Adaptive Line Size Cache [1]

Weiyu Tang Alexander Veidenbaum
Alexandru Nicolau Rajesh Gupta

ICS
Technical Report

Technical Report #-99-56
Nov. 1999

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

Department of Information and Computer Science
University of California, Irvine

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

Abstract

Current cache designs support only fixed line size. Fixed cache line size limits cache's ability in spatial/temporal locality utilization. In this report, we present a cache design with adaptive line size, where an individual cache line can change size dynamically based on the spatial locality of the miss fetched data. For this cache, the spatial locality of a line of data in the cache can be detected by its access pattern during its stay in the cache. Then on replacement from the cache, the optimal line size for a line of data is determined by the detected spatial locality and is recorded in the memory. Overall, a better performance is achieved by this novel cache design. Simulations of SPEC95 benchmarks show that this cache design can significantly reduce cache miss rates. A 2-level adaptive line size (ALS) cache can achieve an average speedup of 1.7 over a 2-level fixed line size (FLS)cache. A 128k direct-mapped L2 ALS cache outperforms a 512K 2-way set-associative L2 FLS cache
.

Contents

List of Figures

List of Tables

1 Introduction

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

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

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

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

An alternative approach is to allow a design parameter to take on a range of values and provide a mechanism for changing towards a more optimal parameter value dynamically during execution. The same approach can also be applied to an algorithm or a policy used by hardware. A general term "adaptivity" will be used to refer to this dynamic approach. Adaptivity can potentially allow each application to approach much closer to an optimal architecture/hardware configuration and thus an optimal performance. It can also allow the system resources to be better utilized and shared within and across applications.

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

- *Adaptive routing* pioneered by ARPANET in computer networks and, more recently, applied to multiprocessor interconnection networks [3, 5] to avoid congestion and route messages

faster to their destination;

- *Adaptive traffic throttling* for interconnection networks [5]. [15] shows that "optimal" limit varies and suggests admitting messages into the network adaptively based on current network behavior;

- *Adaptive cache control* or coherence protocol choices were proposed and investigated in the FLASH and JUMP-1 projects [10, 13];

- *Adapting branch history length* in branch predictors was proposed in [11] since optimal history length was shown to vary significantly among programs;

- *Adaptive adjustment of data prefetch length* in hardware was shown to be advantageous [4], while in [7] the prefetch lookahead distance was adjusted dynamically either purely in hardware or with compiler assistance. [12] is another version of selective prefetch, closest to our work in many ways but also quite different.

Much of the previous work mentioned above addressed a specific problem via adaptivity although not always via adaptive hardware. Adaptivity has received a lot of attention recently with a drastic increase in VLSI complexity and transistor count as well as advances in reconfigurable logic. The research presented here addresses the use of automatic, dynamic hardware adaptivity in the design of data cache. It is a part of a more general effort, the Adaptive Memory Reconfiguration and Management Project (AMRM) at the University of California-Irvine, to apply adaptivity to the design of a memory hierarchy. More information about AMRM can be found in [2, 17].

There are several possible cache parameters that one can dynamically adapt. They include cache size, line size, write policy, write buffering, prefetching, etc. Some of these are only adaptable in theory. For instance, the cache size is largely determined by technology parameters and desired latency and can only be adaptively decreased. This does not make a lot of sense, except possibly as a mechanism to reduce its latency and, as a result, increase the processor clock rate which the cache largely determines, as proposed in [1]. Write policy can be switched between write-through and write-back, in fact the Intel Pentium^{TM} architecture [9] already allows this on a per-page basis but not automatically. However, the parameter that is likely to deliver a significant performance improvement while being feasible to implement adaptively is the cache line size. This paper introduces a cache design with a hardware-adaptive line size. To our knowledge, such an organization has not been previously explored.

Previous research [6, 12] has shown that different applications exhibit different spatial/temporal localities. We have shown in [17] that different parts of an application also exhibit different spatial/temporal localities. It will be ideal if a cache line size can be set per application or the size can be changed dynamically for different parts of an application.

In reality, most processors support only one cache line size. MIPS [14] supports multiple cache line sizes, but line size is configurable at boot time and all the lines are of same sizes. [8] also investigates the advantage of dynamic cache line size change. Their approach relies on profiling or other compiler transformations to find optimal cache line size. And the line size information is conveyed to the processor at run time by special instructions.

The rest of the report is organized as follows. Section 2 describes the experimental environment for cache simulations. Section 3 shows the problems with **f**ixed **l**ine **s**ize (FLS) cache. Section 4 presents our model for line size adaptation. Section 5 shows how to form an algorithm for line size adaptation. Section 6 explores 48 adaptive algorithms to find an optimal one for further investigation. Section 7 shows the performance of **a**daptive **l**ine **s**ize (ALS) cache. Section 8 analyzes the tradeoffs between ALS cache and FLS cache. Section 9 presents the conclusion and discussions for future work.

2 Experimental Setup

2.1 Environment

The benchmarks are compiled on an SGI system for an R3000 processor using MIPS and MIPSPro compilers with the following flags: -n32 (MIPS-III instruction set, 32b executable) and -O2 . They are used in the execution-driven cache and memory simulation of the architecture described in this report. The cache simulator is invoked and driven via MINT-3 [16], which models a single-issue, statically-scheduled processor.

A system architecture used in this study consists of a processor, L1 cache, L2 cache and memory. Both caches block on a cache miss. Every instruction takes one cycle except memory instructions that cause cache miss. The L1 miss penalty is 5 cycles and the L2 miss penalty is 100 cycles.

We have studied both a direct-mapped cache and a 2-way set-associative cache. The cache uses write-back policy. Given current processor implementation, we have studied L1 cache with size 16KB, 32KB and 64KB. L1 line size ranges from 8 to 256 bytes. L2 cache with size 128KB, 256KB and 512KB are studied. L2 line size ranges from 64B to 512B.

Primary performance metrics used in this report are: cache miss rate, data traffic volume between L1 and L2, and speedup of ALS cache over FLS cache.

2.2 Benchmarks

We use two sets of benchmarks in our experiments. The first set of benchmarks is used in the algorithm exploration to determine how adaptive algorithms work. The benchmark description and some execution statistics are shown in Table 1. It consists of selected SPEC92 and SPEC95 integer and floating-point benchmarks and an additional floating-point benchmark–ARC3D. ARC3D is chosen because it has a significant fraction of memory accesses with long strides.

The second set of benchmarks is used to evaluate the impact of ALS cache on miss rate, traffic and processor speedup. This set of benchmarks includes all SPEC95 benchmarks except GCC and VORTEX. These benchmarks exhibit a sufficiently varied memory behavior to thoroughly check our architecture. The benchmark statistics are shown in Table 2. All the benchmarks are used in the study of L1 cache. Only first 8 benchmarks, which have significant L2 cache miss rate, are used in the study of L2 cache and processor speedup with a 2-level cache architecture.

Program Name	Input	Instr (M)	Memory (M)
GCC	stmt.i	88	31
SC	loada1	862	128
LI	li-input.lsp	1014	536
FPPPP	natoms	1335	672
ARC3D	-	64	24
APSI	apsi.in	1472	500
IJPEG	specmun.ppm	1542	500
PERL	scrabbl.pl	1384	500
WAVE	wave5.in	1686	500

Table 1: Algorithm Exploration Benchmarks

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

Table 2: Performance Evaluation Benchmarks

3 Lack of "Optimal" Cache Line Size

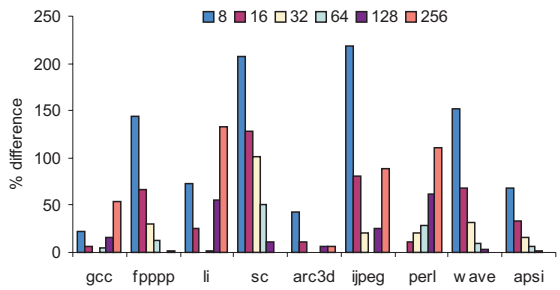

Figure 1: Normalized miss rate difference Δ_M

It has been shown in the past that the optimal line size, which produces the minimal miss rate, varies from benchmark to benchmark. To demonstrate this for our benchmark suite and support the claimed need for adaptivity, a 16K cache with a fixed line size ranging from 8B to 256B is simulated. Similar results for SPEC95 have been shown in [8]. For each benchmark, the "optimal" miss rate is determined and a normalized miss rate difference Δ_M is computed for all possible line sizes as:

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

The normalized miss rate difference for all benchmarks is shown in Figure 1. A missing bar indicates a 0% difference and corresponds to the optimal line size.

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

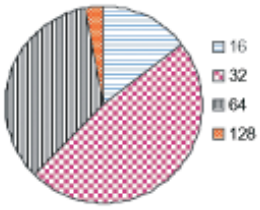

Figure 2: Distribution of optimal line size in "time" for GCC

Next, let us consider the intra-benchmark cache behavior. In each interval of 100K memory references, the line size resulting in the minimal miss rate is determined. Figure 2 shows the distribution of the optimal line sizes in the "time-domain" for GCC. No single "optimal" line size

exists for GCC. Line size 16B, 32B and 64B is the optimal for 15%, 50% and 30% respectively. Even 128B is optimal for a fraction of the time. Results for other benchmarks show similar cache behavior and strongly support the need for adaptivity within a benchmark.

4 Cache Model for Line Size Adaptation

In a FLS cache, one cache line is fetched from lower memory hierarchy on a cache miss. In order to adapt to changing spatial and temporal localities, we fetch one variable sized line on a cache miss. We use the following definitions for a ALS cache:

- physical cache line (PCL)

  Similar to a FLS cache, an ALS cache is divided into same-sized lines. Such a line is called a PCL. Each line has a tag with several additional bits: valid bit, use bit, etc. Its size pcl_size is a power of 2 and is the smallest fetchable unit. A PCL can be represented as:

  $$(line\_num, addr)$$

  where $line\_num$ is the order of one PCL in cache and $addr$ is the starting memory address for a line of data in a PCL. $addr$ is aligned on the boundary of pcl_size.

- virtual line (VL)

  A VL is a line of data in the lower memory hierarchy and is an unit for miss-fetch. The VL size vl_size is a power of 2 and the minimal value is equal to pcl_size. A VL can be represented as:

  $$(addr, vl\_size)$$

  where $addr$ is the starting memory address for a line of data in a VL and $addr$ is aligned on the boundary of vl_size.

- virtual cache line (VCL)

  A VCL consists of multiple continuous PCLs. On a cache miss, one VL from the lower memory hierarchy will fill one VCL. A VCL can be represented as:

  $$(line\_num, addr, num\_lines).$$

- creation of a VCL

  A VCL is created when a VL is fetched into the cache.

- destruction of a VCL

  A VCL is destroyed by a newly created VCL that shares at least one PCL with it.

- life time of a VL

  The life time is the period between the creation of a new VCL by this VL and the destruction of that VCL.

Cache lookup in an ALS cache is the same as that in a FLS cache. Cache hit time, which is on one of the critical paths in processor designs, is not affected.

In an ALS cache, a line can have multiple sizes and the smallest size is equal to *pcl_size*. To be flexible in locality utilization, *pcl_size* should be the smallest possible, such as 8 bytes. But a small *pcl_size* results in higher tag store cost. When hardware cost is an issue, the flexibility can be sacrificed. Even if *pcl_size* is equal to the line size of a FLS cache, an ALS cache still has the advantage in utilizing more spatial locality.

An ALS needs the following hardware support:

- Additional fields for each PCL
  - *useBit*

    The field size is 1 bit. It is used to monitor the usage of a PCL. It is set to 0 when a VCL that this PCL is part of is created; it is set to 1 when the PCL is accessed.
  - *neighborBit*

    The field size is 1 bit. It is used to determine the "neighboring" relationship amony VLs. It is set to 1 if this PCL is the first PCL of a VCL and the "neighboring" VLs are already in cache; it is set to 0 otherwise.
  - *sizeBits*

    The field size is $Log_2 N$ bits, where N is the possible number of VCL size. This field stores the encoding of the size of the VCL that this PCL is part of.
- Memory modification
  - *startBit*

    This 1-bit field is added to every PCL-sized data block. Value 1 means that the block is a VL; value 0 means that the block is part of a VL.

Note that the ALS cache design incurs bandwidth overhead. A FLS cache only needs to write back data when a cache line is dirty. In case of an ALS cache, if line size is predicted to increase or decrease, line size change information needs to be transferred to the lower memory hierarchy even if the cache line is clean. We have included this overhead in the experiments for the rest of the report.

5 Algorithm Design

5.1 General Algorithm

Algorithm *memory_access* in Figure 3 shows the operations in a memory access. First, the PCL that an address is mapped to is checked to determine whether the address hits in the cache. On

Input : $addr$

1. Find the physical cache line pcl that $addr$ is mapped to
2. If tag for $addr$ and tag for pcl match then
3. Set dirty bit for a write access
4. Set $useBit$ of pcl to 1
5. Else
6. $miss\_fetch(addr)$
7. Endif

Figure 3: Algorithm : $memory\_access$

a cache hit, bits used in a traditional cache, such as dirty bit, are set. Then bits specific to an ALS cache such as $useBit$ are set to for future line size prediction. On a cache miss, algorithm $miss\_fetch$ is invoked. It is obvious from the above that cache hit time is not affected in an ALS cache.

Algorithm $miss\_fetch$ in Figure 4 describes operations on a cache miss. In a FLS cache, only one line will be replaced. In an ALS cache, there are following complications:

- The number of VCLs to be replaced is variable and the size of replaced VCLs may be different.

  From line 1 to 3, the miss-fetched VL is first brought into a buffer close to the cache. Then the VCL to be created by this VL and the VCLs to be destroyed are determined.

- Line size for the VLs in the replaced VCLs needs to be predicted;

  From line 4 to 10, for a VL in a VCL to be replaced, it is first sent to a write buffer if it is dirty. Then algorithm $line\_size\_predict$ is invoked to predict the next line size of this VL.

- Adaptivity-related bits need to be updated.

  In line 12, adaptivity-related bits for the newly created VCL are updated for future line size prediction by algorithm $line\_size\_predict$.

Figure 5 shows the algorithm to predict the next line size of a VL. Vl_size should be set according to the predicted spatial locality in a VL. The following rules are used in locality prediction:

- Good spatial locality prediction

  For two VCLs vcl_1 and vcl_2 created by VLs vl_1 and vl_2, if the following conditions are satisfied:

Input : $addr$

1. Find virtual line $vl = (start\_addr, size)$ that $addr$ is part of
2. Fetch vl to a buffer close to cache
3. Calculate virtual cache line $vcl = (line\_num, num\_lines, start\_addr)$ to be created by vl
4. For every virtual cache line vcl_i to be destroyed by vcl Do:
5. Get virtual line vl_i in vcl_i
6. Send vl_i to write buffer if vcl_i is dirty
7. $line\_size\_predict(vcl_i)$
8. Send line size change request of vl_i to lower memory hierarchy when the change is predicted
9. Replace whole or part of vcl_i depending on cache replacement policy used
10. Endfor
11. Bring vl from the buffer to vcl in cache
12. Set useBit and sizeBits for all the physical cache lines in vcl

Figure 4: Algorithm: $miss\_fetch$

  - there exists a "neighboring" relationship between vl_1 and vl_2;
  - vcl_1 and vcl_2 have shared life time;

  then we predict good spatial locality between vl_1 and vl_2. A larger VL can be formed by merging vl_1 and vl_2.

- Poor spatial locality prediction

  Suppose a VCL vcl is created by a VL vl. When the vcl is to be destroyed, if either the first half or the second half of PCLs in vcl are not used, then we predict poor spatial locality in vl. The vcl can be divided into smaller VLs.

Correct "good spatial locality prediction" can eliminate the number of capacity misses. The "neighboring" referred above will be further discussed in Section 5. Correct "poor spatial locality prediction" can eliminate the number of conflict misses and increase cache utilization. Suppose two VLs of same size conflict. In the first VL, only the first half of it is used. In the second VL, only the second half of it is used. Conflict misses between these VLs can be eliminated by

Input: virtual cache line *vcl*
Output: one of the following line size change prediction–increase, decrease, unchange

1. Get VL *vl* that is mapped to *vcl*
2. Set *neighborBits* for *vcl* and its neighboring VCLs they have the same tag
3. If *neighborBit* for *vcl* is set then
4. Predict line size increase for *vl*
5. Else if no *useBit* is set in PCLs in either the first half or the second half of *vcl*
6. Predict line size decrease for *vl*
7. Else
8. Predict line size unchange for *vl*
9. Endif

Figure 5: Algorithm: *line_size_predict*

partitioning both lines into small lines of half size. Cache utilization is increased because more number VLs can coexist in the cache.

In line 2, if the VLs in "neighboring" VCLs have same tags, these VLs can merge into a large VL. Hence spatial locality exists among these VLs.

5.2 Variables in Adaptive Algorithm Design

Several variables are important in the design of an adaptive algorithm. For each variable described below, there are several choices and each choice has different implementation complexity and impact on miss rate and traffic.

5.2.1 Neighboring Definition

Multiple VLs are "neighboring" if they can be part of a VL of larger size. In implementation, we don't want line size to increase very fast. The size of the larger VL is twice the largest size of all "neighboring" VLs. When several "neighboring" VLs have shared life time in the cache, we predict that they will have shared life time in the future. A larger VL can be formed from the "neighboring" VLs to exploit the spatial locality among "neighboring" lines.

There are two possible "neighoring" definitions depending on the line size relationship between "neighboring" VLs.

EQ – if "neighboring" VLs are of the same size.

