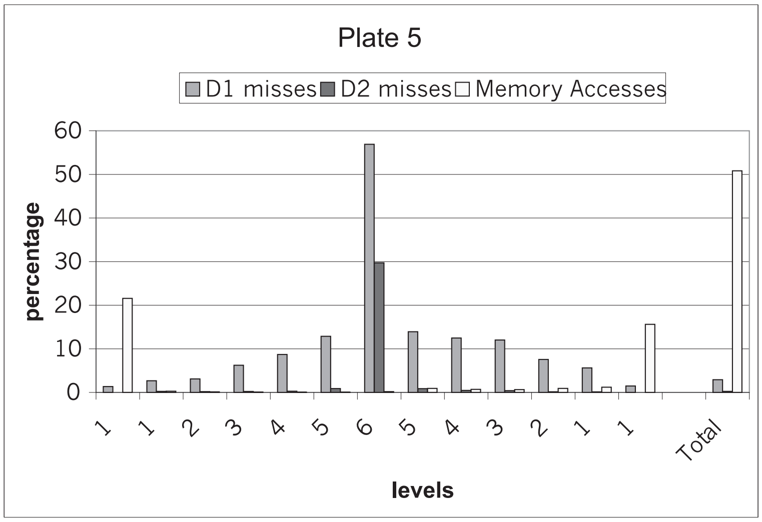


Figure 3: MoM for input Plate 5: Miss Distribution over Different Levels & Average Miss Ratio. The first bar is the data miss ratio for L1 relative to the number of memory accesses in the tree level. The second bar is the data miss ratio for L2 relative to the number of memory accesses in the tree level. The third bar is percentage of memory accesses w.r.t. the total number of instructions. We can see the distribution of misses and the distribution of memory accesses for each level.

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.

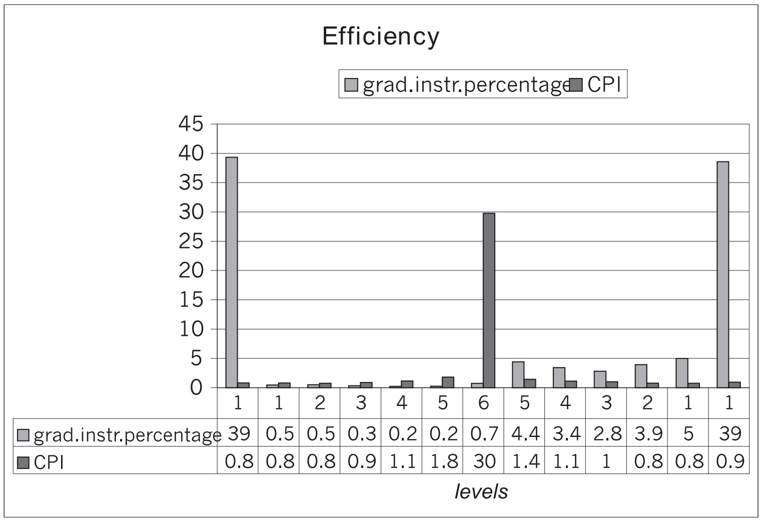


Figure 4: MoM for input Plate 5: Instruction Issue Efficiency. The first bar is the number of graduated instructions w.r.t. the overall number of instructions and the second is the average number of cycles to complete an instruction per level.

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.

3 Background on Adaptations

Four memory adaptation approaches will be described in this Section and they are in order of appearance Stream Buffer (SB) in Section 3.1, Victim Cache (VC) in Section 3.2, Adaptive Line Size Cache (ALS) in Section 3.3 and Adaptive Fetch Size Cache (AFL) in Section 3.4.

3.1 Stream Buffer (SB)

Stream Buffer was first proposed by Jouppi [JOUPPI 1998]. The author proposed a very basic prefetch approach, *prefetch on a miss*, which later was improved in [PALACHARLA 1994]. SB is fully determined by its method to predict what memory references to prefetch and the structure to store and release on-demand the fetched data. Prefetch mechanisms can be classified into Software prefetch and Hardware prefetch. In this paper we focus only on the hardware prefetch but a very brief introduction to the problem in general can be found in Section 3.1.1. In particular we investigate two types of prediction mechanisms: PC Based and Partition Based, Section 3.1.2. In Section 3.1.3 we describe the organization and the prediction mechanisms of SB used to collect experimental results of this paper.

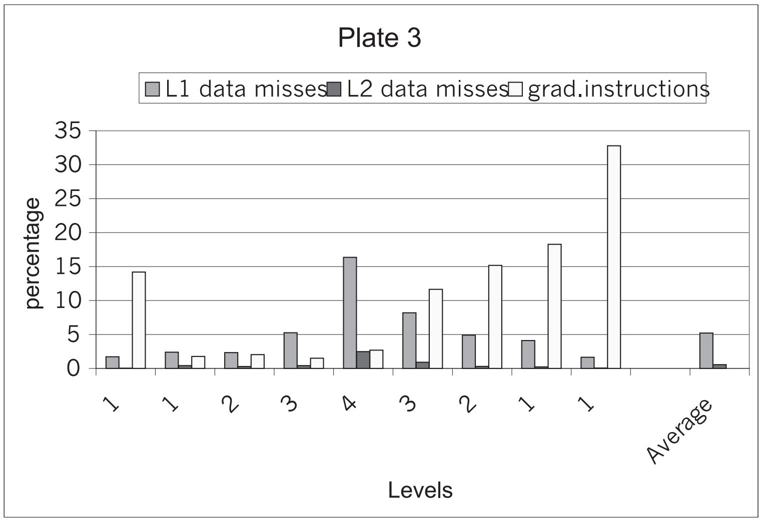


Figure 5: MoM for input Plate 3: Miss Distribution over Different Levels & Average Miss Ratio. The first bar is the data miss ratio for L1 relative to the number of memory accesses in the tree level. The second bar is the data miss ratio for L2 relative to the number of memory accesses in the tree level. The third bar is percentage of memory accesses w.r.t. the total number of instructions. We can see the distribution of misses and the distribution of memory accesses for each level.

3.1.1 Prefetch

Prefetch [VANDERWIEL 2000] is the mechanism that fetches data from the lower-level caches (or memory) before they are actually used. The advantage of Prefetch over the *fetch-on-demand* policy is that it can hide the latency of compulsory and capacity misses. We recall from [HENNESY]: a compulsory miss happens when a cache block is accessed for the first time; a capacity miss happens when a cache block is not in the cache because evicted from it due to insufficient cache space. The code in Figure 7 is an example where compulsory and capacity misses happen. In Figure 7, the linear array A has size N, which is larger than the size of the cache. Suppose that the cache has size 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 Prefetch solution to reduce compulsory and capacity misses: In Figure 8, in order to reduce the latency penalty due to compulsory and capacity misses, prefetching is performed in parallel with processor computations. There are two prefetch instructions in the example. One is outside the loop and one is inside the loop. The one outside prefetches the first C elements of A. The one inside prefetches one element a time, which will be used C iterations later. Instead of waiting for data requests and issuing load instructions, prefetch 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

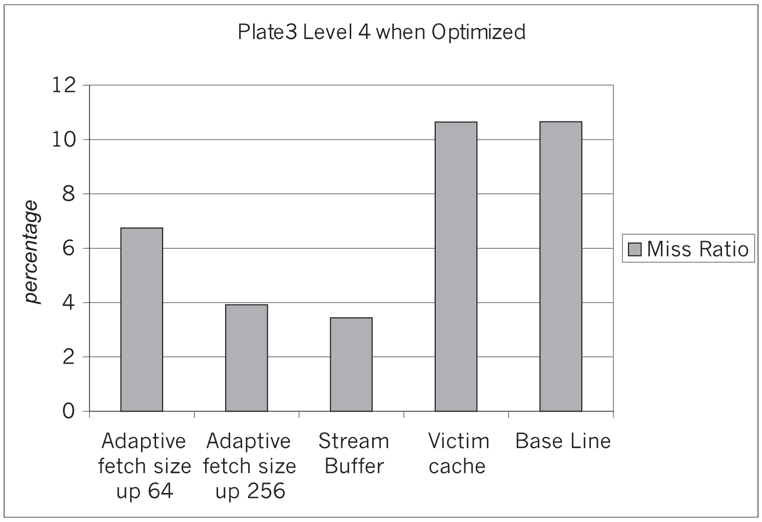


Figure 6: MoM for input Plate 3: Miss Ratios using Different Adaptations. The first bar is data miss ratio when AFL with maximum line size of 64B is applied. The second bar is data miss ratio when the maximum line size is 256B. The third bar is the data miss ratio for SB. The forth bar is the data miss ratio for VC and the last bar is due to the base line.

```
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 with compulsory and capacity misses

data block. By the time data are needed, they are already in the cache and ready to use.

As we mentioned earlier, prefetch commands can be issued either by hardware or by software. We focus on Hardware prefetch, which is described in the following Sections (but the reader interested in software prefetching can start with [BERNSTEIN 1995, MOWRY 1991, CALLAHAN 1991]).

3.1.2 Hardware prefetch

Hardware prefetch 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 prefetch is also called *sequential prefetch*. The simplest approaches of sequential prefetch are based on *one block look ahead* (OBL) [SMITH 1982]. OBL initiates a prefetch for block $b+1$ after block b is accessed. Depending on when to initiate prefetching $b+1$, the implementations of OBL are two:

1. *Prefetch-on-miss*: it initiates a prefetch for block $b+1$ whenever an access to block

```
( 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 introduced to hide latency

b is a miss and block $b+1$ is not in cache.

2. *Tagged prefetch*: it associates a tag bit with every cache block. This bit is set to zero when a block is prefetch. Later, if this block is accessed, the bit is set to one. The zero to one transition triggers the prefetching of the next sequential block $b+1$.

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

One shortcoming associated with sequential prefetch is that prefetch may not be initiated far enough ahead to hide the latency. If prefetching is not completed by the time block $b+1$ is needed, the processor would stall. An approach to solve this problem is to prefetch up to K successive blocks, or a block that is K references ahead, where K can be statically or dynamically [DAHLGREN 1993] determined. But this might result in high traffic from/to the lower-level cache (or memory) [PRZYBYLSKI 1990].

Non-unit-stride prefetching detects the non-unit stride Δ and fetches blocks at Δ units stride away. Notice that if Δ is one, this approach would be sequential prefetch. Based on the special logic used to monitor access patterns, there are two types of non-units stride prefetching: *PC based Prefetch* and *Partition Based Prefetch.*

PC based prefetch [FARKAS 1997, CHEN 1995] is an approach that predicts the stride by comparing the addresses used by successive load or store instructions. For example, three addresses a_1, a_2 and a_3 are used by the same load instruction in three successive iterations. If $a_2 - a_1 = a_3 - a_2$, a stride $\Delta = a_3 - a_2$ is established and the data at address $\dot{a}_4 = a_3 + \Delta$ is prefetched. If $a_2 - a_1 \neq a_3 - a_2$, we remove a_1, and the same process is repeated using a_2, a_3 and a_4. A table (called reference prediction table: RPT in [VANDERWIEL 2000]) 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 prefetch can be defined.

- *Initial state*: an entry is allocated, but no stride is established. The stride is initialized as NULL. The entry contains the operand address a_1 of the most recently executed

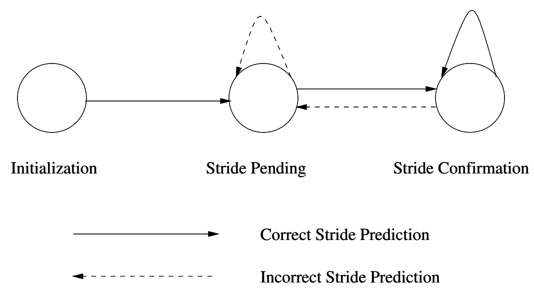


Figure 9: Finite State Machine of PC Based Prefetch

memory instruction. Next time, when the successive address a_2 is referred, a stride $\Delta = a_2 - a_1$ is detected, and the state enters into *Stride Pending State.* A prefetch is issued with the address $\dot{a}_3 = a_2 + \Delta$ 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 a_3. When a_3 comes, we get $\Delta_{new} = a_3 - a_2$. If $\Delta_{new} = \Delta$, the stride is confirmed. The state enters into *Stride Confirmed State.* However, if $\Delta_{new} \neq \Delta$, the entry is updated by the new address a_3, along with the newly detected stride Δ_{new}. The state would stay in *Stride Pending State* .

- *Stride Confirmed state*: a stride Δ_{new} is established. A prefetch for a set of data blocks with distance Δ_{new} is issued. When new address a_4 comes and the new stride $a_4 - a_3 = a_3 - a_2$, the state remains at this state. Otherwise, the state goes back to *Stride Pending State* .