NE – if "neighboring" VLs can have different sizes.

EQ "neighboring" is a one-to-one relationship and a larger line is formed by two VLs. This results in a simple algorithm implementation. Suppose the cache is direct-mapped and there are two "neighboring" lines vl_1 and vl_2. Two VCLs that can be created by vl_1 and vl_2 respectively are determined by the address mapping and are unique. Tag comparison of the first PCLs of both VCLs can be used to find whether vl_1 and vl_2 are in the cache at the same time.

NE "neighboring" is a many-to-many relationship. For a VL $(addr, line\_size)$, it can have as many as $\frac{line\_size}{pcl\_size}$ "neighboring" lines of smaller size. It can also have as many as

$$num\_line\_size - Log_2 \frac{line\_size}{pcl\_size}$$

"neighboring" lines of equal or larger size.

"Neighboring" is also a bi-directional relationship. The number of neighbors for a line can be greatly reduced by changing it to a uni-directional relationship with the following restriction:

a VL only neighbors another line of equal or larger size

Thus a VL $(addr, line\_size)$ can have at most

$$num\_line\_size - Log_2 \frac{line\_size}{pcl\_size}$$

"neighboring" lines.

The implementation of NE "neighboring" is complicated because multiple PCLs, each corresponding to a possible "neighboring" line, need to be checked to find whether a "neighborging" line is in the cache. But NE "neighboring" may enhance the chance of spatial locality detection.

5.2.2 Strategy for Cache Line Replacement

In an ALS cache, cache line replacement is more complicated because lines of different size can coexist in cache. On a miss-fetch, a VCL will be created. There are four possible scenarios on how existing VCLs will be destroyed.

1. no VCL will be destroyed;
2. one VCL with the same size as the newly created VCL will be destroyed;
3. multiple VCLs with smaller size than the newly created VCL will be destroyed;
4. one VCL with larger size than that of the newly created VCL will be destroyed.

In the first scenario, a new VL can be brought into the cache directly since there is no valid data in the cache. In the second scenario, the existing VCL will be sent to a write buffer if it is dirty. Next, algorithm *line_size_predict* is used to determine whether the line size needs change. Then the miss-fetched VL is brought into the cache. In the third scenario, the steps in scenario 2 are applied to every affected VCL.

Different replacement strategies can be used in scenario 4. For a VCL to be destroyed vcl_{old}, the VL vl_{old} in the vcl_{old}, the miss-fetched VL vl_{new} and the newly created VCL vcl_{new},

DEC – If the size of vcl_{old} is larger than the size of vcl_{new}, vcl_{old} and vl_{old} can be partitioned into small lines of half size. One of the smaller VCL will not be destroyed by vcl_{new} and can remain in the cache. If the size of vcl_{old} is equal or smaller than the size of vcl_{new}, line size change of vl_{old} will be determined by algorithm *line_size_predict* and the whole line will be replaced from cache by vcl_{new}.

NDEC –Algorithm *line_size_predict* is used to determine the line size change for vl_{old} and the whole line will be replaced from the cache by vcl_{new}.

PL – Algorithm *line_size_predict* is used to determine the line size change for vl_{old}. vcl_{new} will only replace the PCLs that it maps to.

DEC and NDEC are simple in implementation. The advantage of DEC to NDEC is a better cache utilization by keeping half of the existing line in the cache if the miss-fetched line is of smaller size. But this requires line size decrease that is not based on spatial locality inherent in the line and can have adverse effects on adaptation.

PL is an extension of DEC. It tries to keep as many PCLs of the existing VCL as possible in cache for a better cache utilization. At the same time, line size change is based on the spatial locality, which avoids the drawback of DEC by forcing line size decrease. But the implementation of PL is more complicated and additional measures are needed for line size and data consistency.

In scenario 3, it takes more time to miss-fetch in an ALS cache than that in a FLS cache. But a buffer can be used to swap out the PCLs that will be occupied by the newly created VCL. Then we can proceed to bring the miss-fetched VL into the cache. At the same time, steps for scenario 2 can be applied for every VCL in the buffer.

5.2.3 Order of Line Size Change Test

In algorithm *line_size_predict*, lines 2 and 3 are used for the line size increase test and lines 4 and 5 are used for the line size decrease test. In fact, the order can be reversed. If the increase test is done first, we call it IF. If the decrease test is done first, we call it DF. We will investigate both IF and DF for their effects in line size adaptation.

5.2.4 Policy for Actual Line Size Change

Line size change needs not occur immediately after the lower memory hierarchy receives line size increase/decrease requests. The following policies are possible:

- line size changes after one size change request.
- line size changes after N accumulated same size requests. "Opposite" requests offset each other.
- line size changes when current size satisfies a certain condition, such as being larger or smaller than a specific size.

In addition, increase and decrease requests can have different policies. We have evaluated the following policies for our experiments:

DT – direct; line size increases/decreases after one request.

3S – three states; line size increases/decreases after two accumulated increase/decrease requests.

IF – increase fast; line size increases after one increase request, line size decreases after two accumulated decrease requests.

DF – decrease fast; line size decreases after one decrease request, line size increases after two accumulated increase requests.

PF – partial fast; use IF policy for small line size, use DT policy for large line size.

DT is of interest to us since we want to see how quickly line size can adapt to changing spatial locality in a program. DT is also the simplest to implement. But one concern is that thrashing may occur during adaptation. Therefore, we investigate the 3S policy as well. IF policy gives priority to line size increase. It benefits miss rate reduction, but possibly at the cost of increased traffic. In contrast, DF policy gives priority to line size decrease. It benefits traffic reduction at the cost of increased miss rate. PF policy is conceived to have the benefits of both IF and DF.

5.3 Algorithm Naming Convention

In section 6, we will investigate how the adaptive algorithms work and which combination of variable choices described above will be beneficial to algorithm performance. Figures 3, 4 and 5 give only a general description of an adaptive algorithm, an actual algorithm can be described by four variables. We use a combination of the four variables to describe an adaptive algorithm. For example, an algorithm named EQ-DEC-IF-DT uses following set of selection for the four variables:

- neighboring definition: EQ
- cache replacement strategy: DEC
- line size increase/decrease test: IF
- line size change policy: DT

5.4 Line Size Consistency

A new consistency problem, line size consistency, arises in an ALS cache. When an address is in the cache, it has two line sizes: size in the cache and size in the lower memory hierarchy. Line size consistency requires these two sizes to be equal. Otherwise data in the lower memory hierarchy might be fetched into the cache and overwrite the newer copy of itself already in the cache.

For example, suppose address 0 has line size 8 in the cache and line size 16 in the lower memory hierarchy and address 8 is not in the cache. When address 8 is miss fetched, data from address 0 to 15 will be fetched into the cache and will overwrite data from address 0 to 7 already in the cache.

Line size consistency occurs in two possible scenarios:

1. On replacement from the cache, one of the neighboring lines makes line size increase decision and some neighboring lines are still in the cache. Then the newly formed line is miss-fetched and will overwrite the neighboring lines in the cache.

2. When PL replacement strategy is used and a line is to be replaced by another line with smaller size, some PCLs of the line to be replaced will remain in the cache and become "orphan" lines with line size equal to pcl_size. "Orphan" lines can be part of a miss-fetched line and will be overwritten by it.

An additional bit–$partBit$, which is one per PCL, can be used to tolerate line size inconsistency. For scenario 1, the neighboring lines in the cache will have their $partBits$ set. For scenario 2, the "orphan" lines will have their $partBits$ set. On a cache miss, for every PCL to be occupied by a miss-fetched line, if it $partBit$ is set and its tag matches the miss-fetched line, then this PCL is part of the miss-fetched line and will not be overwritten. On replacement from the cache, a line with $partBit$ set will not make line size change decision.

5.5 Set-associative ALS Cache

A n-way set-associative FLS cache with size of m KB can be implemented as n independent direct-mapped sub-caches each with size m/n KB. Similarly, a n-way set-associative ALS cache with size of m KB can be implemented as n independent ALS sub-caches each with size m/n KB. Cache lookup requests to either a set-associative FLS cache or a set-associative ALS cache can be sent to the n independent caches simultaneously. On a cache miss, one of the n possible cache lines, each within one of the independent cache, will be replaced with miss-fetched data. In case of a FLS cache, LRU or random replacement strategies can be used. If the same cache replacement strategies are used for an ALS cache, "neighboring" virtual lines can be distributed among n independent caches. This will increase the difficulty of spatial locality discovery and implementation complexity. Thus, the following replacement strategy is adopted:

> If "neighboring" VLs of a miss-fetched VL are in a sub-cache, then the miss-fetched line will also be filled into the same sub-cache. Othersie, LRU or random replacement strategies can be used to select a sub-cache to fill the miss-fetched line.

6 Algorithm Exploration

In this section, we explore the adaptive algorithm design space for an optimal algorithm. Unless otherwise noted, all caches in the experiments have the following configurations:

- 16KB direct-mapped;
- possible line sizes for FLS caches are– 8B, 16B, 32B, 64B, 128B and 256B;
- possible line sizes for ALS caches are– 8B, 16B, 32B, 64B, 128B and 256B;
- the initial line size of the ALS caches is 256B.

6.1 Adaptivity vs. Initial Line Size

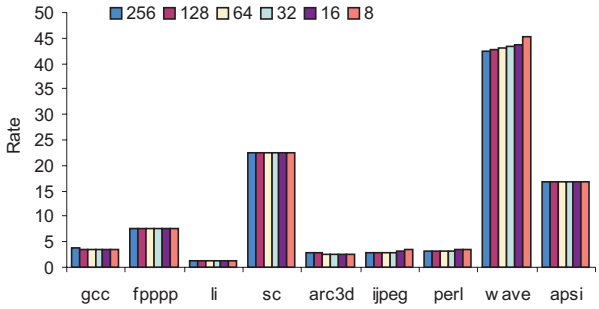

Figure 6: Miss rate with different initial line sizes in an ALS cache

Figure 6 shows miss rate in an ALS cache with different initial line sizes. The adaptive algorithm used is EQ-PF-IF-DEC. This figure clearly demonstrates that adaptivity works. The miss rates for each benchmark are almost same regardless the initial line sizes. This indicates that the initial line size selection is not important. Therefore, we use maximum line size (256B) as the initial line size in exploration phase below,

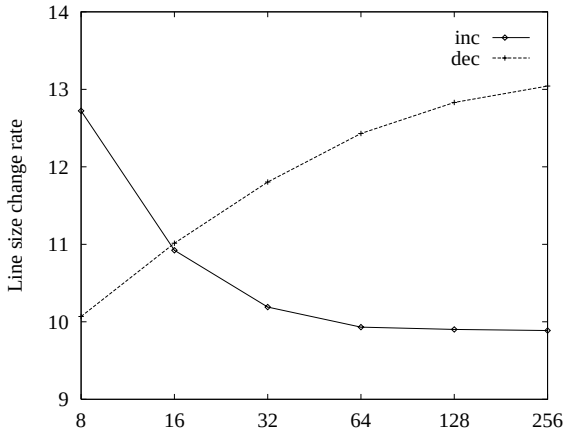

Figure 7: % misses producing line size change in GCC

Figure 7 shows the frequency of line size change during the execution of GCC. It is measured as a fraction of miss fetches resulting in replacement with a size change. The change is frequent, with over 20% of all the lines replaced, either increasing or decreasing on replacement. As can be expected, the relative number of increases vs. decreases changes with the initial virtual line size. This is another proof that adaptivity works well.

6.2 Miss Rate and Traffic in Direct-mapped Caches

With 2 neighboring definitions, 3 cache replacement strategies, 2 line size change orders and 4 line size change policies, we can construct a total of 48 adaptive algorithms. Cache behavior for every adaptive algorithm was simulated using the exploration benchmarks, and the following statistics were collected: miss rate, the fetch traffic between the cache and the lower memory hierarchy.

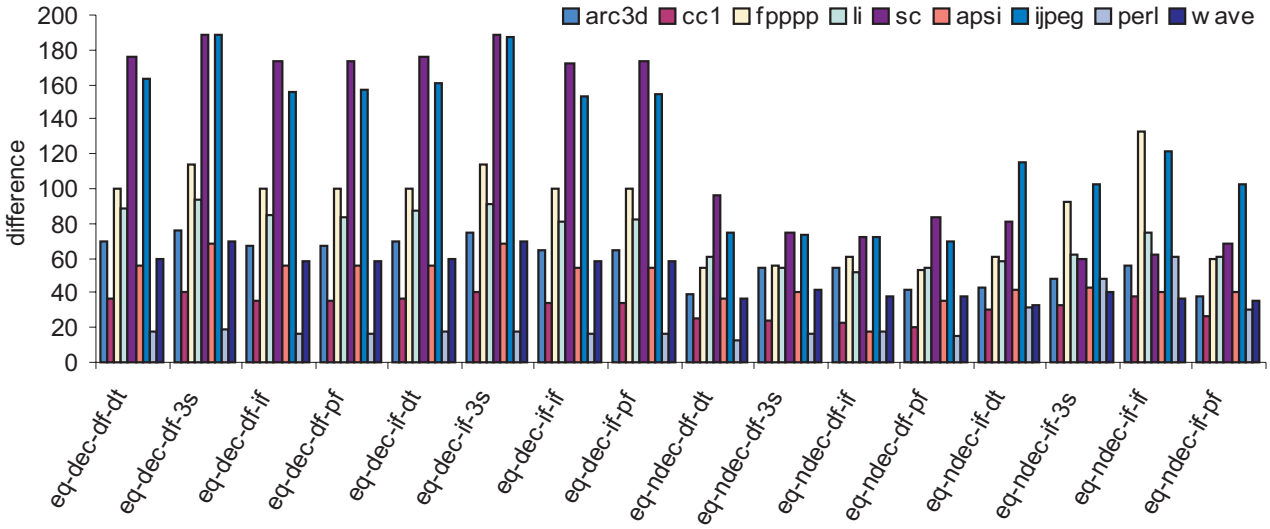

Figure 8: Normalized miss rate difference Δ_M in direct-mapped caches (part 1)

Figure 8, 9 and 10 show the normalized miss rate difference in direct-mapped caches. In the figures, there are 48 groups of bars, each group representing an adaptive algorithm. Most groups have 9 bars corresponding to 9 exploration benchmarks. If a bar is missing in a group, then the algorithm corresponding to the group achieves the best miss rate for the benchmark corresponding to the bar. For each exploration benchmark, miss rates M_{name} for 48 adaptive algorithms are obtained. Then best miss rate M_{opt} of the 48 miss rates is calculated. The normalized miss rate difference is defined as:

$$M^{\Delta}_{name} = \frac{M_{name} - M_{opt}}{M_{opt}} * 100$$

Since different benchmarks have different memory access behavior, different adaptive algorithms can have different effects on them. As can be seen from Figure 8, 9 and 10, miss rates for some algorithms can be more than twice the best miss rate. There is no single algorithm achieving best miss rates for all benchmarks. For example, NE-DEC-IF-3S achieves best miss rate for PERL; NE-PL-DF-IF achieves best miss rate for FPPPP and SC; NE-PL-DF-PF achieves best miss rate for GCC, LI and IJPEG; NE-PL-IF-IF achieves best miss rate for APSI and WAVE. A good algorithm should have smaller normalized miss rate differences for all benchmarks. NE-PL-DF-3S performs the best, with all miss rates within 8% of the best miss rates. NE-PL-DF-DT,

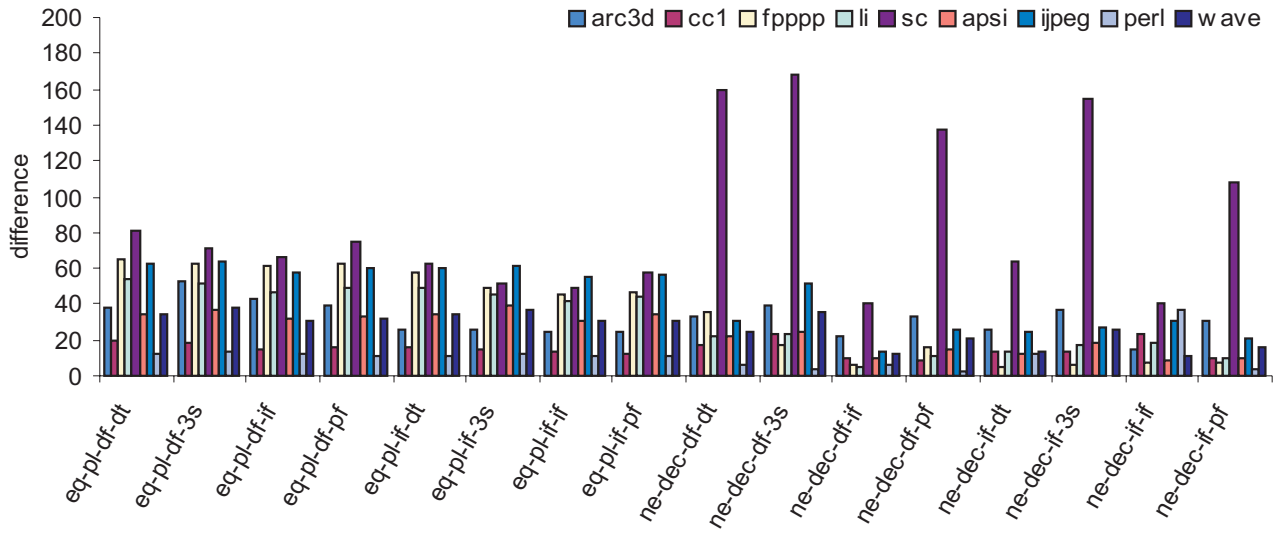

Figure 9: Normalized miss rate difference Δ_M in a direct-mapped caches (part 2)