Partition based prefetch can be implemented as consecutive-address based scheme [PALACHARLA 1994], which is used in our experiments. Consecutive-address based scheme is similar to PC based prefetch [FARKAS 1997, CHEN 1995]. 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 chunk, they index the same entry. In practice, the division in chunks 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 prefetched. The mechanism for detecting strides is the same as PC based prefetch, so FSM in Figure 9 can be employed as well. For example, in Figure 10, the strides of 1 are detected for vector A and B, so $A[i+2]$ is prefetched after $A[i+1]$, and $B[i+2]$ is prefetched after $B[i+1]$. PC based prefetch can capture the locality in one reference instruction of different iterations. Partition based prefetch can catch the locality in a group of references. From the mechanism point of view, we cannot say that

```
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

one is better than the other. The example presented in Figure 11 is a case where PC based prefetch outperforms the other. In this example, PC based prefetch can detect the strides for the four different memory references, $V1[i], V1[2i], V1[4i]$ and $V2[i]$. Thus, it issues prefetch commands with appropriate stride for individual reference. Partition based prefetch, 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 prefetch will distinguish it from the reference to $V1$, and prefetch for $V2$ will succeed.

In another example, see Figure 12, Partition based prefetch outperforms PC based prefetch. In this example, the index computations in vector $V1$ are randomized. PC based prefetch 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 prefetching is issued for $V1[i * randq + 96]$.

In our experiments, both PC based prefetch and Partition Based prefetch are employed. The experimental results show that there are (real) 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, implementation

A single-way Stream Buffer [JOUPPI 1998] has one small FIFO buffer to store prefetched data and a hardware prefetch mechanism. On a cache miss and a hit in the SB, the data are sent to cache and removed from FIFO. The prediction mechanism used in Stream Buffer can be either PC or Partition based. It can prefetch data in unit stride or non-unit stride. The experimental results from both prediction mechanisms will be presented in this report. Figure 13 illustrates the memory hierarchy with the Stream Buffer between the Level one

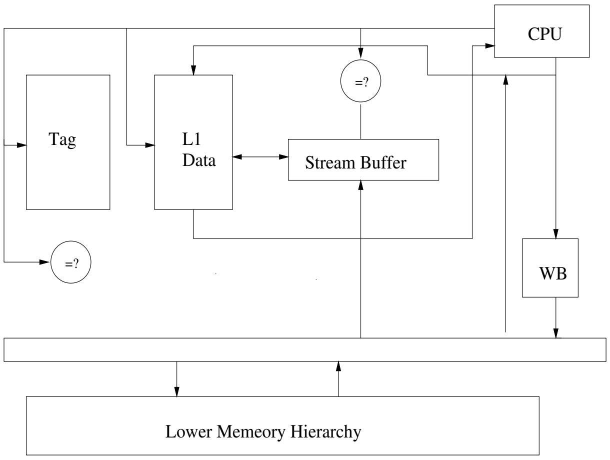


Figure 13: Memory Hierarchy with Stream Buffer

and lower level cache (or memory). The data blocks are prefetched into the FIFO and evicted in order. More specific, once a prefetch is started, up to K cache blocks with stride Δ are loaded, where K is the depth of the FIFO buffer, and Δ is the established stride. Each time data are required, SB is accessed in parallel with data cache. Only the tag of the first line in the FIFO buffer is compared with the tag of the reference address. If there is a miss in cache and a hit in SB, the FIFO buffer provides the data to the cache. The first line is evicted and the following lines shift up. In case that data are not in the first line, the contents in the SB is flushed.

One of the key factors is the depth K of the FIFO. The larger K is in a regular stream, the more future references will be prefetched ahead and this may increase data 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 prefetched data for the previous one. To solve this problem, multi-way Stream Buffer is propose in [JOUPPI 1998], where there are multiple FIFO buffers.

In general, SB is used to remove compulsory and capacity misses in multiple concurrent streams. Obviously, the larger the number of FIFOs buffer, the more concurrent streams can be captured. The optimal number of FIFOs 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: suppose PC-based prefetching, then reference $DIN(j, k)$ evicts $D(j, i)$ and is evicted by $DOUT(j, k)$.

3.2 Victim Cache (VC)

In a memory hierarchy composed of several levels of caches, the Victim Cache [STILIADIS 1997, SCHILLING 2000] 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. For its small size and associativity, VC intends to hold data with temporal locality evicted from the upper level cache due to conflict misses. Figure 15 illustrates the memory hierarchy with VC between the Level one and

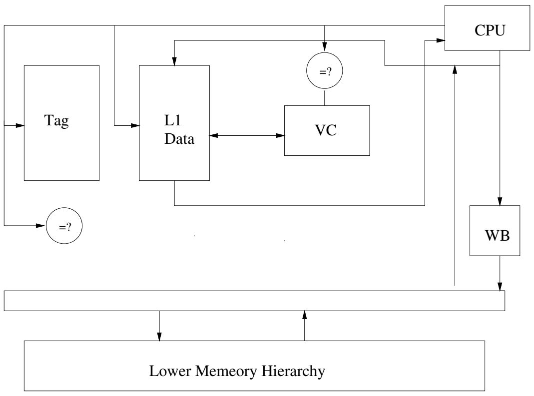


Figure 15: Memory Hierarchy with Victim Cache

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 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.

In this paper, we use a 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 [GONZALEZ 1995, SAULSBURY 1996], the authors present experiments results showing that different applications have different spatial and temporal locality, and also different parts of an application may have such characteristics [VEIDENBAUM 1999]. In this scenario, applications may be unable to take full advantage of the spatial locality offered by a cache with fixed line size. ALS [TANG 1999] is a cache design that uses cache lines of different sizes concurrently. The size of each line changes dynamically on demand of application needs. We introduce some definitions

Definition 1 *ALS cache is composed of same-sized lines, called* physical cache line *(PCL). Each of them is a power of two, from hereafter is 16B.*

Definition 2 *The composition of contiguous PCLs is a* virtual cache line *(VCL), whose size is a power of two.*

Definition 3 *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 4 *Two VLs with starting address $Avcl_1$ and $Avcl_2$ are said to be neighboring if they have the same size, L, and $\lfloor \frac{Avcl_1 - Avcl_2}{2L} \rfloor = 0$ (see [TANG 1999]).*

For a certain VCL or VL, its line size can be changed dynamically during its lifetime in power of two. Line size adjustment is based on the algorithm we will describe shortly. 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. The VL, which is mapped to the VCL, and its neighboring virtual line are determined.

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 maximum line size is limited by the fixed bandwidth between ALS and lower cache (or memory) unless multiple transmission is allowed. Another issue of ALS is the initial line size. Experiments have shown that the selection of initial line size is not important [VEIDENBAUM 1999]. In our experiments, the candidate line sizes are 16B, 32B, 64B, 128B, 256B, and the initial line size is 32B.

3.4 Adaptive Fetch Line Size Cache (AFL)

The AFL [TANG 2000] has the same motivations of 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 the fetch line size is one for all references, and its prediction is based on the global information, thus is not biased to a particular virtual line or reference. Depending on the type of observation used to predict the line size, we distinguish two prediction approaches: *Sampling-based* and *Locality-based*.

The sampling-based approach is able to choose any of the possible line sizes. The optimal fetch size 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. The approach is based on the assumption that the optimal fetch line size for a small interval is very likely to be optimal for a longer interval [TANG 2000]. The choice of the right interval is the result of a compromise. The monitoring and line size adaptation in small intervals permits to exploit the variation of 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 advantage of the sampling-based approach is it takes one interval time to determine the optimal fetch size because all possible fetch sizes are tested.

The Locality-based approach can choose at any time between two line sizes. 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 tendency of the size adjustment 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. They characterize locality. At the end of each interval, $IncCounter$ and $DecCounter$ are compared to each other and 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. Thresholds that will result in the smallest miss rates are different for different applications [TANG 2000]. For example, high $threshold_{dec}$ and low $threshold_{dec}$ will cause a small fetch size, the other way around will cause a large fetch size. Of course, if both thresholds are too high, the adaptation is insensitive to the change of locality. To overcome this sensitivity problem in [TANG 2000] the authors propose an adaptive threshold algorithm, called *aging threshold algorithm*. The thresholds are decreased by an aging ratio as long as there is no fetch size change.

In our experiment, we use the locality-based prediction with aging thresholds. If the line size in an interval is ℓ, the possible fetch size in the following interval is either one of: ℓ, 2ℓ or $\frac{\ell}{2}$. The possible line sizes are 16B, 32B, 64B, 128B, and 256B.

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 (testing the memory hierarchy memory), they are *Data Intensive Systems (DIS) Benchmark* [MUÑOZ 1999, MUÑOZ 1998], *DIS Stressmark* [MUÑOZ 2000] and *SPEC95* [SPEC 1995].

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. It is described in detail in Section 4.1.

DIS Stressmark suite is complementary to the 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 on the 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). It is described in detail in Section 4.2.

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. It is described in detail in Section 4.3.

4.1 DIS Benchmark Suite

DIS benchmark suite [MUÑOZ 1999, MUÑOZ 1998] was developed as representative set 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. It should not be assumed that the "overhead" is negligible. DIS suite is composed of five benchmarks and we introduce each one in the following.

- *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 multipole method (FMM) for fast matrix-vector multiplications. The 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 $O(N \log N)$ and memory requirement is $O(N)$. There is one memory-related bottlenecks: 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. See Section 2.5 for a more detailed description of the benchmark organization.

- *Simulated SAR Ray Tracing* . The algorithm is the simulation of 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. The final step is the process of converting the 2-D array of EM responses into complex images. This involves large data passing and should pose some problems to the 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 I/O instructions up to two times operations (implementation dependent). Thus data starvation may be frequent. The operational and I/O 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. The cost depends on the number of features or targets presented in the input image.

- *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 $O(N \log N)$ operations. 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 attempts, 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). Both index searching and non-index searching require index query and index management. 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 insertion 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 [MUÑOZ 2000] is designed to be complementary to DIS Benchmark suite. Since it is difficult to measure specific elements of interest from large applications, smaller procedures are gathered as DIS Stressmark suite.

DIS Stressmarks are small and focus only on particular elements of a problem. Usually they will loose realistic when representing applications. So DIS Stressmarks should be used in support of DIS Benchmarks, not replacing them.

DIS Stressmarks have a large overhead in data initialization and it should be ignored during simulations: the performance of the kernel would be difficult to measure otherwise. There are seven kernels in DIS Stressmark suite.

- *Pointer*: it repeatedly follows the input-dependent pointers, alias *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 window. Since a window is contiguous, the larger it is, the more spatial locality is exploited. The kernel exploits no spatial locality if the window is one.

- *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 hopping behaviors as Pointer, thus spatial locality is difficult to exploit.

- *Matrix*. It characterizes operations dealing sparse matrices 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 $Ax = b$, where A is a sparse $n \times n$ 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 [EISENSTAT 1977]: a sparse matrix is stored using three vectors. The row elements are continuously stored in one vector, the original column indexes are in another vector and the last vector stores the index to point the first element of each row. Row-wise accesses are faster than column-wise accesses because spatial locality can be exploited. In Matrix-vector-multiplications, vectors are indexed non-contiguously and the hopping behaviors may happen.

- *Neighborhood*. It deals with data that is organized in a two-dimensional grid (image), and computed by neighborhood operators. The operator can be described as follows: given a ray and a distance along the direction of the ray, 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 operators are describe as it follows. 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/ray. 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 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 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*, which is 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 overwrites 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 problems. The procedure utilizes the Floyd-Warshall all-pairs shortest path algorithm [CORMEN]. It accepts as input an adjacency matrix of a directed graph, which is stored in row major format. It then exploits a recursive solution by dynamic programming 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

The benchmarks are developed to be as resistant as possible to compiler optimizations, which might not translate into real world performance gains. The benchmarks have long execution 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, and SPEC CFP95, which includes a set of computing-intensive floating-point benchmarks.

In the following two sections, each benchmark will be described in more details covering the application areas represented, tasks accomplished, and characteristics of the routines (Note that often the description is taken as it is from the benchmark documentation [SPEC 1995]).

4.3.1 SPEC CINT95

It includes a set of eight computing-intensive integer point benchmarks:

- *099.go* is an example that utilizes artificial intelligence in game playing. It plays the game of *go* against itself. The benchmark is a stripped down version of a successful computer program. There is a great deal of pattern matching and look-ahead logic. It is very common 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 file 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 from DEC.

- *132.ijpeg* represents an image processing application. It performs JPEG image compression with various parameters. First, an original bitmap image (usually GIF, 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 implementation of *jpeg* routines 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 $90 \sim 98\%$ vectorizable.

- *102.swim* is a 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.

- *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 applications. 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.

Platform: our experiments are performed on Sun Ultrasparc 5.

Baseline Simulator: the baseline simulation is a processor simulator *Sim-Outorder*, which is from SimpleScalar [BURGER 1997]. In turn, the SimpleScalar architecture is derived from the MIPS-IV ISA [PRICE 1995] with small modifications to the semantics. It supports non-blocking cache and speculative execution. Users can statically configure the architecture by the following flags:

- **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.
- **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.

Adaptive Cache Modules: two modules are introduced on top of the baseline simulator: Victim Cache and Stream Buffers. Two modules are introduced in place of the baseline simulator: Adaptive Line Size cache model and 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 prefetch 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 when a miss happens. 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 will take place next time when 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 the set of 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 with aging ratio 0.01, lower bound of threshold 0.4, middle threshold 0.55.

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.

Baseline Architecture: the baseline architecture for level one data cache is: cache size 32KB; block size 32B (line size); associativity 2-way; least recently used replacement policy

5.2 Example Benchmark Analysis

The analysis of the benchmark improvements is very important to our evaluation: it helps to understand how adaptations exploit the spatial and temporal locality of the application.

Three benchmarks are chosen for evaluation in this section. Each one illustrates one adaptation approach. The stressmark *Transitive Closure* illustrates PC based Stream

Buffer (SB-PC). The SPEC95 benchmark *wave5* illustrates VC, and *applu*, always from SPEC95, illustrates ALS and AFL.

5.2.1 Transitive Closure: Why Stream Buffer Works

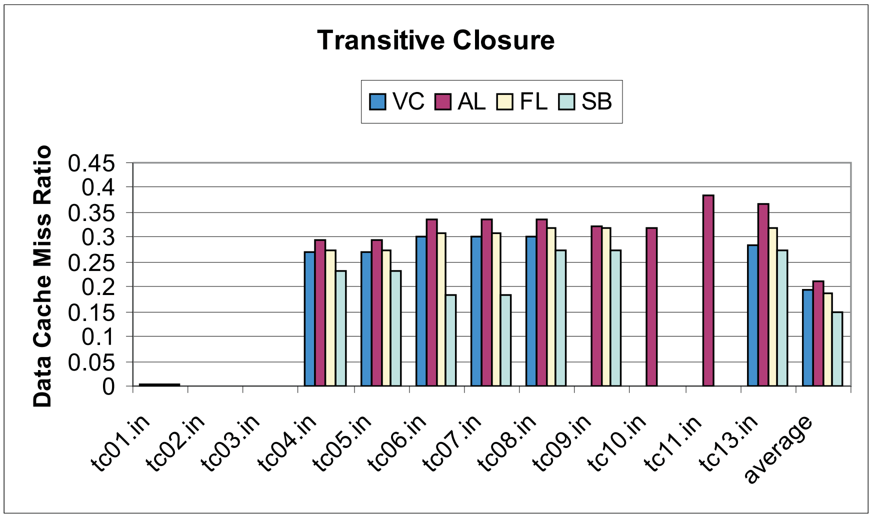


Figure 16: Average Miss Ratios of Adaptations on Transitive Closure

As we can see from Figure 16 SB (SB-PC) has the best performance over all adaptations. The reason must be found in the implementation of the algorithm. The Transitive Closure is the Floyd-Warshall algorithm using the adjacent matrix stored in row major format. The majow flaw is that the implementation does not exploit the layout and update the matrix by row (instead of by column). Adaptation as AFL and ALS cannot be applied successfully.

While SB-PC improves *Transitive Closure* (15%), SB-PA does not (0.13%). We report the kernel in Figure 17. DIN is a $N \times N$ matrix stored in row major format, so is $DOUT$. Using SB-PA, the first two references: $DIN(j, i)$ and $DIN(j, k)$ (FORTRAN notation),

```
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 17: Kernel of Transitive_Closure

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. 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. In contrast, SB-PC recognizes the accesses by row and detects the strides. SB-PC is able to hide the latency, but it reads more than it should. It fetches a whole line. If the number of columns of the adjacent matrix is larger that the cache, the cache will use just one element at any time.

5.2.2 Wave5: Why Victim Cache Works

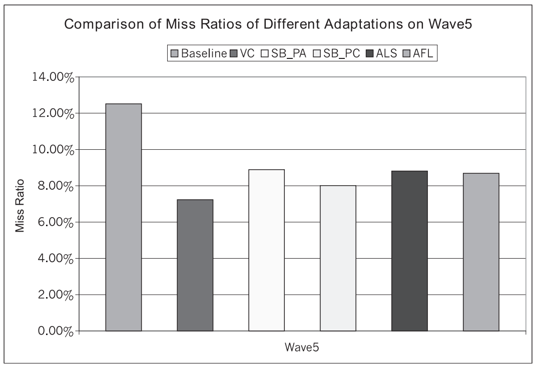


Figure 18: Average Miss Ratios of Adaptations on wave5 with reference input. From left to right we can find the following bars: the data miss ratio for the baseline, VC, SB-PA, SB-PC, ALS and AFL

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. By construction, the VC removes conflicts, therefore at least 42% misses are conflict misses. This is counter-intuitive with respect to the miss distribution achieved in Section 2.3, where conflict misses are a small fraction of the total misses. It may happen for the following two reasons. 1) 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. 2) We recall that the conflict misses measured in Section 2.3 are underestimated.

We did profiling to determine the relationships among procedure calls and their frequencies. We assume that the procedure with the longest execution time causes the largest 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 : 1) four vectors with high temporal locality have unitary stride accesses; 2) some vectors have the hopping characteristic access, i.e. A[B[j]].

For most of the time, more than two vectors are accessed concurrently. So conflicts may happen frequently in a 2-way associative cache. Hopping characteristics may cause more conflicts (but with a random behavior). VC fits the job because it can capture conflict misses from vector accesses with high temporal locality.

5.2.3 Applu: Why Adaptive Line Size and Fetch Line Size Work

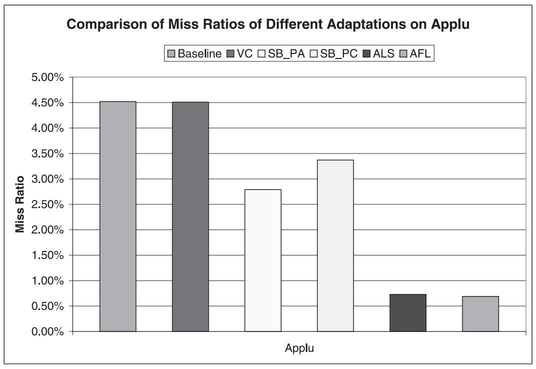


Figure 19: Average Miss Ratios of Adaptations on applu with reference input. From left to right we can find the following bars: the data miss ratio for the baseline, VC, SB-PA, SB-PC, ALS and AFL

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, see Figure 5.2.3. 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: 1) there are unitary stride accesses; 2) there are seven streams in the same loop; each of them has a constant stride; four of them refer to the same vector and the other three refer to different vectors.