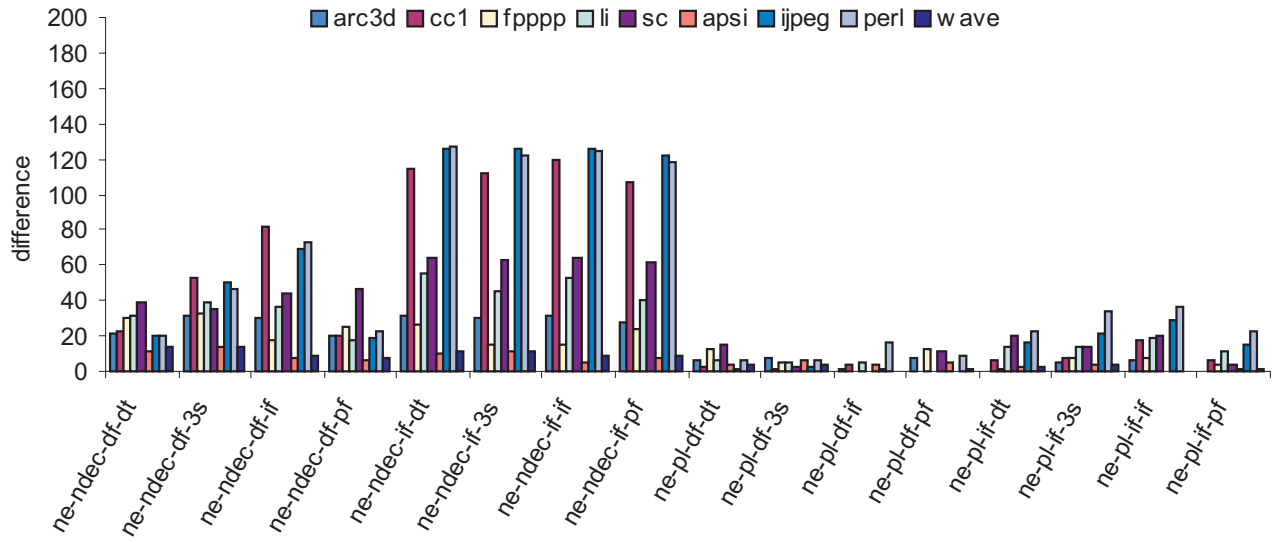

Figure 10: Normalized miss rate difference Δ_M in a direct-mapped caches (part 3)

NE-PL-DF-IF and NE-PL-DF-PF are also good algorithms, with miss rates within 15%, 16% and 14% of the best miss rates respectively.

Miss rates for algorithms EQ-PL-*-* in Figure 9 are better than those for algorithms EQ-NDEC-*-* in figure 8. In Figure 10, miss rates for algorithms NE-PL-*-* are better than those for algorithms NE-NDEC-*-*. The only difference between algorithms EQ-PL-*-*, NE-PL-*-* and EQ-NDEC-*-*, NE-NDEC-*-* is that EQ-PL-*-* and NE-PL-*-* keep partial VLs in cache when the size of newly created VCL is smaller than the size of the VCL to be destroyed. Therefore, the reduction in miss rates result from better cache utilization.

Algorithms EQ-*-*-* have much higher miss rates than NE-*-*-*. This proves that NE "neighboring" definition is beneficial for spatial locality discovery.

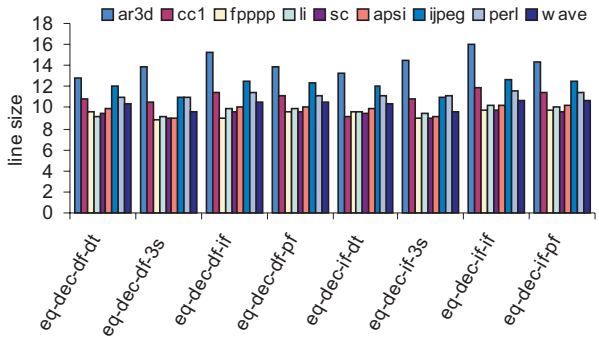

Figure 11: Average line size for EQ-DEC-*-* algorithms

From Figure 8, miss rates for all algorithms with name: EQ-DEC-*-* are among the worst. Figure 11 shows that the average line sizes for algorithms EQ-DEC-*-* range from 9B to 17B, very close to the minimum line size of 8B. The reason is that DEC decreases line size of a VL to be replaced from cache when this line is twice the size of the miss-fetched line. This kind of line size decrease is not based on the access behavior of a line and will result in more decrease requests than increase requests.

In Figure 10, algorithms NE-NDEC-IF-* also have worse miss rates. The reason is that the frequency of line size increase is much higher than the frequency of line size decrease. This leads to larger line size and causes more conflict misses.

In terms of the order of line size increase and decrease tests, sometimes IF results in lower miss rates and sometimes DF results in lower miss rates. What matters more is the algorithm context in which the tests are performed. For example, for algorithms NE-PL-*-*, DF is better than IF since IF will result in larger average line size and will cause more number of conflict misses.

For line size change policy, there is no optimal choice. Algorithm context matters here as well.

Figure 12, 13 and 13 show normalized traffic in direct-mapped caches. T^{32}_{name} is the traffic between a FLS cache with line size of 32B and the lower memory hierarchy for benchmark $name$. $T^{adaptive}_{name}$ is the traffic between an ALS cache using algorithm $name$ and the lower memory hierarchy for benchmark $name$. Normalized traffic rate is calculated as:

$$T^{normalized}_{name} = \frac{T^{adaptive}_{name}}{T^{32}_{name}} * 100$$

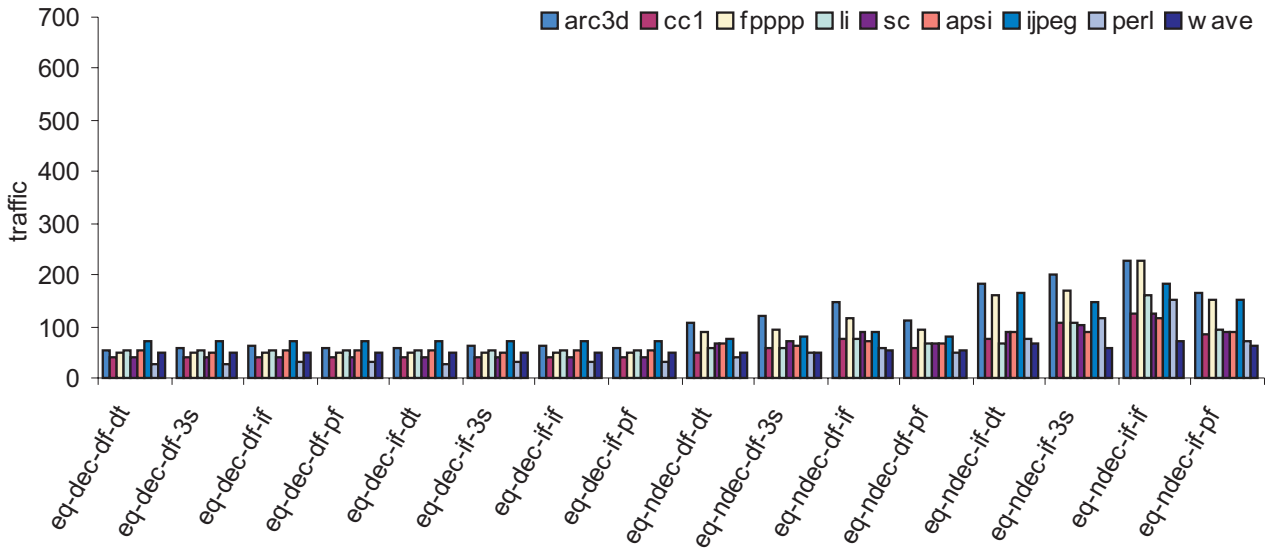

Figure 12: Normalized traffic rate in direct-mapped caches (part 1)

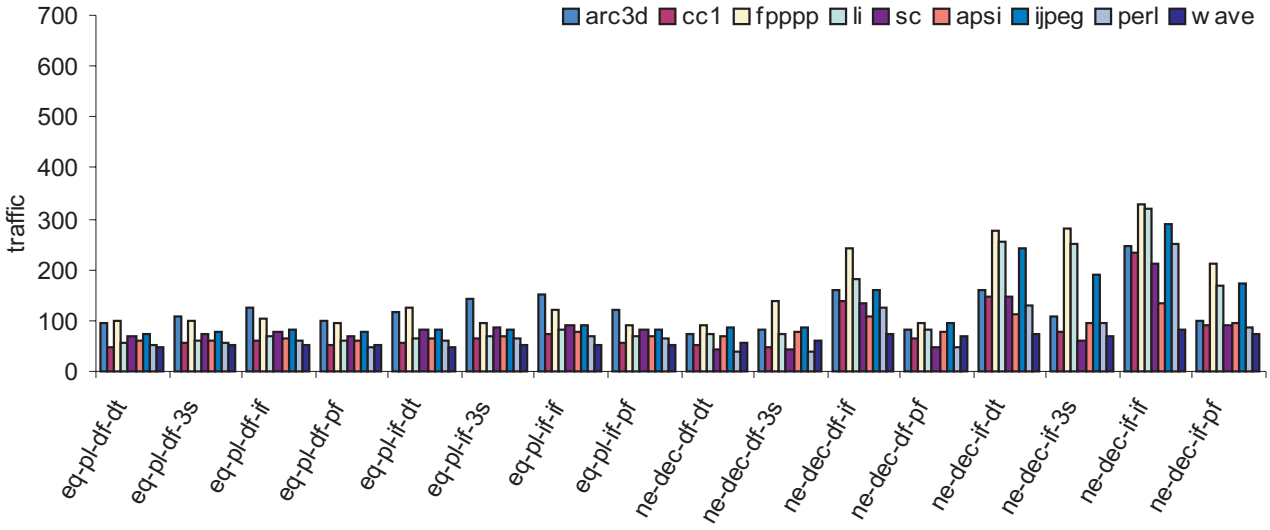

Figure 13: Normalized traffic rate in direct-mapped caches (part 2)

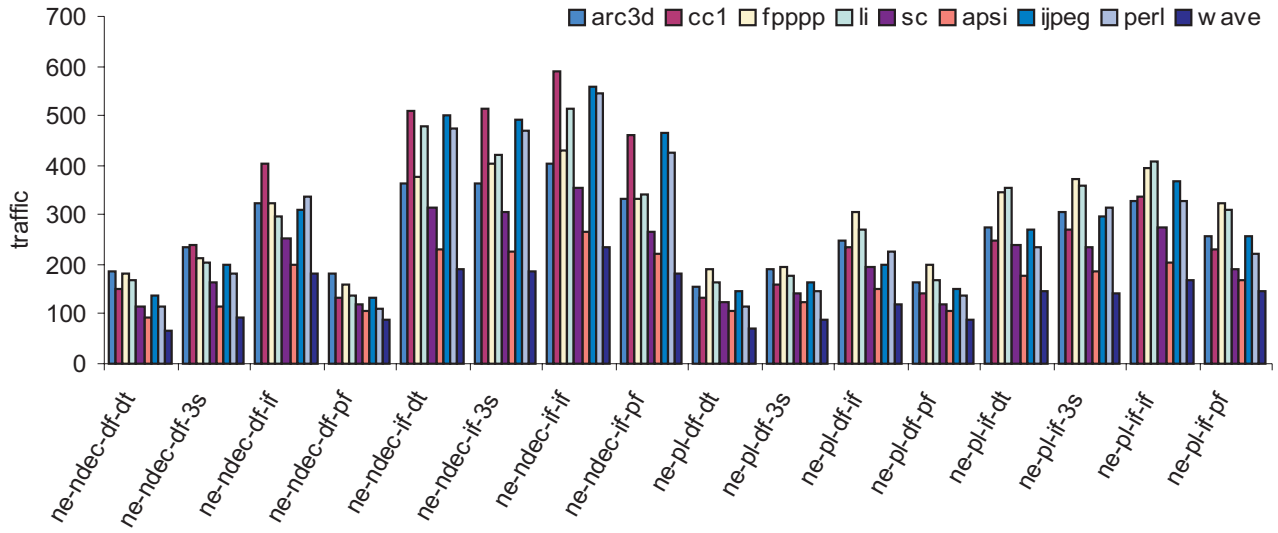

Figure 14: Normalized traffic rate in direct-mapped caches (part 3)

Algorithms NE-*-*-* have much higher traffic than EQ-*-*-*. The miss rate reduction comes at the cost of increased traffic. However, the increase in traffic is very small for "near-optimal" algorithms, eg. NE-PL-DF-DT, NE-PL-DF-3S and NE-PL-DF-PF.

6.3 Miss Rate and Traffic in 2-way Set-associative Cache

The same 48 adaptive algorithms are used in 16KB 2-way set-associative caches. Normalized miss rate difference is shown in Figure 15, 16 and 17. Normalized traffic rate is shown in Figure 18, 19 and 20.

The miss rate and traffic rate patterns are similar to those in direct-mapped caches. They are summarized below:

- Miss rates for NE-PL-*-* algorithms are among the best of all algorithms.
- Miss rates for EQ-DEC-*-* algorithms are the worst among all algorithms. They are also much worse than those in direct-mapped caches.
- The orrder of line size increase/decrease test is affected by the algorithm context.
- There is no optimal line size change policy. The algorithm context matters more.

6.4 Optimal Algorithm Selection

In terms of the optimal algorithm selection, we are primarily interested in the miss rate reduction. From Figures 10 and 17, we can see that the miss rates for algorithm NE-PL-DF-IF are

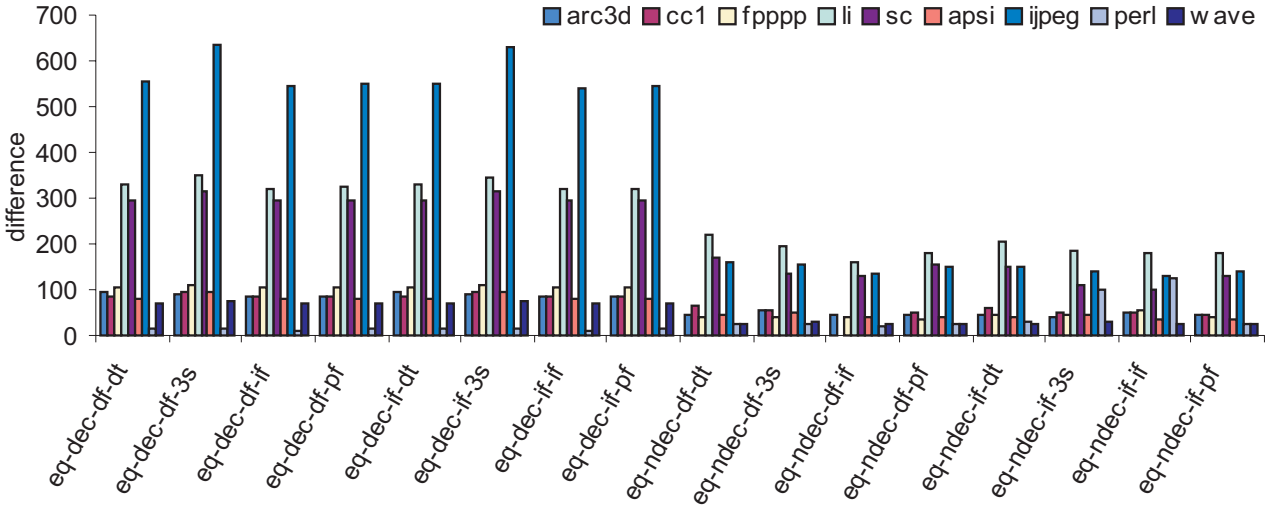

Figure 15: Normalized miss rate difference Δ_M in 2-way set-associative caches (part 1)

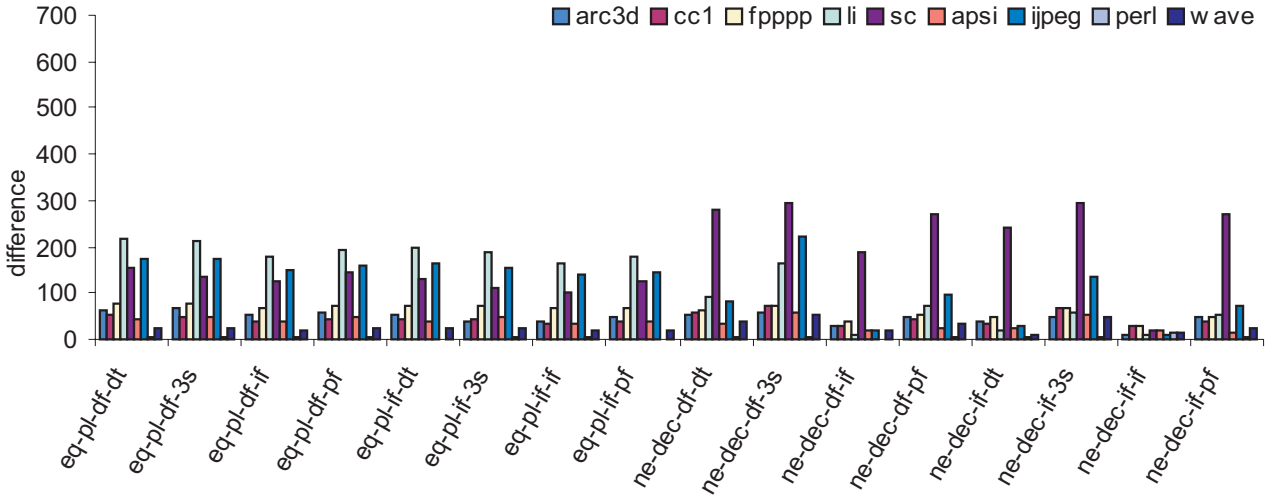

Figure 16: Normalized miss rate difference Δ_M in 2-way set-associative caches (part 2)