ALS and AFL are able to exploit the spatial locality for both types of accesses. SB-PC has only 4 ways of prefetch buffers, which is insufficient for seven streams. Notice that SB-PA is a little bit more effective than SB-PC: indeed, it recognizes and allocates buffers only for the last three reference streams on different vectors, and the number of prefetch 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 ex-

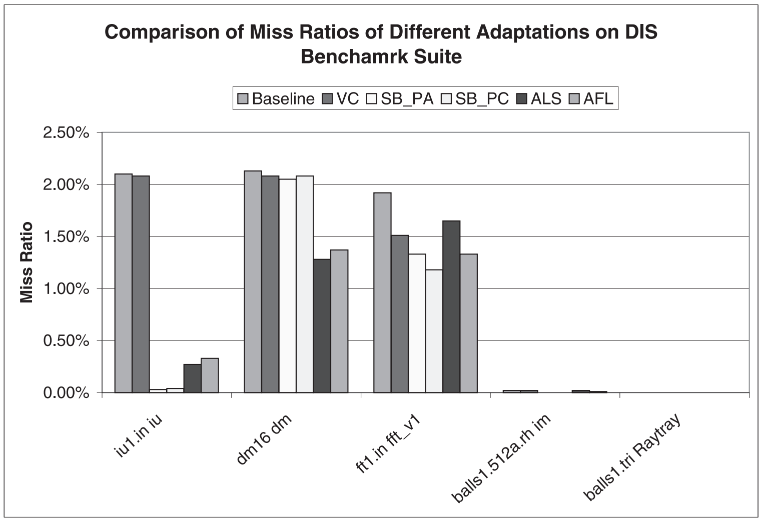


Figure 20: Average Miss Ratios of Adaptations on DIS Benchmark Suite. The benchmarks are in the following order: IU, DM, FFT, IM Imageform and Ray Tray (both in Ray tray). From left to right we can find the following bars: the data miss ratio for the baseline, VC, SB-PA, SB-PC, ALS and AFL

ample to show that adaptation is very effective in short periods of the application execution time, but is undetectable by the average miss ratio.

DIS Benchmarks have small miss ratios because they have been developed to fit modern 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 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.

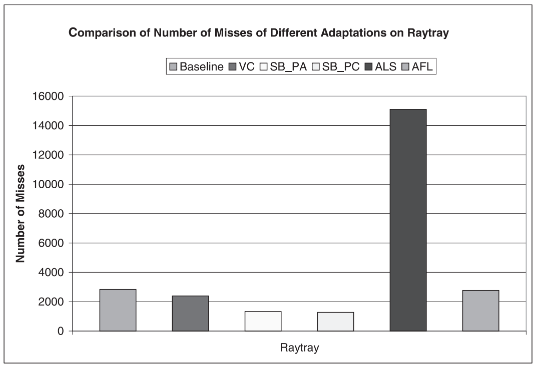


Figure 21: Average Miss Ratios of adaptations on Raytray, magnification. From left to right we can find the following bars: the data miss ratio for the baseline, VC, SB-PA, SB-PC, ALS and AFL

SB is effective for *IU, FFT,* and *IM.* SB-PA and SB-PC have similar performance. This suggests that the references access separate memory chunks and have constant strides (furthermore due to their similar performance we do not discriminate SB-PA from SB-PC, we use SB-PC as representative and we identify it simply by SB). Almost every 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 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* (2% improvement). We infer from the experimental results (comparing with ALS and AFL) that two possible situations exist in *DM*: 1) there is no regular reference streams; 2) there are too many concurrent reference streams.

For Raytray, SB hides the latency of 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 somehow effective for *FFT* but interference does not allow better performance. ALS and AFL are not effective for *Raytray*. ALS introduces misses (12 times the baseline).

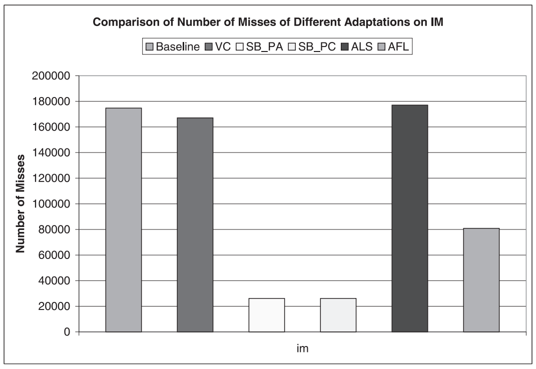


Figure 22: Average Miss Ratios of adaptations on IM (Image form part of RAY TRAY) magnification. From left to right we can find the following bars: the data miss ratio for the baseline, VC, SB-PA, SB-PC, ALS and AFL

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 fully effective. The kind of conflict in this benchmark is independent from the line size. In *IU* and *DM*, VC reduces the miss ratios by 0.95% and 2% respectively.

In DIS Benchmark suite, we have observed the following:

- SB is the most effective among the three types 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 (taking out *Raytray* from the average). 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, e.g. a large number of reference streams and irregular memory access patterns.

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

5.3.2 Experimental Results for DIS Stressmark Suite

Memory adaptations improve four out of six DIS Stressmarks. They are not effective for *Pointer* and *Update*, for which, every memory access is random. Of course, random memory accesses cannot be optimized, and therefore we will not discuss them in our analysis. But it is worthwhile to observe that, even though the kernels have very high miss ratios, they achieve only 27% data cache miss. This is because our compiler (`gcc`) generates an executable that has *spills*. Since spills are generally hit in cache, the miss rate of the kernel drops.

In this section, our focus is on the Stressmarks that have space for improvements. We illustrate the miss ratios in six different charts, one for each benchmark, in Figure 23.

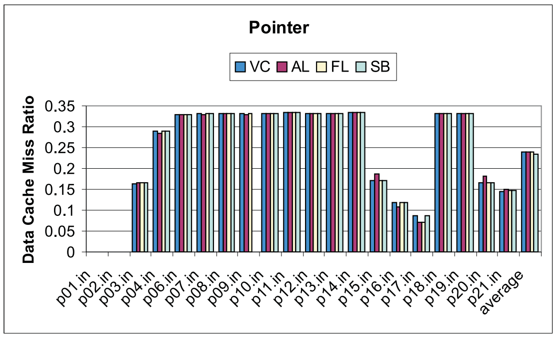
Note that data initialization of the DIS Stressmarks is not negligible and we tested the stressmark kernels separately.

The simulation is done in two steps. 1) Using a fast simulation it is computed the number of instructions to initialize the inputs. 2) A detailed simulation is fast forward, and 3 billions instructions are simulated. The baseline is not reported in Figure 23. The reader may use the miss ratio of VC as baseline: most of the stressmarks do not have interference misses for which the VC is effective. Therefore the miss ratio of VC is the miss ratio of the base line.

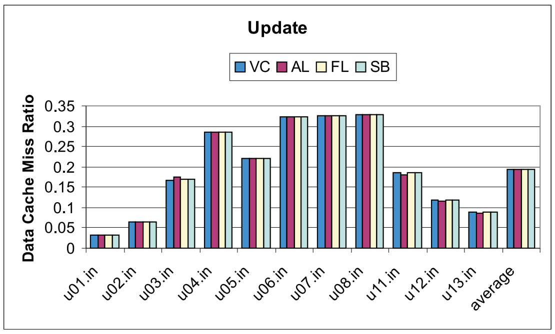
As previously adopted, we use the notation "SB" to refer both prefetch schemes (PA and PC). SB hides the latency of 30% of the total misses for *Matrix* and around 50% for *Neighborhood*. It hides nearly all the miss latencies in *Field* (75%). SB is effective because most memory accesses have constant 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. The average miss ratio reduction is 50% (we did not consider *Pointer* and *Update*).

ALS and AFL are effective for *Matrix*, in which they have better performance than SB. They are also very effective for *Field* because of its spatial locality. They are effective for *Neighborhood*. ALS and AFL are not effective for *Transitive Closure* . In fact, there is no spatial locality due to the row access of the adjacent matrix.

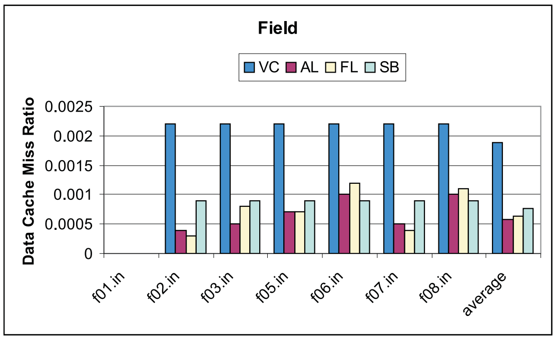
VC is not effective for all the stressmarks. For *Neighborhood*, VC does not have any hit. For *Transitive Closure*, VC has a hit ratio of 1/10, we use an input for which eight columns of the adjacent matrix can fit in the cache and conflict misses do not happen. *Field* has no hit.