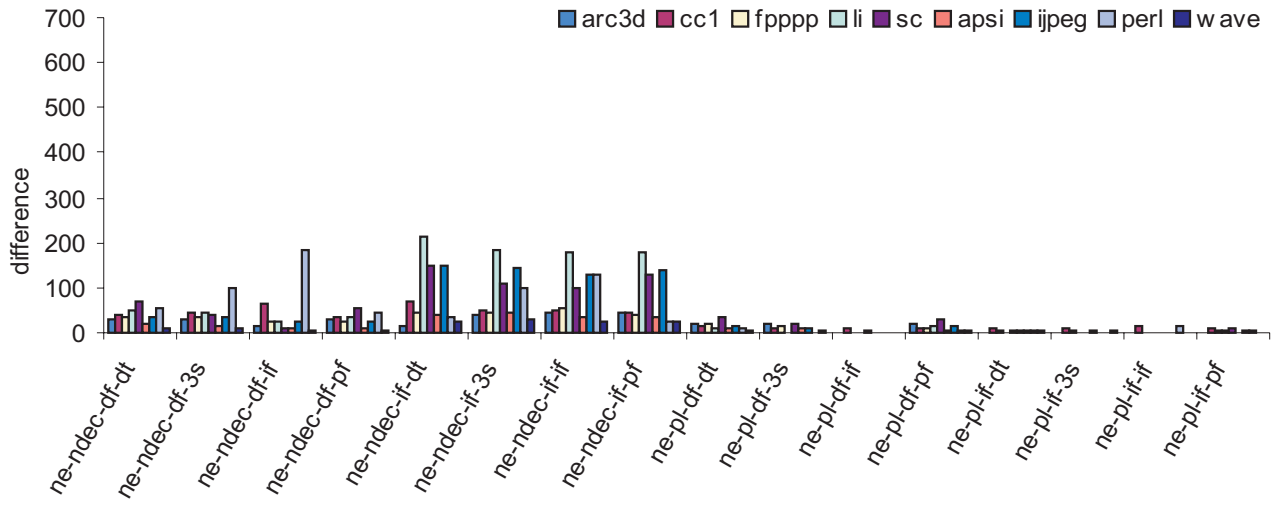

Figure 17: Normalized miss rate difference Δ_M in 2-way set-associative caches (part 3)

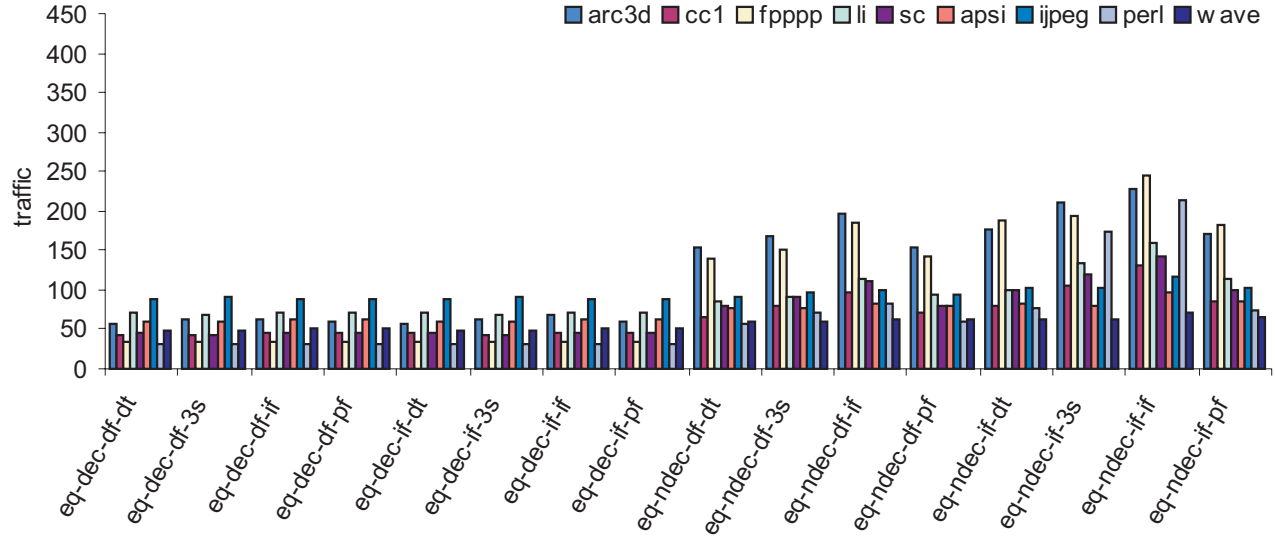

Figure 18: Normalized traffic rate in 2-way set-associative caches (part 1)

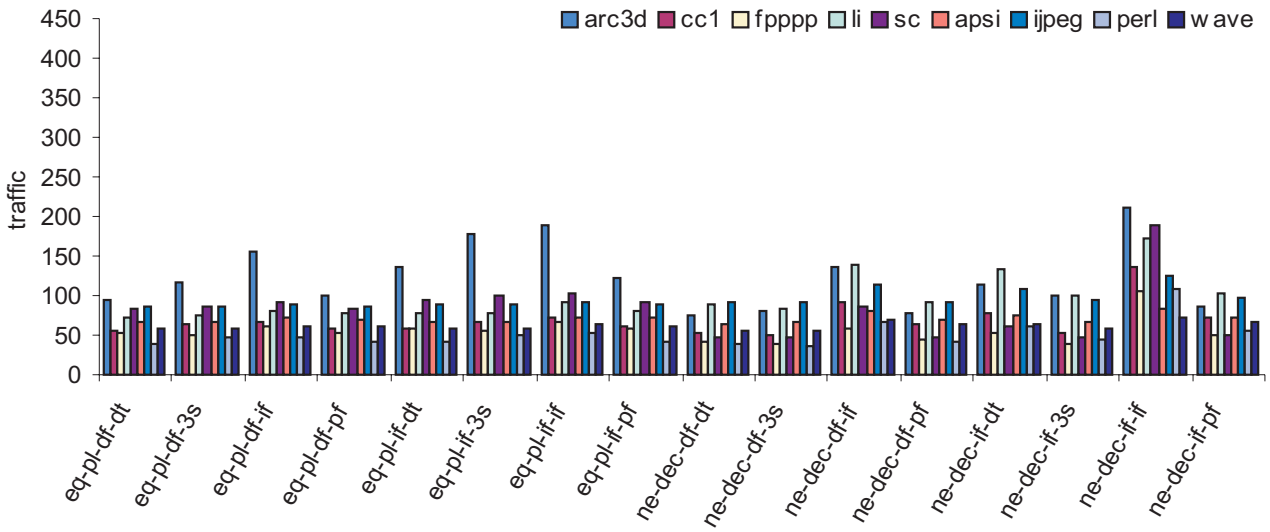

Figure 19: Normalized traffic rate in 2-way set-associative caches (part 2)

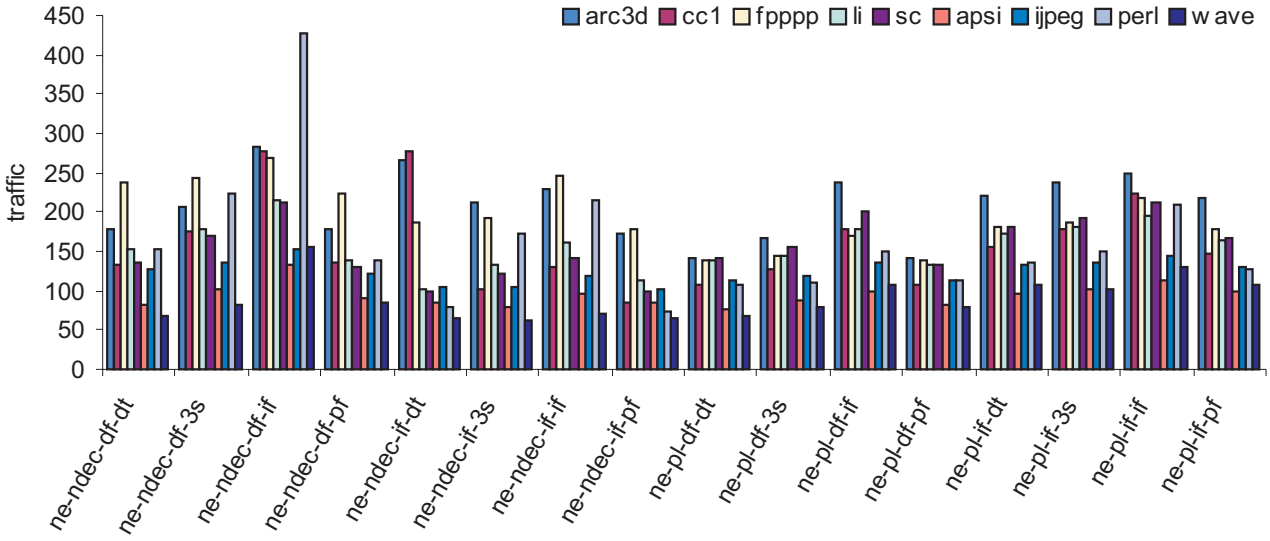

Figure 20: Normalized traffic rate in 2-way set-associative caches (part 3)

consistently within 20% of the optimal miss rates for all adaptive algorithms. NE-*-*-* algorithms always achieve better miss rates than EQ-*-*-* algorithms through better spatial locality discovery. *-PL-*-* algorithms also achieve better miss rates than *-DEC-*-* and *-NDEC-*-* through better cache utilization. DF for line size change test gives priority to line size decrease. IF for actual line size change policy gives priority to line size increase. DF and IF form a balance for line size increase and decrease. Therefore, we select NE-PL-DF-IF as the optimal algorithm for miss rate reduction. This algorithm is used in the next Section for performance evaluation.

7 Performance Evaluation

The following cache configurations are used for L1 caches:

- Possible L1 cache sizes are 16KB, 32KB and 64KB;
- Possible line sizes of FLS caches are 8B, 16B, 32B, 64B, 128B, and 256B;
- Initial line size of ALS caches is 256B;
- Algorithm for ALS caches is: NE-PL-DF-IF.

The following cache configurations are used for L2 caches:

- Possible L2 cache sizes are 128KB, 256KB and 512KB;
- Possible line sizes of FLS caches are 64B, 128B, 256B and 512B;
- Initial line size of ALS caches is 512B;
- Algorithm for ALS caches is: NE-PL-DF-IF.

The metrics used for evaluation are:

- Miss rate;
- Speedup of ALS caches over FLS caches.

L1 miss rate is calculated as:

$$L1\_miss\_rate = \frac{num\_L1\_miss}{num\_mem} * 100 \quad (1)$$

L2 miss rate can be calculated as:

$$L2\_miss\_rate = \frac{num\_L2\_miss}{num\_L1\_miss} * 100 \quad (2)$$

or as:

$$L2\_miss\_rate = \frac{num\_L2\_miss}{num\_mem} * 100 \quad (3)$$

The choice doesn't affect our performance analysis since we concentrate on miss reduction instead of the absolute difference in miss rate. We use equation 3 because the L2 miss rate calculated is independent of L1 cache configuration.

The execution time T in clock cycles for a benchmark with a 2-level cache is calculated using the following formula:

$$T = num\_instr + num\_L1\_miss * L1\_miss\_penalty + num\_L2\_miss * L2\_miss\_penalty \quad (4)$$

Although T is an approximation of real execution time, it should give an idea how well our adaptive algorithm works.

The speedup of an ALS cache over a FLS cache with line size of l is calculated as:

$$speedup_l^{adaptive} = \frac{T_l^{fixed}}{T^{adaptive}} \quad (5)$$

For comparison, a perfect memory hierarchy, which has zero L1 cache miss rate and zero L2 cache miss rate is used. The minimum execution time for a benchmark would be: num_instr. The maximum speedup for a FLS cache with line size of l is calculated as:

$$speedup_l^{maximum} = \frac{T_l^{fixed}}{num\_instr} \quad (6)$$

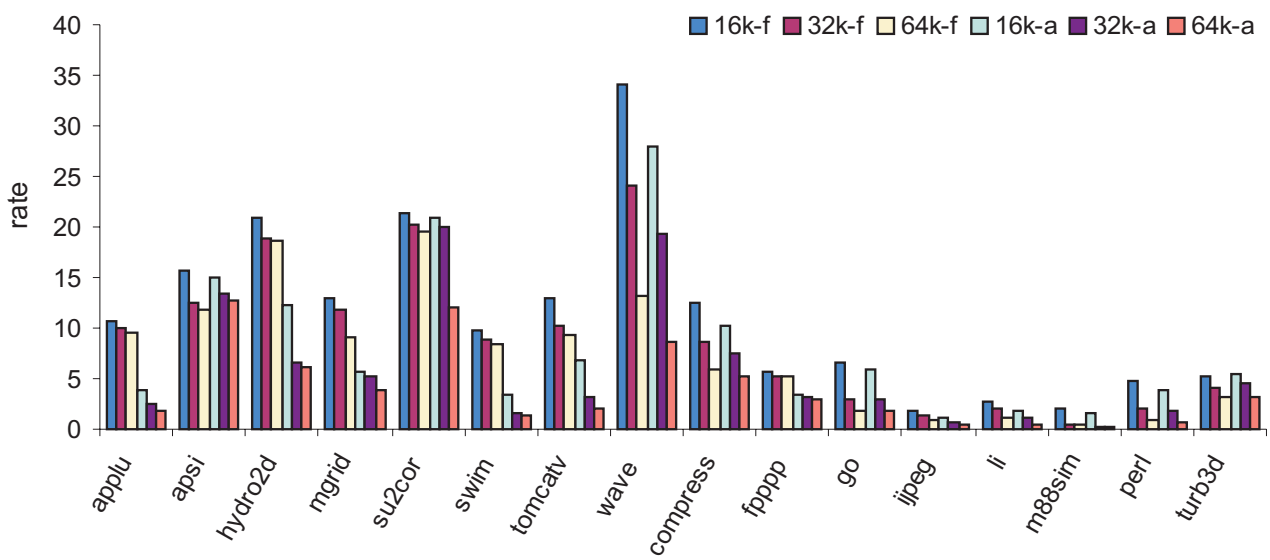

Figure 21: L1 miss rate for direct-mapped caches

Figure 21 shows L1 miss rate for direct-mapped caches. There are 6 bars for each benchmark. The first three bars correspond to miss rate for FLS caches with size 16KB, 32KB and 64KB

respectively. The line size is 32B, which is typical for today's L1 cache. Second three bars correspond to miss rates for ALS caches with size 16KB, 32KB and 64KB. For all benchmarks except APSI, GO and TURB3D, an ALS cache has better miss rates than a FLS cache with same size. For seven benchmarks, the ALS caches significantly reduce miss rate, with more than 50 percent reduction for APPLU, HYDRO2D, MGRID, SWIM and TOMCATV. For APPLU, HYDRO2D, MGRID, SWIM, TOMCATV and FPPPP, a 16KB ALS cache achieves better miss rate than a 64KB FLS cache. For IJPEG, a 16KB ALS cache outperforms a 32KB FLS cache. For M88SIM, a 32KB ALS cache outperforms a 64KB FLS cache.

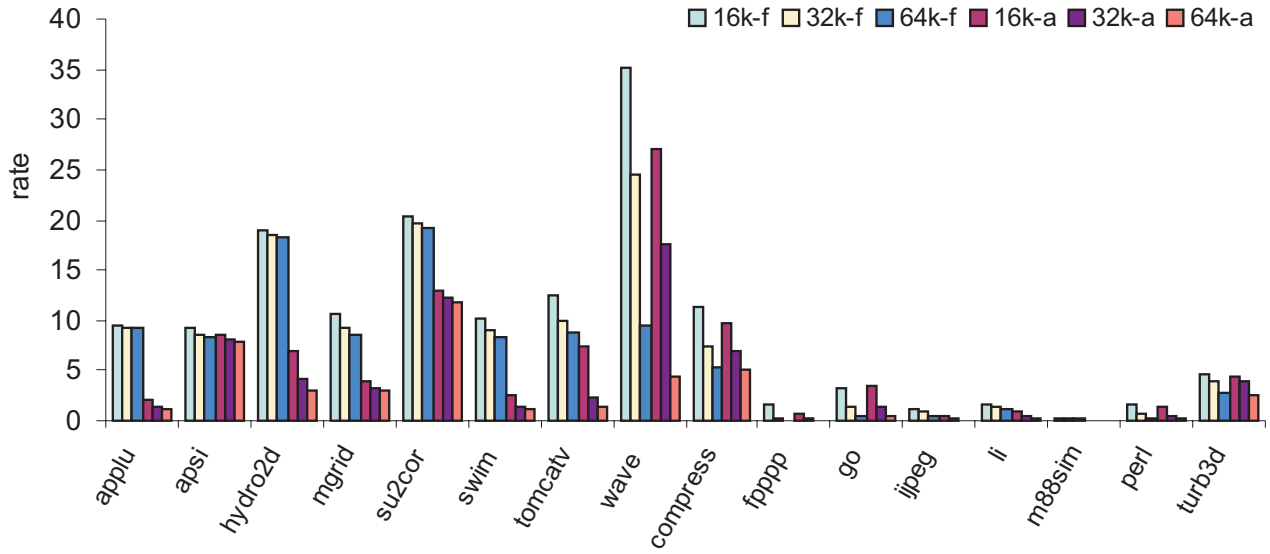

Figure 22: L1 miss rate for 2-way set-associative caches

Figure 22 shows L1 miss rate for 2-way set-associative caches. Line size of 32B is used for the FLS caches. For all benchmarks, ALS caches outperform FLS caches of same size.

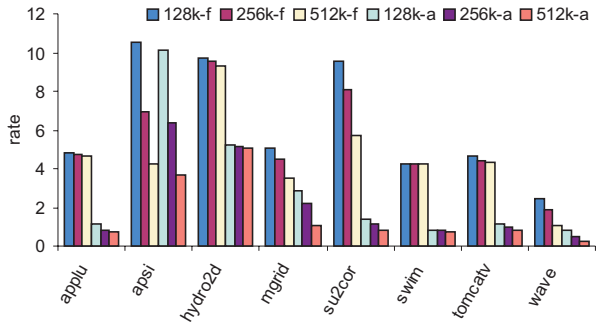

Figure 23: L2 miss rate for direct-mapped caches

Figure 23 shows L2 miss rates for direct-mapped caches. Line size 64B is used for FLS caches. For every benchmark, miss rate for ALS caches is much lower than that for FLS caches of same size. Comparing with Figure 21, we can see that L2 cache benefits more from line size adaptation than L1 cache does. A L2 ALS cache achieves higher miss rate reduction than L1 ALS cache does. Except for APSI, a L2 ALS cache with size 128KB outperforms a L2 FLS cache with size 512KB.

Figure 24 shows L2 miss rates for 2-way set-associative caches. The line size for FLS caches is 64B. For every benchmark, miss rate for an ALS cache is much lower than that for a FLS

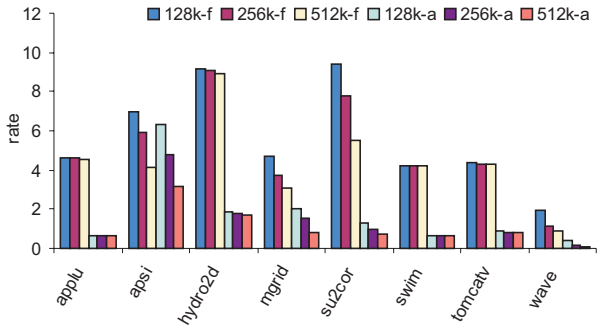

Figure 24: L2 miss rate for 2-way set-associative cache

cache with same cache size. Except for APSI, A L2 ALS cache with 128KB size outperforms a FLS cache with 512KB size. Comparing with Figure 23, we can see a similar pattern. A 2-way set-associative L2 ALS cache achieves higher miss rate reduction than a direct-mapped L2 ALS cache does.

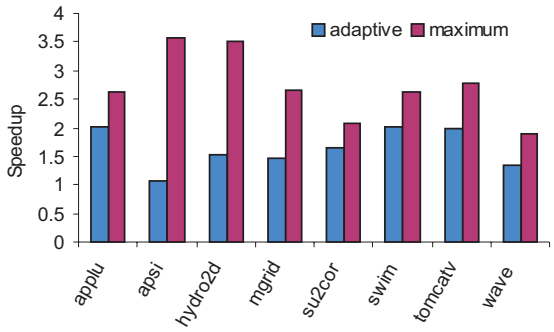

Figure 25: Normalized Speedup for direct-mapped ALS caches

Figure 25 shows the speedup of direct-mapped ALS caches. L1 cache size is 32KB and L2 cache size is 256KB. For FLS caches, the line size is 32B and 64B for L1 cache and L2 cache respectively. All benchmarks except APSI achieve show good speedups with ALS caches. The average speedup for all benchmarks is 1.63.

Figure 26 shows the speedup of 2-way set-associative ALS caches. L1 cache size is 32KB and L2 cache size is 256KB. For FLS caches, line size is 32B and 64B for L1 cache and L2 cache respectively. The average speedup by ALS caches is 1.81. The average maximum speedup by a perfect memory hierarchy is 2.6.

Figure 27 compares the L1 cache miss rate for a 2-way set-associative FLS cache and a direct-mapped ALS cache. The line size of the FLS cache is 32B. For 9 of the 16 benchmarks, an ALS cache outperforms a 2-way set-associative FLS cache. This is an advantage for the ALS cache because set-associative cache has higher hardware cost and the cache hit time of it is longer than

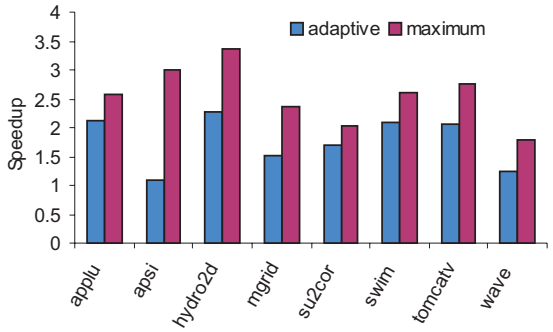

Figure 26: Normalized Speedup for 2-way set-associative ALS caches

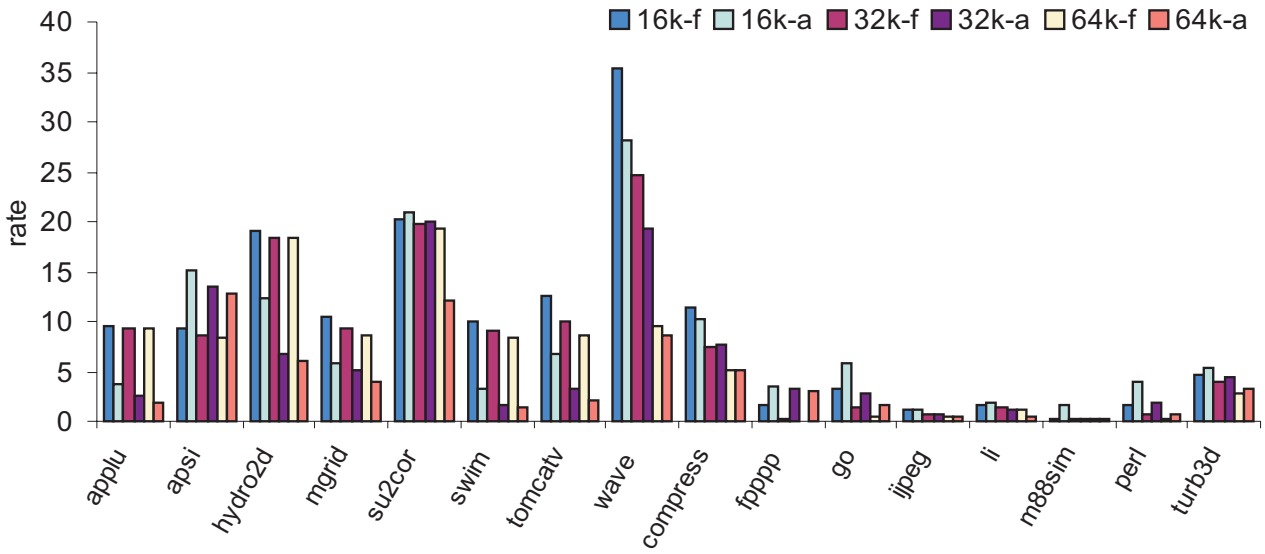

Figure 27: L1 Miss rate comparison: 2-way set-associative FLS cache vs. direct-mapped ALS cache

that of the ALS cache.

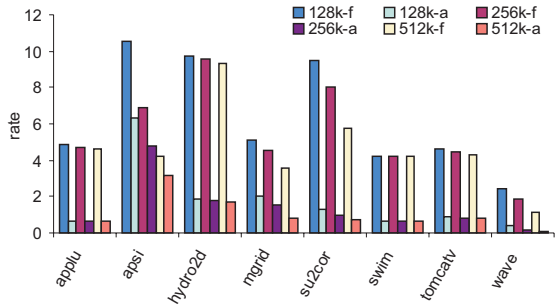

Figure 28: L2 Miss rate comparison: 2-way set-associative FLS cache vs. direct-mapped ALS cache

Figure 28 compares the L2 miss rate for a 2-way set-associative FLS cache and a direct-mapped ALS cache. The line size of the FLS cache is 64B. For every benchmark, miss rate by a direct-mapped ALS cache are much lower than those by a 2-way set-associative FLS cache. Comparing with 2-way set-associative FLS cache with size 512KB, the direct-mapped ALS cache of size 128KB results in less than half miss rate for all the benchmarks except APSI. This will greatly reduces hardware cost.

8 Analysis of Miss

An ALS cache predicts the next line size and there is a penalty if the prediction is not correct. In order to better understand the behavior of ALS caches, we simulate an ALS cache and a FLS cache with line size l simultaneously.

If the prediction by an ALS cache is wrong, the penalty is a hit in a FLS cache and a miss in an ALS cache. There are three scenarios between line size k for the ALS cache and line size l for the FLS cache:

- the FLS cache hits in a longer line size ($l\ GT\ k$);
- the FLS cache hits in an equal line size ($l\ EQ\ k$);
- the FLS cache hits in a shorter line size ($l\ LT\ k$).

We use three counters fhl, fhe and fhs to record corresponding number of memory accesses in each of the above scenario.

If the prediction by an ALS cache is correct, the benefit is a hit in the ALS cache and a miss in the FLS cache. There are three scenarios between line size k for the ALS cache and line size l for the FLS cache:

- the ALS cache hits in a longer line size ($k\ GT\ l$), where the advantage of the ALS cache comes from better spatial locality utilization;

- the ALS cache hits in an equal line size ($k\ EQ\ l$);
- the ALS cache hits in a shorter line size ($k\ LT\ l$), where the advantage of the ALS cache comes from reduced number of conflict misses.

We use three counters ahl, ahe and ahs to record corresponding number of memory accesses in each of the above scenario.

For completeness, we also use 3 additional counters: mem, number of memory accesses; $miss_{both}$, number of memory accesses missing in both caches; and hit_{both}, number of memory accesses hitting in both caches. The following equation holds:

$$mem = miss_{both} + hit_{both} + fhs + fhe + fhl + ahs + ahe + ahl \tag{7}$$

Number of cache misses in a FLS cache is:

$$miss_{fixed} = miss_{both} + ahs + ahe + ahl \tag{8}$$

Number of cache misses in ALS cache is:

$$miss_{adapt} = miss_{both} + fhs + fhe + fhl \tag{9}$$

We can derive from equations 8 and 9:

$$miss_{adapt} = miss_{fixed} + fhs + fhe + fhl - ahs - ahe - ahl \tag{10}$$

From simulation, fhe and ahe are much smaller than other counters. Therefore we use the following approximation:

$$miss_{adapt} \approx miss_{fixed} + fhs + fhl - ahs - ahl \tag{11}$$

In the miss figures for the L1 cache, there are six groups of bars corresponding to line size 8B, 16B, 32B, 64B, 128B and 256B for the FLS cache. In the miss figures for the L2 cache, there are four groups of bars corresponding to line size 64B, 128B, 256B and 512B for the FLS cache. For each group, there are five bars as follows:

- $miss$ bar, calculated as $\frac{miss_{fixed}}{mem} * 100$
- fhl bar, calculated as $\frac{fhl}{mem} * 100$
- fhs bar, calculated as $\frac{fhs}{mem} * 100$
- ahl bar, calculated as $\frac{ahl}{mem} * 100$
- ahs bar, calculated as $\frac{ahs}{mem} * 100$

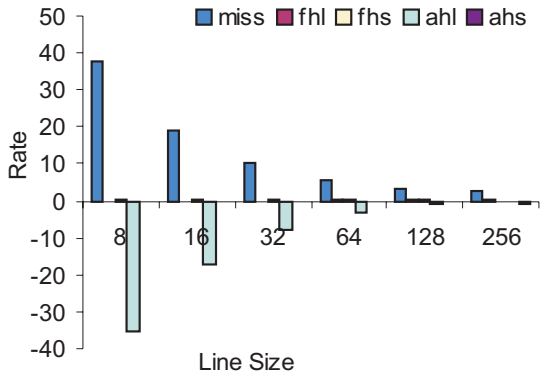

Figure 29: Miss analysis for APPLU with L1 32KB cache

Bars for *ahl* and *ahs* are plotted to the negative side of the axis to indicate the benefits of the ALS cache.

Miss rate for the ALS cache is not shown in the figures since it doesn't change with fixed line size. In addition, $miss_{adapt}$ for the ALS cache can be calculated from other metrics as follows:

$$miss_{adapt} = miss_{fixed} + fhl + fhs - ahl - ahs$$

Figure 29 shows miss analysis for APPLU. Bar *ahl* is the longest when the fixed line size is 8B. The ALS cache benefits the most from spatial locality utilization. As the fixed line size increases, bar *ahl* decreases because the FLS cache is able to exploit more spatial locality.

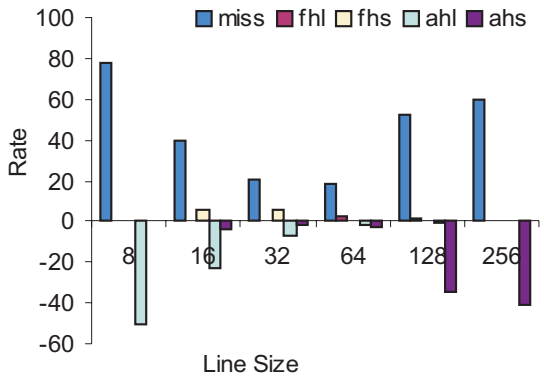

Figure 30: Miss analysis for SU2COR with L1 32KB cache

Figure 30 shows miss analysis for SU2COR. Bar *ahs* is long when fixed line size is 128B or 256B. Comparing Figure 29 with Figure 30, we can see different patterns of misses for the FLS cache. For APPLU, miss rate decreases with larger line size, the optimal miss rate is achieved at the largest line size. In SU2COR we can see a "U" shape miss rate curve where the optimal line

size for the FLS cache is 64B instead of 256B for APPLU. Miss rate increases sharply when line size changes from 64B to 128B and from 128B to 256B because the number of conflict misses increases with large line size. Yet the ALS cache can partition conflicting cache lines into smaller lines so that accesses to the smaller lines may not conflict. Thus we can see long *ahs* bars for line sizes of 128B and 256B, which is the result of conflict miss elimination by the ALS cache.

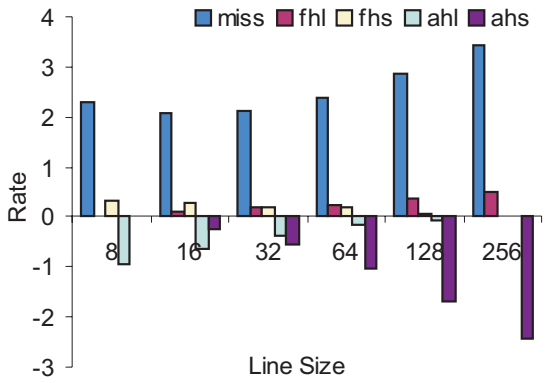

Figure 31: Miss analysis for PERL with L1 32KB cache

Figure 31 shows miss analysis for PERL. It shows how the ALS cache works for integer programs. The miss pattern is similar to that in Figure 30, where we can see a "U" shape miss rate in the FLS cache with the optimal line size of 32B. Miss rate decreases when line size changes from 8B to 16B and 16B and 32B as more spatial locality can be utilized by the FLS cache. The ability of the ALS cache in exploiting more spatial locality than the FLS cache becomes smaller. Hence bar *ahl* decreases. When line size increases from 64B to 256B, the number of conflict misses increases in the FLS cache. The ALS cache has more opportunities in partitioning large lines into small lines to eliminate conflict misses. Therefore bar *ahs* increases.

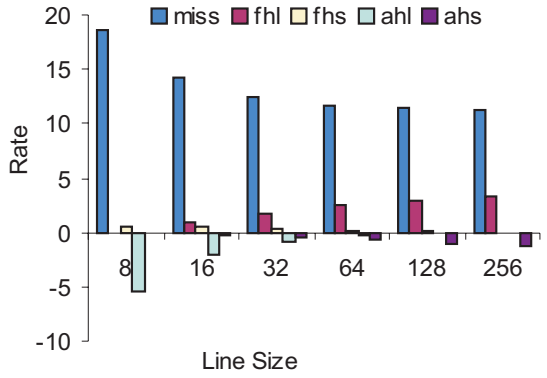

Figure 32: Miss analysis for APSI with L1 32KB cache

The ALS cache is not perfect. In Figures 29, 30 and 31 there are short bars for fhs and fhl, which signals mispredictions by the ALS cache. In Figure 32 for APSI with L1 32KB cache, bar fhl is long. This benchmark doesn't benefit from the ALS cache.