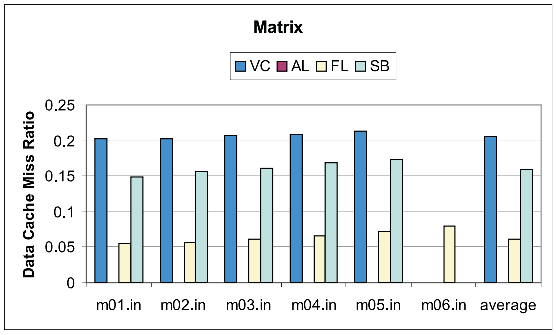
1)



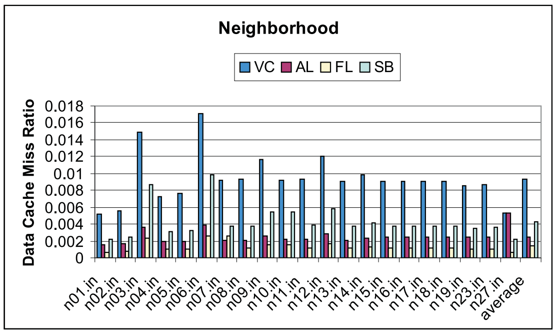
2)



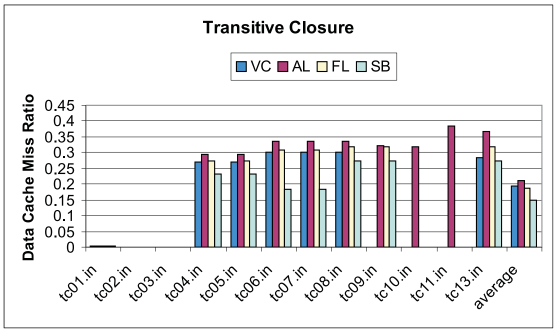
3)



4)



5)



6)

Figure 23: 1) Pointer Stressmark 2) Update Stressmark 3) Field Stressmark 4) Matrix Stressmark 5) Neighborhood Stressmark 6) Transitive Closure Stressmark. When one bar is not present (but other are present) it is because it is missing simulation results. Note for Matrix AL could not be simulated. The average measure, last bars on the right, is computed on the intersections of the input tests for which all adaptations have simulation results.

5.3.3 Experimental Results for SPEC95

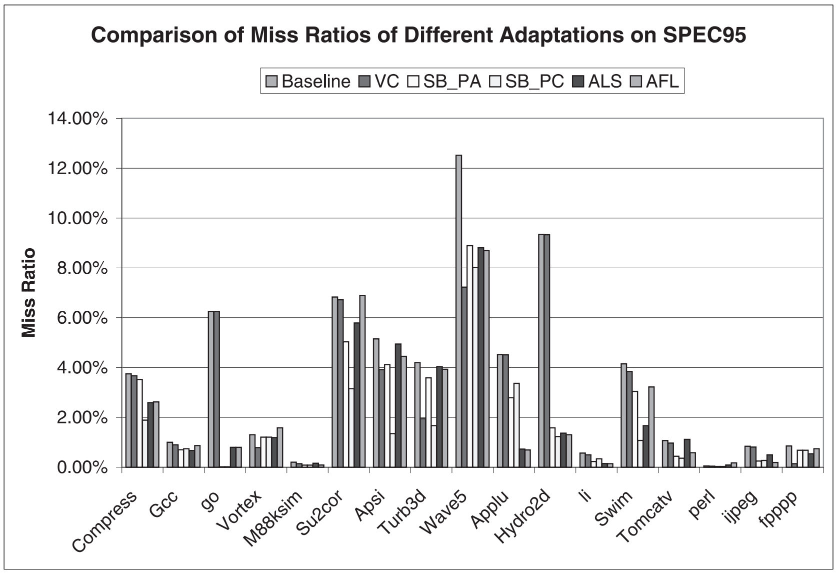


Figure 24: Average Miss Ratios of Adaptations on SPEC95. The benchmarks are in the following order: compress, gcc, go, vortex, m88ksim, su2cor, apsi, turbo3d, wave5, applu, hydro2d, li, swim, tomcatv, perl, ijpeg and fpppp. From left to right we can find the following bars: the data miss ratio for the baseline, VC, SB-PA, SB-PC, ALS and AFL

We simulated 17 out of 18 SPEC95 benchmarks with the reference inputs. The average miss ratio is shown in Figure 24. 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. *Swim* is a good example to explain why: there are references that belong to different memory chunks but the sequence of loads referring to the same chunk does not exploit a constant stride (even though the references have constant strides).

For *applu* and *li*, SB-PA works better than SB-PC. We use *applu* as example previously in Section 5.2.3. We recall very briefly why: there are more than four reference streams; 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 before, we use SB-PC as representative and we refer it by SB. 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 improvements.

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 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 *Vortex* 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. 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.

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%.

5.4 An Alternative Metric

We conclude the Section with a final evaluation of the same assists but from a very different prospective (it may be just a curiosity): data traffic and address bus activation.

The data traffic (in bytes) to the second level of cache, or memory, is a good metric for the estimation of energy. It offers a descriptive but simple enumeration of how many time the data bus has been toggled and therefore an average measure of the energy spent to feed the first level of cache (this is true for cache with low associativity, see [KAMBLE 1997] for a model of energy dissipation in modern caches).

The address bus activation is a good metric for the estimation of the number of fetches towards the lower levels, but also an estimation of the energy to drive the address bus as well.

These measures can be used to estimate the (energy) cost of the performance achieved by an assistant. If we consider the VC as baseline for our consideration, any improvement in performance with the positive effect of cost reduction is very interesting. In Figure 27 we can see the data traffic and in Figure 26 (see Appendix) the address bus activations for some of the benchmarks presented in this paper.

We expected the SB behavior, indeed, SB increases the data traffic by increasing the number of fetches. ALS and AFL have the very nice property to decrease data traffic and number of fetches: this means they exploit locality and utilization of data at first level of cache.

Extra hardware is introduced to assists adaptation in ALS and AFL, therefore a complete model should comprise some cost estimations for them. Without a complete energy model (energy per hit and miss) any assumptions or comparisons on the energy costs and savings are premature. Nonetheless, we think that a simplified version of AFL without hardware-driven adaptation but with compiler driven adaptations instead, will have the pleasant characteristic to improve performance and energy consumption at the same time.

6 Conclusions

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 cache (AFL). The average miss ratio reduction is shown is Figure 25. Based on the experiments 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 are higher than baseline because of unsuccessful prediction. AFL can improve 25 benchmarks. VC is effective only for 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 ones. However, they have some problems in predicting the optimal line size at run time, for no regular applications. Indeed, the prediction is based on the information collected in a interval and on the assumption that the behavior will be the same in the next interval. Such as assumption is not always apply (see also in [VAN VLEET 1999]).

But there are applications for which the line size can statically be predicted at compile time (e.g. SWIM). No hassle to predict the line size, no extra hardware should be active during the execution. The performance stability of the cache line adaptivity can be

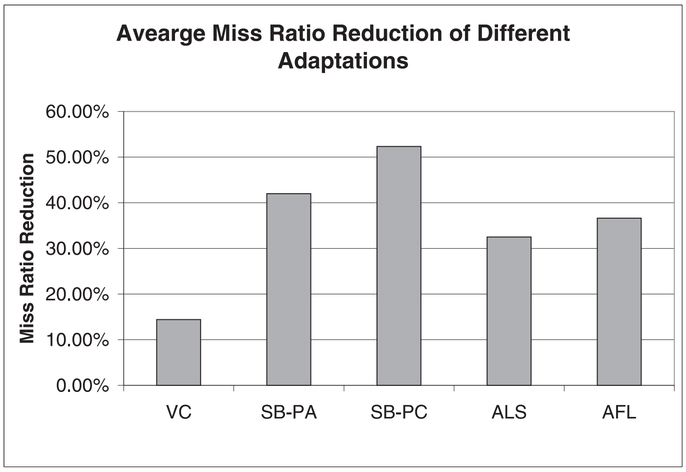


Figure 25: Average Miss Ratio Reduction of Different Adaptations over all benchmarks. From left to right we can find the following bars: the data miss ratio for the baseline, VC, SB-PA, SB-PC, ALS and AFL

exploited by the compiler, analyzing the application and optimizing the line size without any reference to the cache latency, making the code portable (without modifications) on different architecture but with same memory hierarchy features.