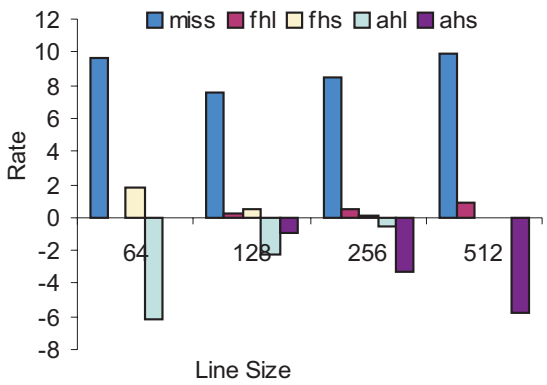

Figure 33: Miss analysis for HYDRO2D with L2 256KB cache

Figure 33 shows miss analysis for HYDRO2D with L2 256KB cache. It exhibits the similar pattern as SU2COR with L1 32K cache. We can see "U" shape miss rates for the FLS cache. Long bar for ahs indicates the ability of the ALS cache in conflict miss elimination. The miss rate for the ALS cache is lower than that for the FLS cache, which is achieved at line size 128B.

9 Conclusion

In this report, we have proposed a novel cache design with adaptive line size to suit the changing inter-application and intra-application spatial and temporal localities. This design results in significant reduction in cache misses. As discussed in section 6 and section 8, the benefits come from the following factors:

- More ability to utilize spatial locality inherent in an application;
- Reduction in the number of conflict misses, since line size change can partition conflicting lines into smaller size so that they may not conflict in the future.

The results show that a direct-mapped L2 128KB ALS cache outperforms a 2-way set-associative L2 512KB FLS cache. As L2 cache has low miss rate and high miss latency, it may be better to implement adaptivity in the L2 cache because of the large miss penalty.

Although this research is done for a simplified single-issue processor model, it is also useful for multiple-issue processors. In multiple-issue processors, the ratio of memory instructions per issue is much higher than that in single-issue processors. Hence the pressure on memory hierarchy to avoid cache misses is higher than that in single-issue processors. The benefits from the ALS cache will be more pronounced.

References

[1] D. H. Albonesi. Dynamic ipc/clock rate optimization. In *Int'l Symp. Computer Architecture*, pages 282–292, 1998.

[2] A. Chien and R. Gupta. MORPH: A system architecture for robust high performance using customization. In *Frontiers*, pages 268–277, July 1992.

[3] A. Chien and J. Kim. Planar adaptive routing: low-cost adaptive networks for multiprocessors. In *Int'l Symp. Computer Architecture*, pages 268–277, 1992.

[4] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Int'l Conf. Parallel Processing*, 1993.

[5] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Trans. Parallel and Distributed Systems*, 4:466–475, 1993.

[6] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Int'l Conf. on Supercomputing*, pages 338–347, 1995.

[7] E. H. Gornish and A. Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. In *Int'l Conf. Parallel Processing*, 1994.

[8] K. Inoue, K. Kai, and K. Marukami. High bandwidth, variable line-size cache architecture for merged DRAM/logic LSIs. *Japanese IEICE Transactions on Electronics*, E81-C(9):1438–1447, Sep. 1999.

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

[10] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Int'l Symp. Computer Architecture*, pages 302–313, 1994.

[11] T. Juan, S. Sanjeevan, and J. Navaro. Dynamic history-length fitting: a third level of adaptivity for branch prediction. In *Int'l Symp. Computer Architecture*, pages 155–166, 1998.

[12] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Int'l Symp. on Computer Architecture*, pages 357–368, 1998.

[13] T. Matsumoto, K. Nishimura, T. Kudoh, K. Hiraki, H. Amano, and H. Tanaka. Distributed shared memory architecture for jump-1: a general-purpose mpp prototype. In *Int'l Symp. Parallel Architectures, Algorithms, and Networks*, pages 131–137, 1996.

[14] MIPS Corporation. *MIPS R3000 hardware manual, MIPS Corporation.*

[15] S. Turner and A. Veidenbaum. Scalability of the cedar system. In *Int'l Conf. on Supercomputing*, pages 247–254, 1994.

[16] J. Veenstra and R. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Int'l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 201–207, 1994.

[17] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Int'l Conf. on Supercomputing*, pages 145–154, 1999.

APPENDIX

Detailed cache simulation results are shown in the appendix. L1 cache is simulated for all benchmarks. L2 cache is simulated for following benchmarks with significant L2 miss rates: APPLU, APSI, HYDRO2D, MGRID, SU2COR, SWIM, TOMCATV, WAVE.

The configurations for L1 cache are:

- cache size– 16KB, 32KB, 64KB
- line size for FLS cache– 8B or 16B or 32B or 64B or 128B or 256B
- line sizes for ALS cache– 8B, 16B, 32B, 64B, 128B and 256B

The configurations for L2 cache are:

- cache size– 128KB, 256KB, 512KB
- line size for FLS cache– 64B or 128B or 256B or 512B
- line sizes for ALS cache– 8B, 16B, 32B, 64B, 128B and 256B

Figures 34 to 81 show the miss rates for FLS caches and ALS caches.

Figures 82 to 129 show the normalized traffic rates for ALS caches. L1 normalized traffic is compared to traffic obtained for FLS cache with cache size of 16KB and line size of 8B. L2 normalized traffic is compared to traffic obtained for FLS cache with cache size of 128KB and line size of 64B .

Figures 130 to 153 show the miss analysis to find the tradeoffs in using ALS cache vs. FLS cache.

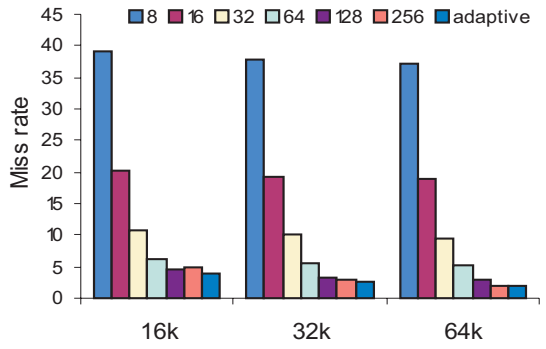

Figure 34: APPLU: L1 direct-mapped cache

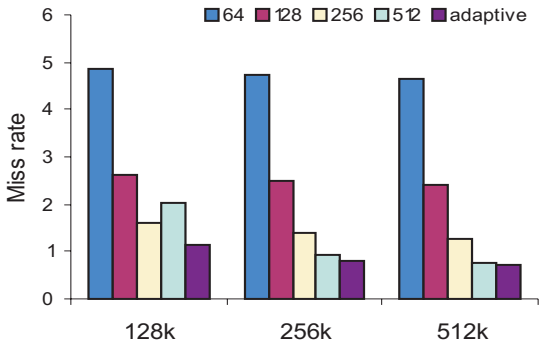

Figure 35: APPLU: L2 direct-mapped cache

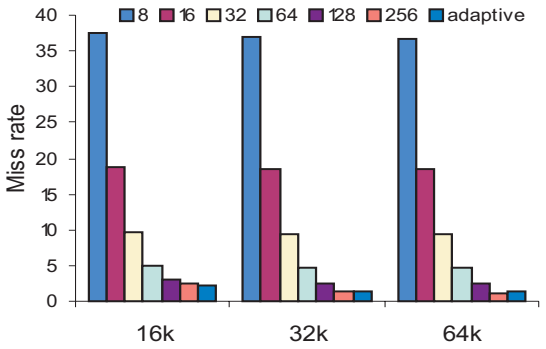

Figure 36: APPLU: L1 2-way set-associative cache

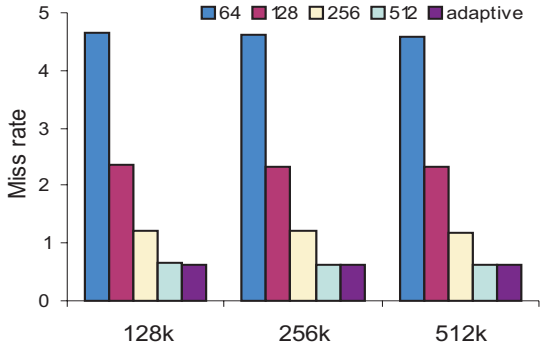

Figure 37: APPLU: L2 2-way set-associative cache

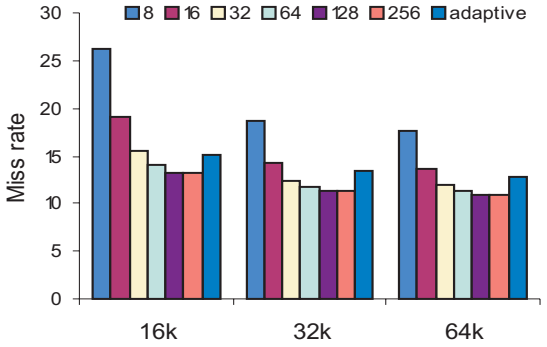

Figure 38: APSI: L1 direct-mapped cache

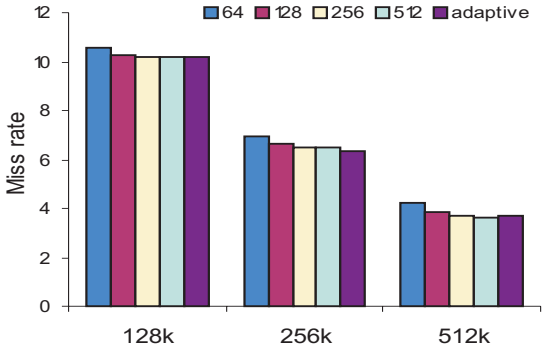

Figure 39: APSI: L2 direct-mapped cache

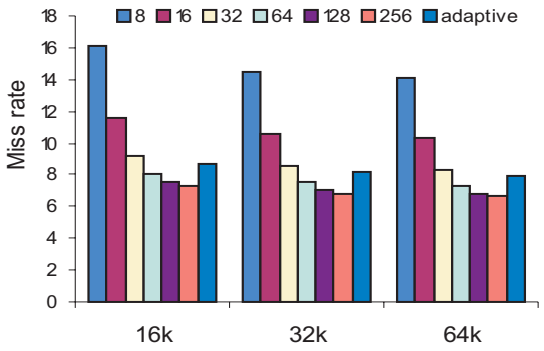

Figure 40: APSI: L1 2-way set-associative cache

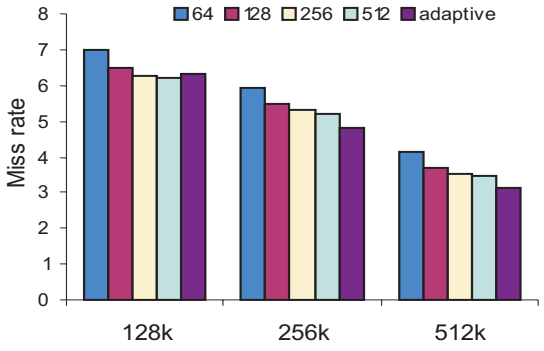

Figure 41: APSI: L2 2-way set-associative cache

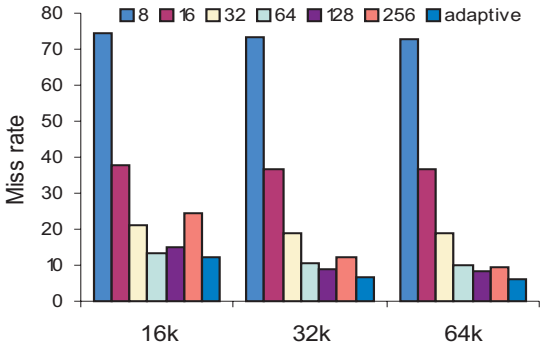

Figure 42: HYDRO2D: L1 direct-mapped cache

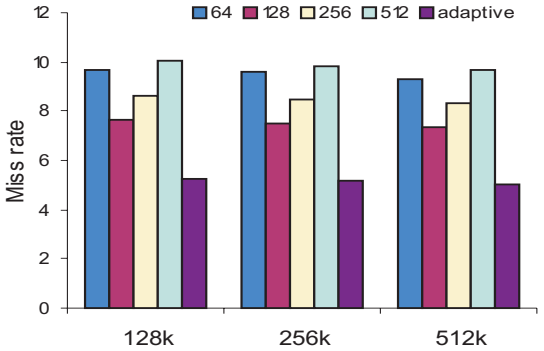

Figure 43: HYDRO2D: L2 direct-mapped cache

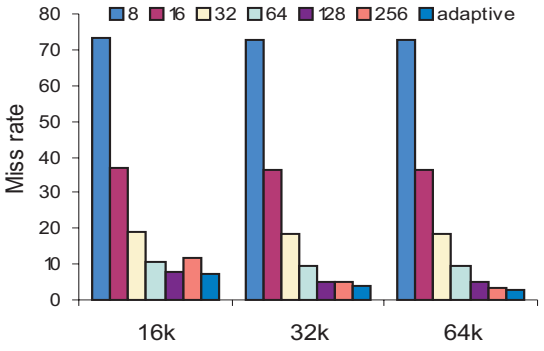

Figure 44: HYDRO2D: L1 2-way set-associative cache

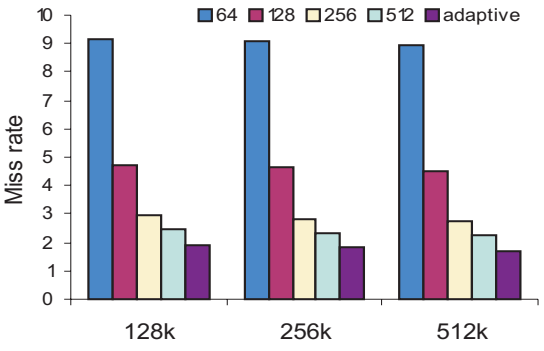

Figure 45: HYDRO2D: L2 2-way set-associative cache

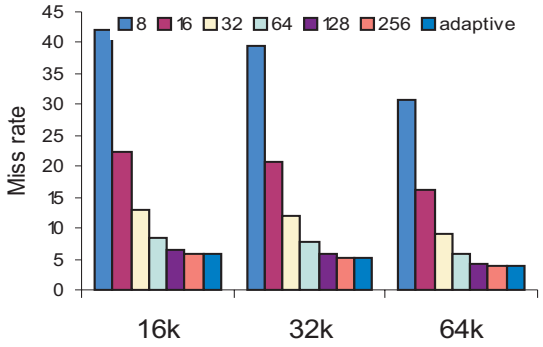

Figure 46: MGRID: L1 direct-mapped cache

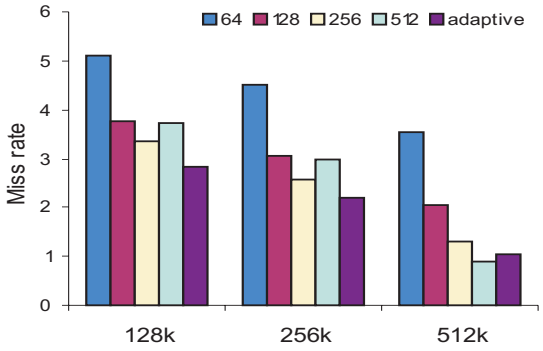

Figure 47: MGRID: L2 direct-mapped cache

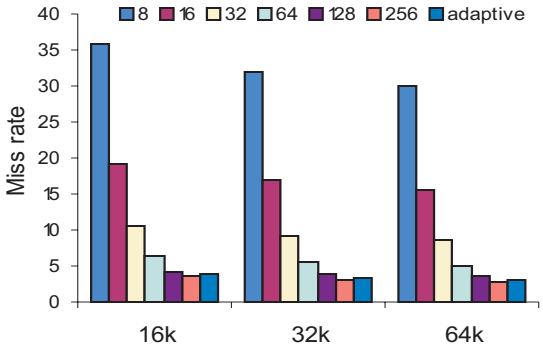

Figure 48: MGRID: L1 2-way set-associative cache

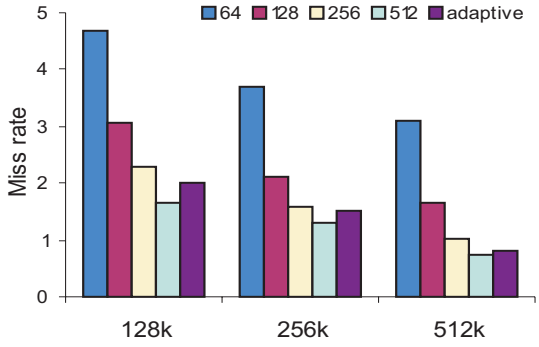

Figure 49: MGRID: L2 2-way set-associative cache

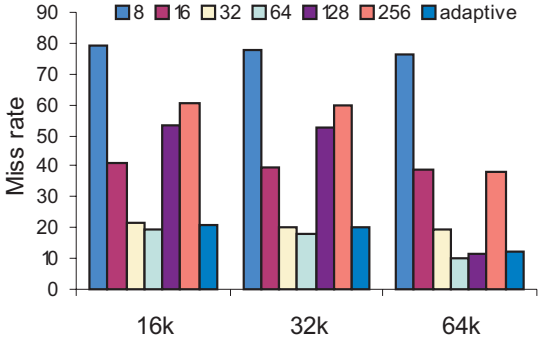

Figure 50: SU2COR: L1 direct-mapped cache

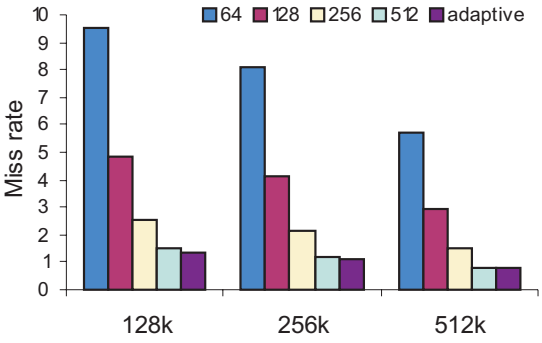

Figure 51: SU2COR: L2 direct-mapped cache

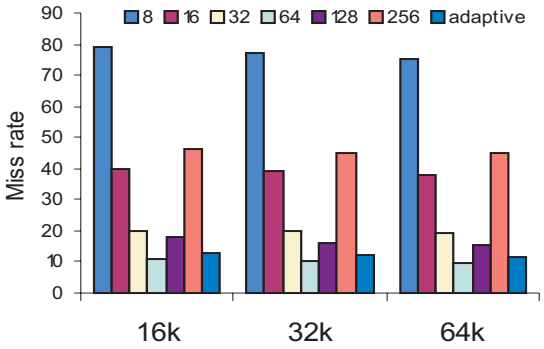

Figure 52: SU2COR: L1 2-way set-associative cache

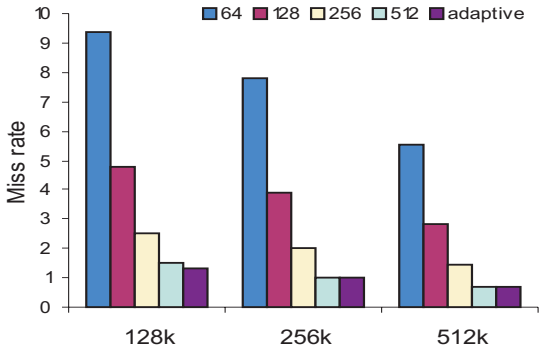

Figure 53: SU2COR: L2 2-way set-associative cache

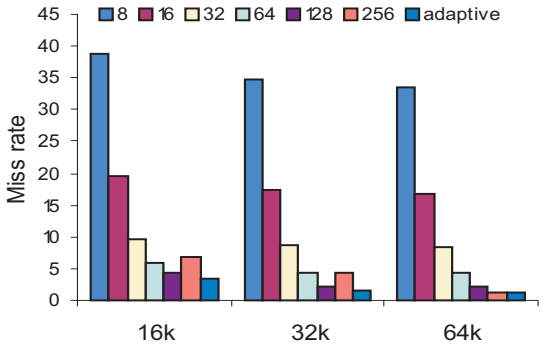

Figure 54: SWIM: L1 direct-mapped cache

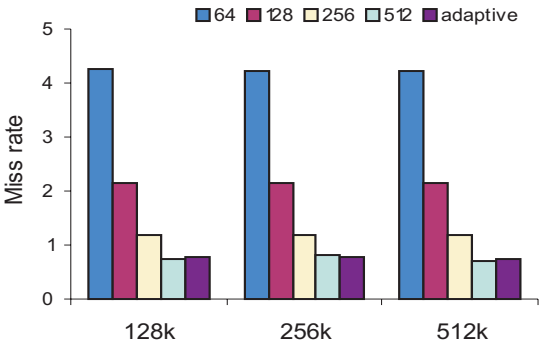

Figure 55: SWIM: L2 direct-mapped cache

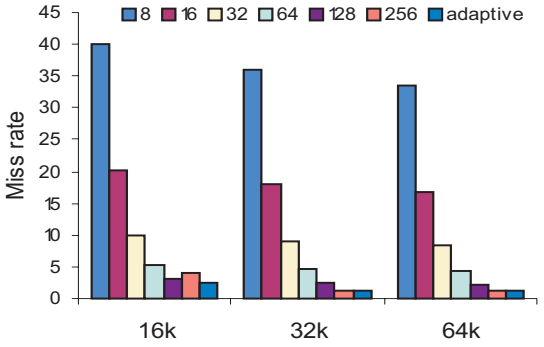

Figure 56: SWIM: L1 2-way set-associative cache

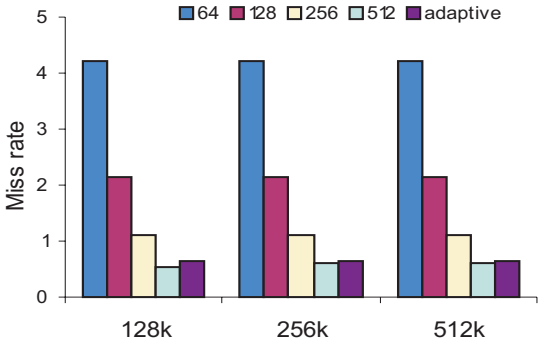

Figure 57: SWIM: L2 2-way set-associative cache

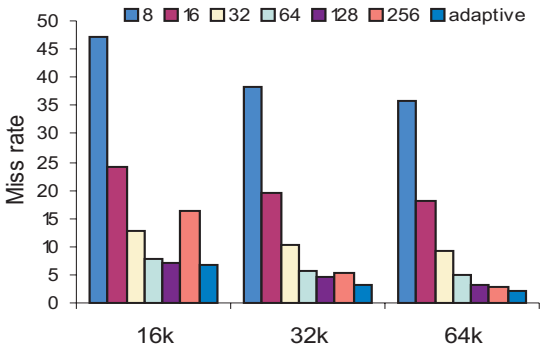

Figure 58: TOMCATV: L1 direct-mapped cache

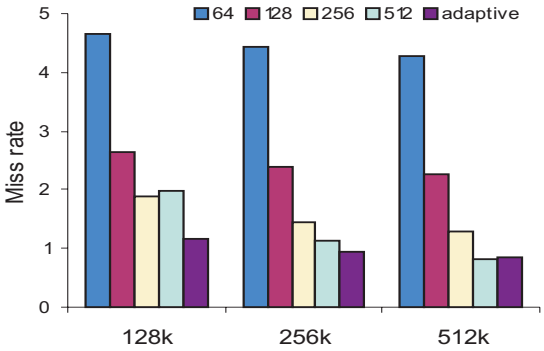

Figure 59: TOMCATV: L2 direct-mapped cache

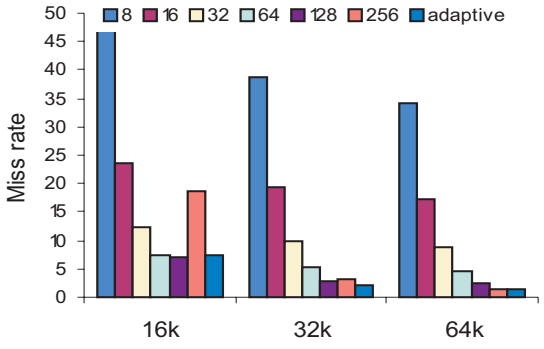

Figure 60: TOMCATV: L1 2-way set-associative cache

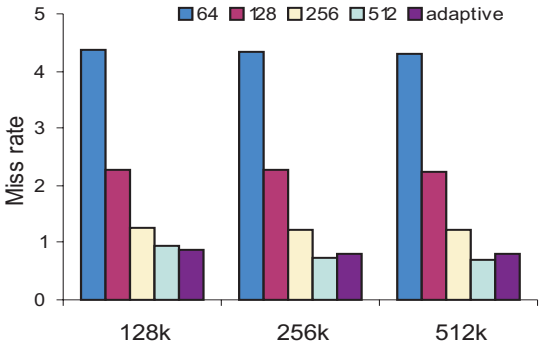

Figure 61: TOMCATV: L2 2-way set-associative cache

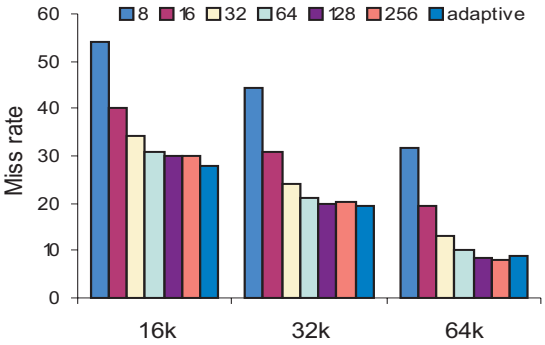

Figure 62: WAVE: L1 direct-mapped cache

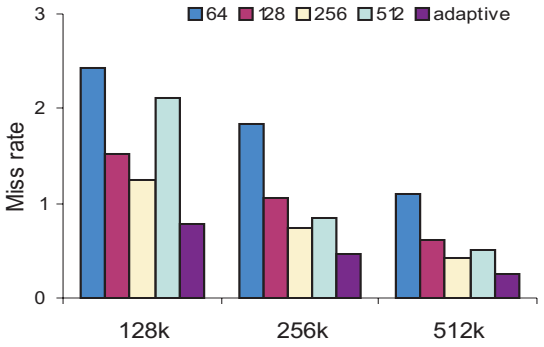

Figure 63: WAVE: L2 direct-mapped cache

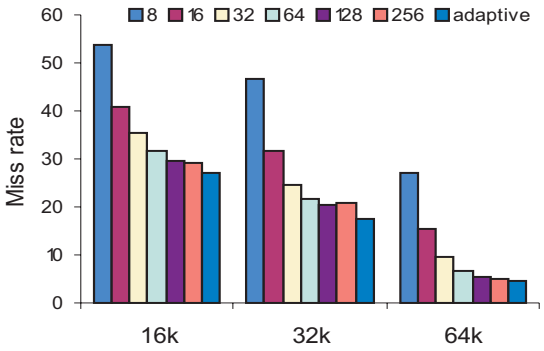

Figure 64: WAVE: L1 2-way set-associative cache

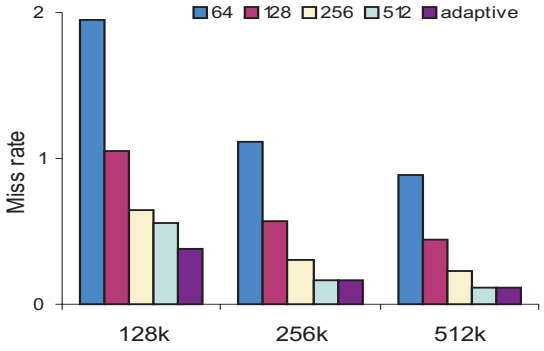

Figure 65: WAVE: L2 2-way set-associative cache

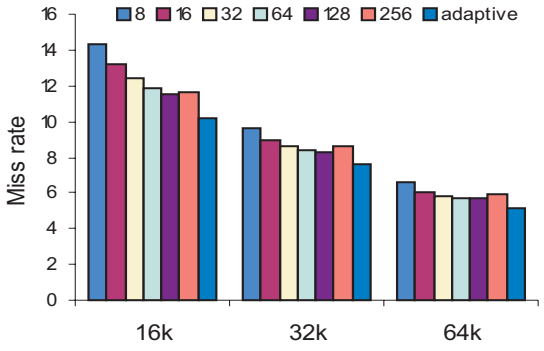

Figure 66: COMPRESS: L1 direct-mapped cache

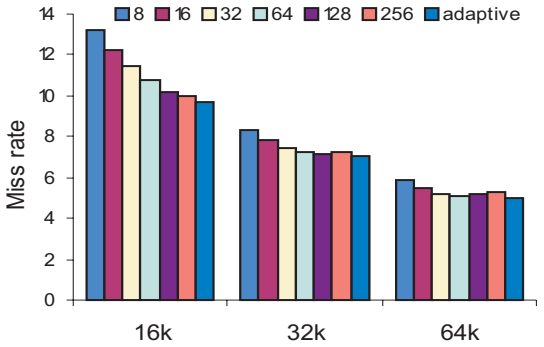

Figure 67: COMPRESS: L1 2-way set-associative cache