For some benchmarks (e.g. DIS Benchmarks), we have found that the average miss ratio was not able to express the efficiency of adaptation. The adaptation may be effective in part of the application, but cannot 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), even though there is no improvement for the average behavior.

7 Thanks and Acknowledge

This work is supported by DARPA/ITO under contract DABT63-98-C-0045. The authors thank the members of AMRM project for their help and useful suggestions to improve the earlier draft of this paper. A particular grateful thank goes to Weiyu Tang for his advises on how to interpret the simulation results. The authors thank also Sanjay Velamparambil for the effort to disclose the *inner beauty* of MoM and collect experimental results.

References

[AGGARWAL 1987] AGGARWAL, A., CHANDRA, A.K., and SNIR, M. Hierarchical memory with block transfer. *In 28th Annual Symposium on Foundations of Computer Science.* (Los Angeles, California, 12-14 October 1987), 204-216

[ALEXANDER 1996] ALEXANDER, T., and KEDEM, G. Distributed prefetch-buffer/cache design for high performance memory systems. *Proceedings of 2nd IEEE Symposium on High-performance Computer Architecture.* (1996, IEEE Press, Piscataway, NJ), 254-263.

[AGGARWAL *et all* 1987] AGGARWAL, A., ALPERN, B., CHANDRA, A.K., and SNIR, M. A model for hierarchical memory. *Proceedings of 19th Annual ACM Symposium on the Theory of Computing.* (New York, 1987), 305-314.

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

[BILARDI and PESERICO 2000] BILARDI, G., and PESERICO, E. A characterization of temporal locality and its portability across memory hierarchies. *ICALP 2001, International Colloquium on Automata, Languages, and Programming.* (Crete, July 2001).

[BILARDI 2000] BILARDI, G., PIETRACAPRINA, A., and D'ALBERTO, P. On the space and access complexity of computation DAGs. *26th Workshop on Graph-Theoretic Concepts in Computer Science.* (Konstanz, Germany, June 2000).

[BILARDI 2001] BILARDI, G., D'ALBERTO, P. and NICOLAU, A. Fractal matrix multiplication: a case study on portability of cache performance. *Workshop on Algorithm Engineering 2001.* (Aarhus, Denmark, 2001).

[BURGER 1997] BURGER, D.C., and AUSTIN, T.M. The simplescalar tool set, version 2.0. *Computer Architecture News*, 25 (3), (June 1997) pp. 13-25, (Extended version appears as UW Computer Sciences Technical Report No.1342, June 1997).

[CABEZA 1999] CABEZA, M.L.C., CLEMENTE, M.I.G., and RUBIO, M.L. Cachesim: a cache simulator for teaching memory hierarchy behavior. *Proceedings of the 4th annual Sigcse/Sigue on Innovation and Technology in Computer Science education.* (1999), 181.

[CALLAHAN 1991] CALLAHAN, D., KENNEDY, K., and PORTERFIELD, A. Software prefetching. *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS-IV).* (1991), 40 - 52.

[CHIOU 2000] CHIOU, D., JAIN, P., and RUDOLPH, L. 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).

[CORMEN] CORMEN, T.H., LEISERSON, C.E., and RIVEST, R.L. *Introduction to Algorithms*, MIT Press.

[CHEN 1995] CHEN, T.F., and BAER, J.L. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers.* 44, 5 (May 1995), 609-623.

[CROWLEY 1999] CROWLEY, P., and BAER, J.L. 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).

[DAHLGREN 1993] DAHLGREN, F., DUBOIS, M., and STENSTROM, P. Fixed and adaptive sequential prefetching in shared-memory multiprocessor. In *Proceedings of the International Conference on Parallel Processing.* (St. Charles, IL, 1993), 56-63.

[D'ALBERTO 2000] D'ALBERTO, P., BILARDI, G., and NICOLAU, A. Fractal LU-decomposition with partial pivoting. *ICS Technical Report TR# 00-22.*

[DAN 1990] DAN, A., and TOWSLEY, D. An approximate analysis of the lru and FIFO buffer replacement schemes. *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems.* (1990), 143-152.

[DONGARRA] DONGARRA, J.J., DUFF,I.S., SORANSEN, D.C. and VAN DER VORST, H.A. *Numerical Linear Algebra for Performance Computers.* SIAM Asymptotic Analysis (ed).

[DU 2001] DU, H., D'ALBERTO, P. and GUPTA, R. Memory adaptation techniques: a unified overview across benchmark suites. ICS Technical Report #01-41. (August 13, 2001).

[EISENSTAT 1977] EISENSTAT, S.C., GURSKY, M.C., SCHULTZ, M.H., and SHERMAN, A.H. Yale sparse matrix package. *Yale University Department of Computer Science Technical Report.* Volumes 112 and 114, 1977.

[FARKAS 1997] FARKAS, K., CHOW, P., JOUPPI, N., and VRANESIC, Z. Memory-system design considerations for dynamically-scheduled processors. *Proceeding of the 24th Annual International Symposium on Computer Architecture (ISCA).* (June 1997).

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

[FRIGO 1999] FRIGO, M., LEISERSON, C.E., PROKOP, H., and RAMACHANDRAN, S. Cache oblivious algorithms. *Proceedings 40th Annual Symposium on Foundations of Computer Science*, (1999).

[GONZALEZ 1995] GONZALEZ, A., ALIAGAS, C., and VALERO, M. A data cache with multiple caching strategies tuned to different types of locality. *Intl. Conference on Supercomputing.* (7 April 1995), 338-347.

[GUPTA 2000] GUPTA, R. Architectural adaptation in AMRM machines. *Proceedings of IEEE Computer Society Workshop on VLSI 2000* . (Los Alamitos, CA, USA), p75-79.

[GUPTA 1998] GUPTA, R. 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*

[HONG 1981] HONG, J., and KUNG, T.H.: I/O complexity, the red-blue pebble game. *Proceedings of the 13th Ann. ACM Symposium on Theory of Computing.* (Oct.1981), 326-333.

[HENNESY] HENNESY, J.L., and PATTERSON, D.A. *Computer architecture a quantitative approach.* Morgan Kaufman publishers 2-nd edition.

[JI 2000] JI, X., NICOLAESCU, D., VEIDENBAUM, A., NICOLAU, A., and GUPTA R. Compiler-directed cache assist adaptivity. *ICS Technical Report #00 17.* (May 2000).

[JOHNSON 1998] JOHNSON, T.L., CONNORS, D.A., HWU, W.W. Run-time adaptive cache management. *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, 7, (Kohala Coast, HI, USA 6-9 Jan. 1998), 774-775.

[JOHNSON 1997] JOHNSON, T.L., and HWU, W.W. Run-time adaptive cache hierarchy management via reference analysis. *24th Annual International Symposium on Computer Architecture ISCA'97.* (May 1997), 315-326.

[JOUPPI 1998] JOUPPI, N.P. 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), 388 - 397.

[KAGSTROM 1998] KAGSTROM, B., LING, P. and VAN LOAN, C. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software.* 24, 3. (Sept. 1998), 268-302.

[KAMBLE 1997] KAMBLE, M.B., and GHOSE, K. Analytical energy dissipation models for low-power caches. *Proceedings of the International Symposium on Low Power Electronics and Design.* (1997), 143 - 148.

[KNOWLES 1997] KI, A., KNOWLES, A.E. Adaptive data pre-fetching using cache information. *Proceedings of the International Conference on Supercomputing.* (July 1997, Vienna, Austria) 204-212.

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

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

[MARTONOSI 1993] MARTONOSI, M., GUPTA, A., and ANDERSON, T. Effectiveness of trace sampling for performance debugging tools. *Proceedings 1993 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*. (1993).

[MOWRY 1991] MOWRY, T., and GUPTA, A. Tolerating latency through software-controlled prefetching in shared-memory multiprocessor. *Parallel Distrib. Comput.* 12, 2. (June, 1991), 87-106.

[MIPSR10K] MIPS R10000 Microprocessor. User Manual Version 2.

[MUÑOZ 1998] MUÑOZ, J.D. Presentation at data-intensive systems principal investigators meeting. (1 October, 1998).
http://www.darpa.mil/ito/research/pdf_files/dis_approved.pdf .

[MUÑOZ 1999] MUÑOZ, J.D. Data-intensive systems benchmark suite analysis and specification. (Submitted by Atlantic Aerospace Electronic Corp to DARPA/ITO, 30 June 1999).
http://www.aaec.com/projectweb/dis/DIS_Benchmarks_v1.pdf

[MUÑOZ 2000] MUÑOZ, J.D. DIS Stressmark Suite: Specification for the Stressmarks of the DIS Benchmark Project Version 1.0. (Submitted by Atlantic Aerospace Division, Titan System Corp. to DARPA/ITO, 24 August, 2000).
http://www.aaec.com/projectweb/dis/DIS_Stressmarks_v1_0.pdf

[PALACHARLA 1994] PALACHARLA, S., KESSLER, R. E. Evaluating stream buffer as a secondary cache replacement. *Intl. Symposium on Computer Architecture*. (May 1994) 24-33.

[PANDA 1999] PANDA, P.R., NAKAMURA, H., DUTT, N.D. and NICOLAU, A. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*. 48, 2. (Feb. 1999),142-9.

[PRICE 1995] PRICE, C. MIPS IV instruction set, revision 3.1. *MIPS Technologies, INC.*. (Mountain View, CA, January 1995).

[PRZYBYLSKI 1990] PRZYBYLSKI, S. The performance impact of block sizes and fetch strategies. In *Proceedings of the 17th International Symposium on Computer Architecture*. (Seattle, WA 1990), 160-169.

[SAULSBURY 1996] SAULSBURY, S., PONG, F. and NOWATZYK, A. Missing the memory wall: the case for processor/memory integration. *Intl. Symposium on Computer Architecture.* (1996), 90-101.

[SAVAGE 1993] SAVAGE, J.E. Space-time tradeoff in memory hierarchies. *Technical report.* (Oct 19, 1993).

[SCHILLING 1999] SCHILLING, W.W., ALAM, M. The impact of pre-fetching and victim caching on computer systems performance. *Proceedings of the ISCA 12th International Conference (ISCA).* (1999), 271-276.

[SCHILLING 2000] SCHILLING, W.W., ALAM, M. Performance simulation of the combination of pre-fetching and victim caching. *Proceedings of the ISCA 15th International Conference Computers and Their Application (ISCA).* (2000), 284-287.

[SMITH 1982] SMITH, A.J. Cache memories. *ACM Computing Surveys.* 14, 3. (Sept, 1982), 473-530.

[SONG 1999] SONG, Y., and LI, Z. New tiling techniques to improve cache temporal locality. *Proceedings of the ACM SIGPLAN Conference on Programming language Design and implementation.* (1999), 215-228.

[SPEC 1995] *http://www.spec.org/osg/cpu95/*

[STILIADIS 1997] STILIADIS, D., and VARMA, A. Selective victim caching: a method to improve the performance of direct-mapped caches. *IEEE Transactions on Computers,* 46, 5. (May 1997), 603-610.

[TANG 1999] TANG, W., VEIDENBAUM, A., NICOLAU, A., and GUPTA, R. Adaptive line size cache. *UC, Irvine, Technical Report ICS-TR-99-56.* (Nov. 1999).

[TANG 2000] TANG, W., VEIDENBAUM, A., NICOLAU, A. and GUPTA, R. Cache with adaptive fetch size. *UC, Irvine, Technical Report ICS-TR-00-16.* (April 2000).

[TOLEDO 1997] TOLEDO, S. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.* 18, 4. (Oct. 1997), 1065-1081, .

[VANDERWIEL 2000] VANDERWIEL, S.P., and LILJA, D.J. Data prefetch mechanisms. ACM Computing Surveys. 32, 2. (June 2000).

[VEIDENBAUM 1999] VEIDENBAUM, A.V., TANG, W., GUPTA, R., NICOLAU, A., and JI, X. Adaptive cache line size to application behavior. *Proceedings of International Conference on Supercomputing (ICS).* (June 1999), 145-154.

[VAN VLEET 1999] VAN VLEET, P., ANDERSON, E., BROWN, L., BAER, J., and KARLIN, A. Pursuing the performance potential of dynamic cache line sizes. *Proceedings of the International Conference on Computer Design (ICCS'99).* (October 1999).

[WHALEY] WHALEY, R.C., and DONGARRA, J.J. Automatically tuned linear algebra software.
http://www.netlib.org/atlas/index.html

[WOLFE 1989] WOLFE, M. More iteration space tiling. *Proceedings of Supercomputing.* (Nov.1989), 655-665.

[WOLFE 1991] WOLFE, M., and LAM, M. 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).

[ZAGHA 1996] ZAGHA, M., LARSON, B., TURNER,S., and ITZKOWITZ, M. Performance analysis using the mips r10000 performance counters. *Proceedings Supercomputing 1996.* (Nov. 96, Pittsburgh, PA).
http://www.sgi.com/processors/r10k/timing/perfcount.html

[ZUCKER 2000] ZUCKER, D.F., LEE, R.B., FLYNN, M.J. Hardware and software cache pre-fetching techniques for mpeg benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology.* 10, 5. (Aug. 2000), 782-796.

Appendix: Figures

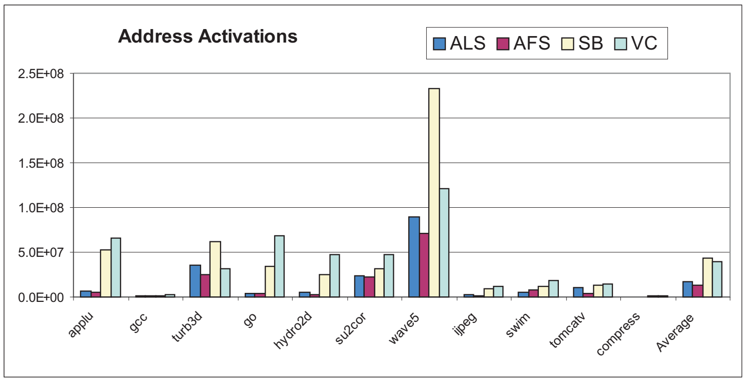


Figure 26: Total number of times the address bus towards L2 or memory is activated. From left to right we can find the following bars: ALS, AFL, SB and VC

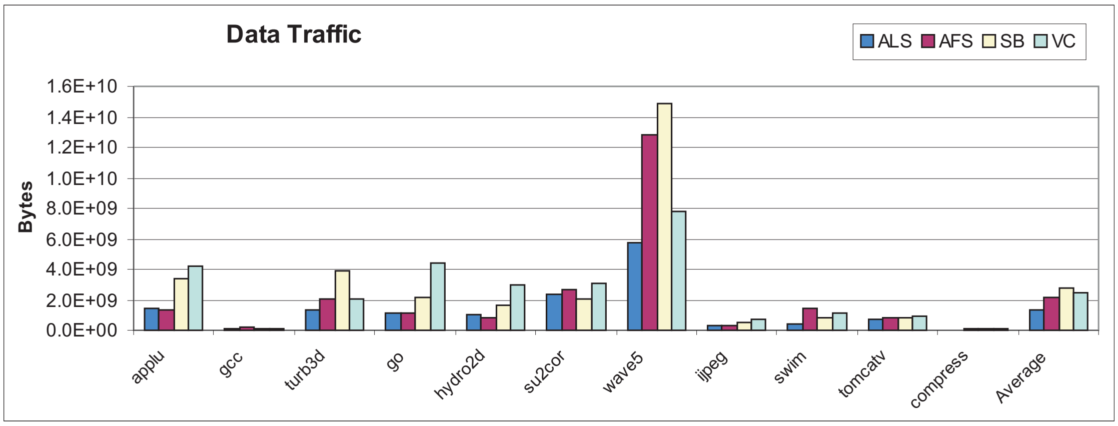


Figure 27: Total number of bytes transmitted on the data bus to and from L2 or memory From left to right we can find the following bars: ALS, AFL, SB and VC