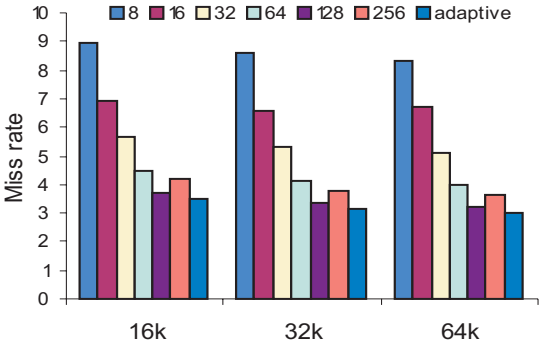

Figure 68: FPPPP: L1 direct-mapped cache

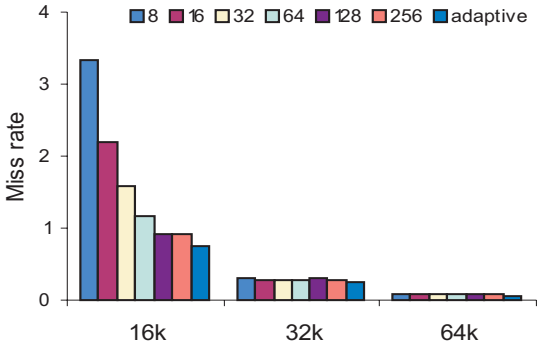

Figure 69: FPPPP: L1 2-way set-associative cache

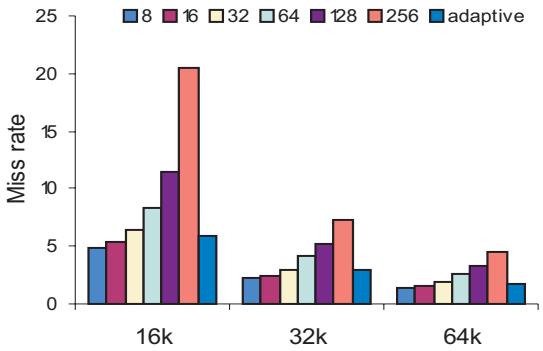

Figure 70: GO: L1 direct-mapped cache

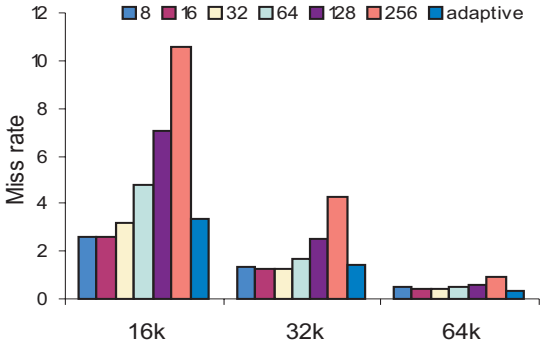

Figure 71: GO: L1 2-way set-associative cache

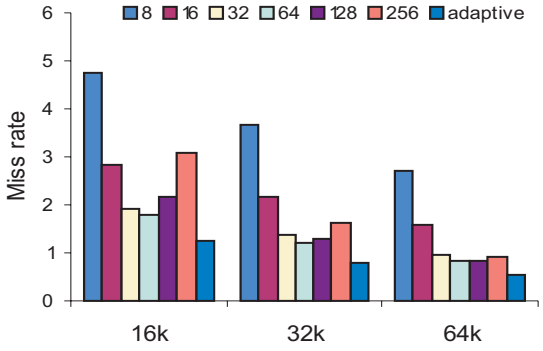

Figure 72: IJPEG: L1 direct-mapped cache

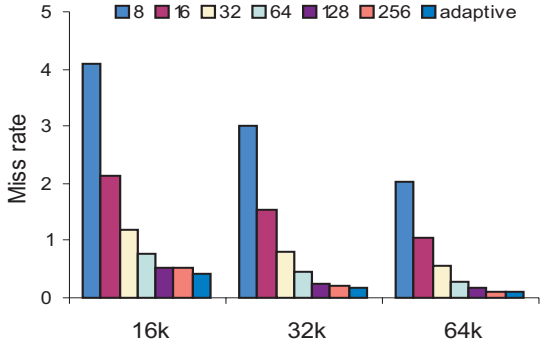

Figure 73: IJPEG: L1 2-way set-associative cache

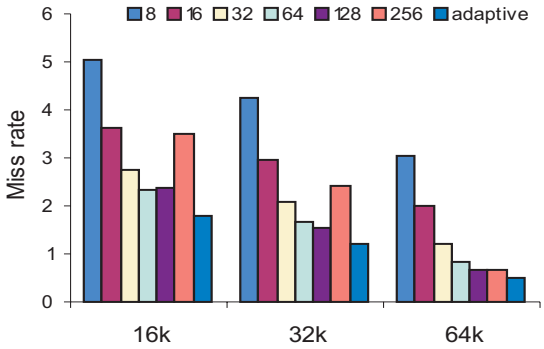

Figure 74: LI: L1 direct-mapped cache

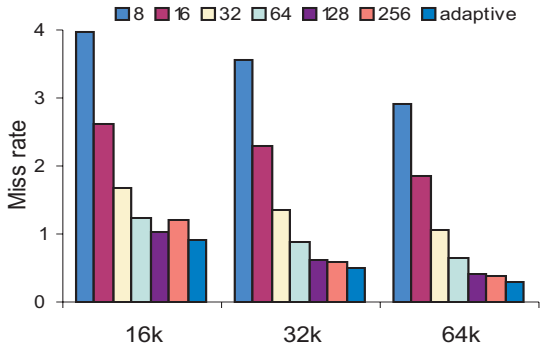

Figure 75: LI: L1 2-way set-associative cache

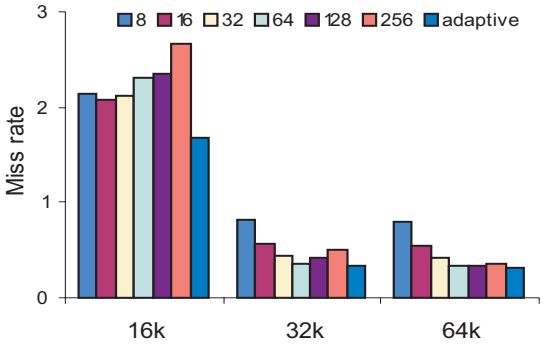

Figure 76: M88SIM: L1 direct-mapped cache

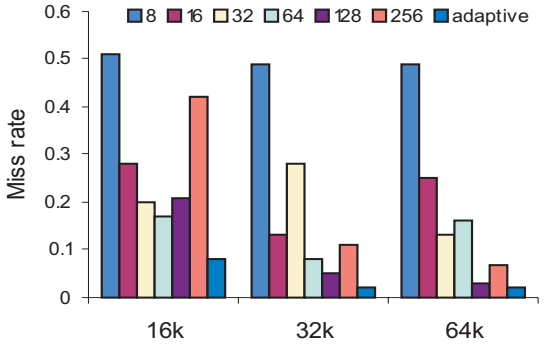

Figure 77: M88SIM: L1 2-way set-associative cache

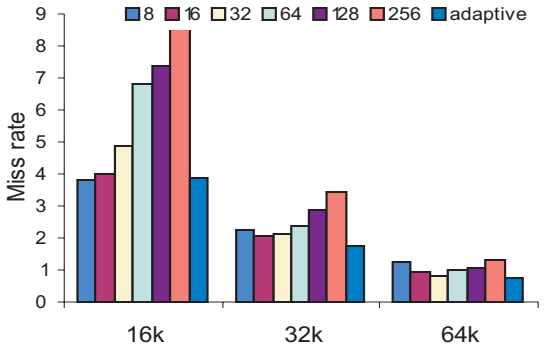

Figure 78: PERL: L1 direct-mapped cache

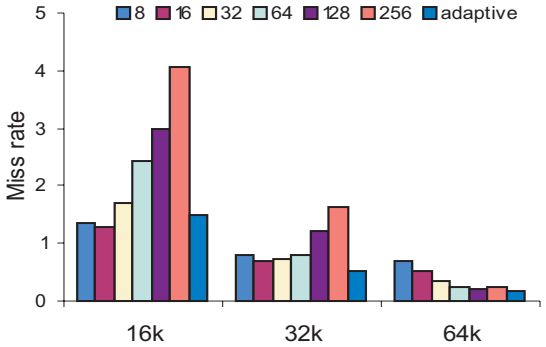

Figure 79: PERL: L1 2-way set-associative cache

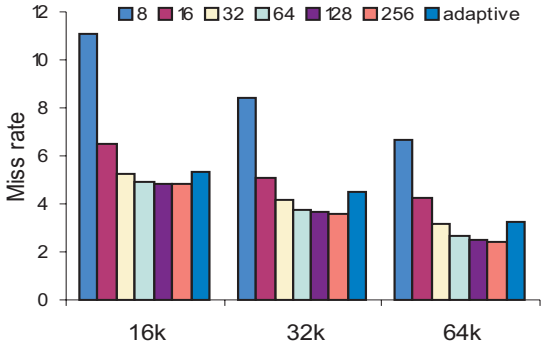

Figure 80: TURB3D: L1 direct-mapped cache

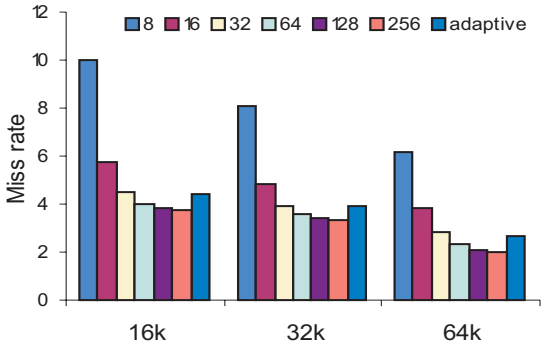

Figure 81: TURB3D: L1 2-way set-associative cache

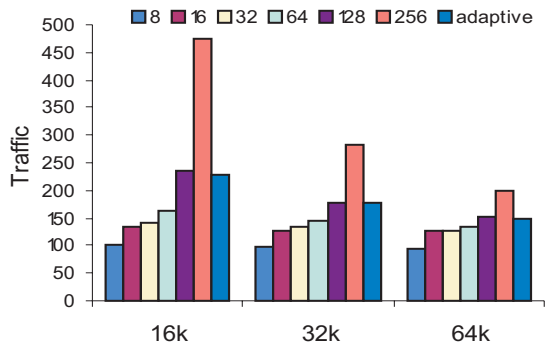

Figure 82: APPLU: L1 direct-mapped cache

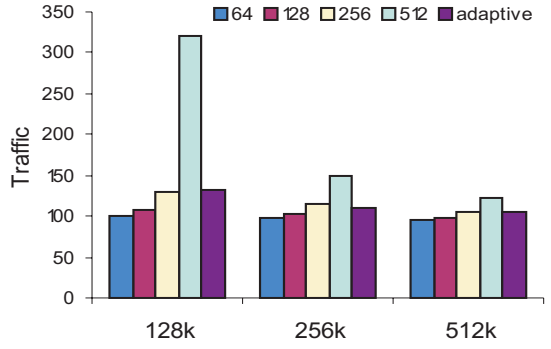

Figure 83: APPLU: L2 direct-mapped cache

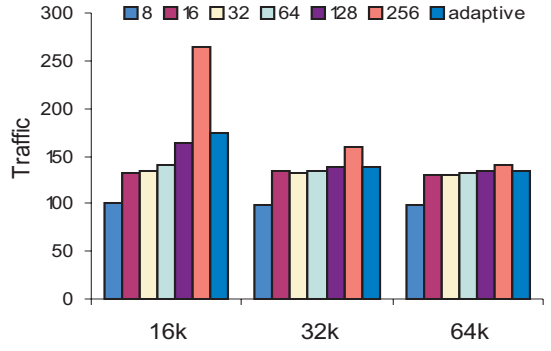

Figure 84: APPLU: L1 2-way set-associative cache

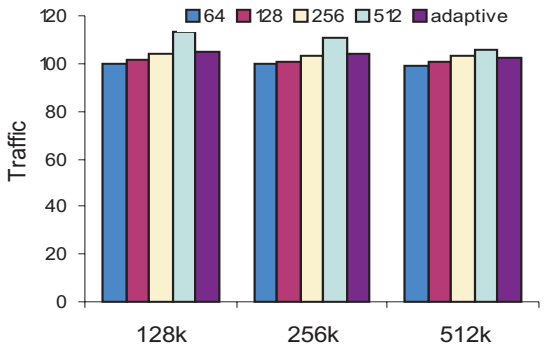

Figure 85: APPLU: L2 2-way set-associative cache

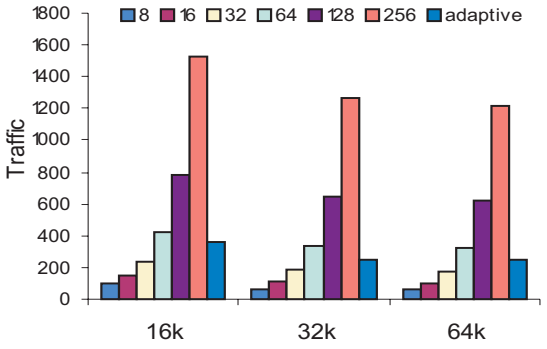

Figure 86: APSI: L1 direct-mapped cache

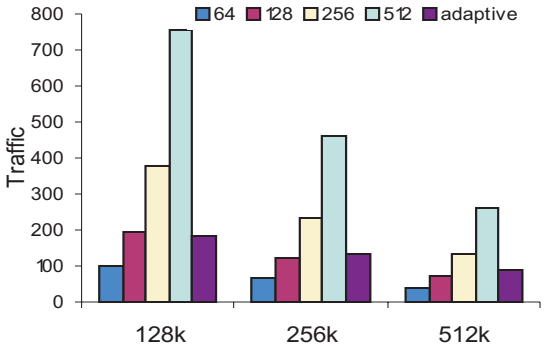

Figure 87: APSI: L2 direct-mapped cache

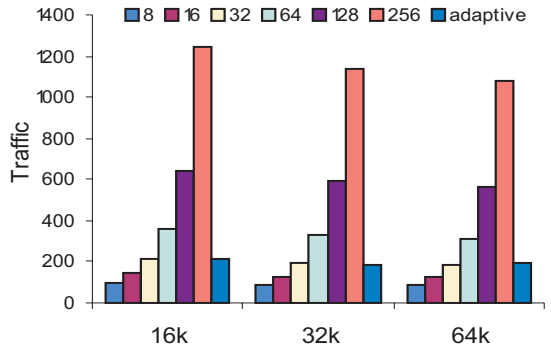

Figure 88: APSI: L1 2-way set-associative cache

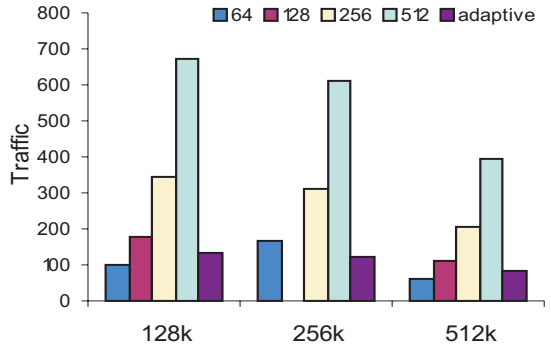

Figure 89: APSI: L2 2-way set-associative cache

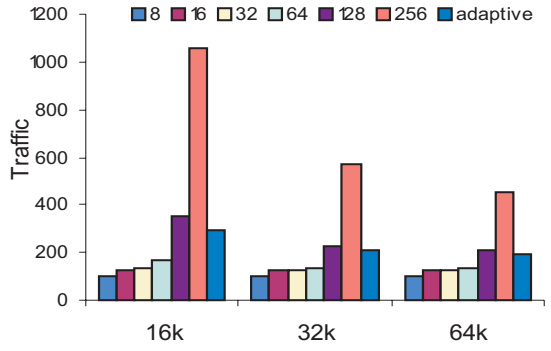

Figure 90: HYDRO2D: L1 direct-mapped cache

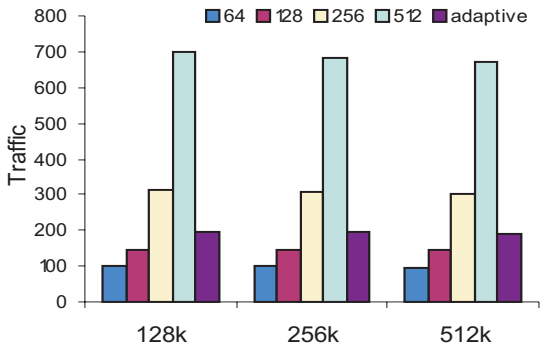

Figure 91: HYDRO2D: L2 direct-mapped cache

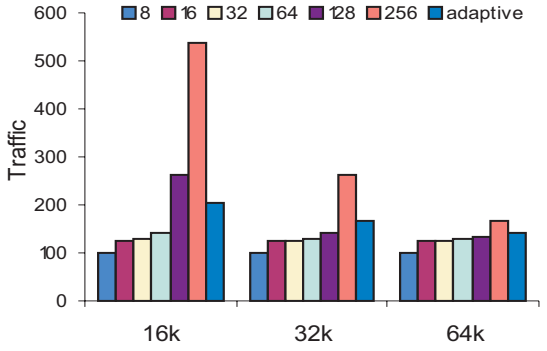

Figure 92: HYDRO2D: L1 2-way set-associative cache

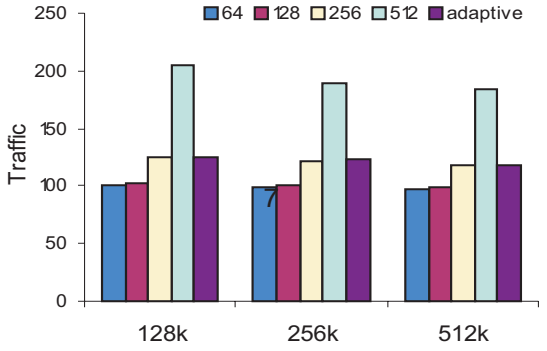

Figure 93: HYDRO2D: L2 2-way set-associative cache

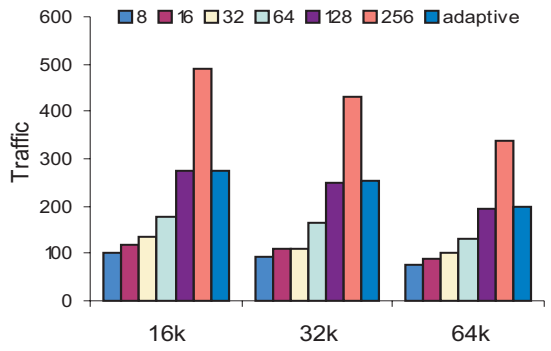

Figure 94: MGRID: L1 direct-mapped cache

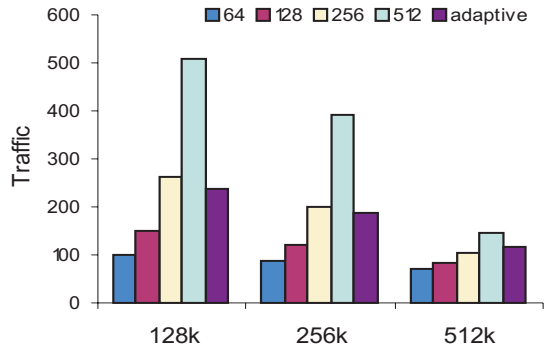

Figure 95: MGRID: L2 direct-mapped cache

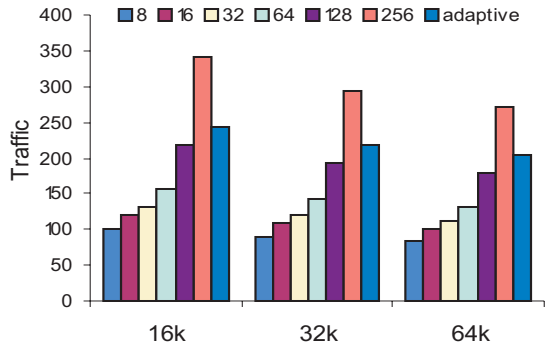

Figure 96: MGRID: L1 2-way set-associative cache

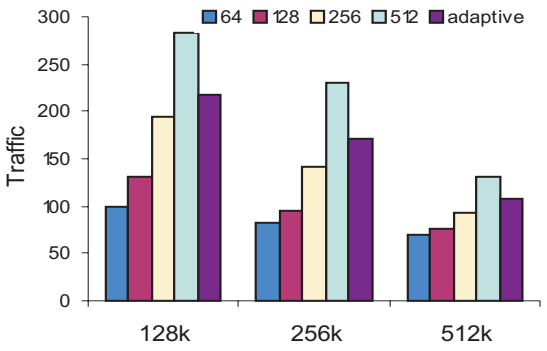

Figure 97: MGRID: L2 2-way set-associative cache

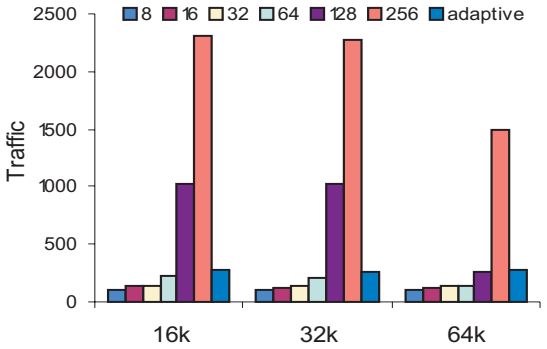

Figure 98: SU2COR: L1 direct-mapped cache

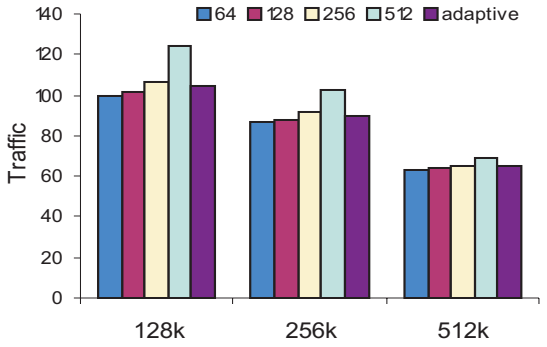

Figure 99: SU2COR: L2 direct-mapped cache

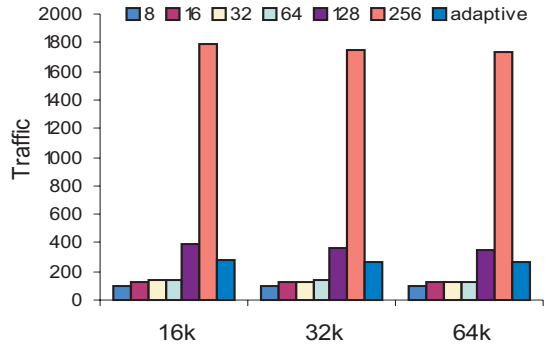

Figure 100: SU2COR: L1 2-way set-associative cache

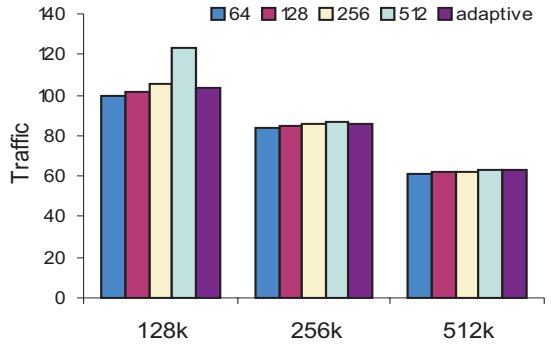

Figure 101: SU2COR: L2 2-way set-associative cache

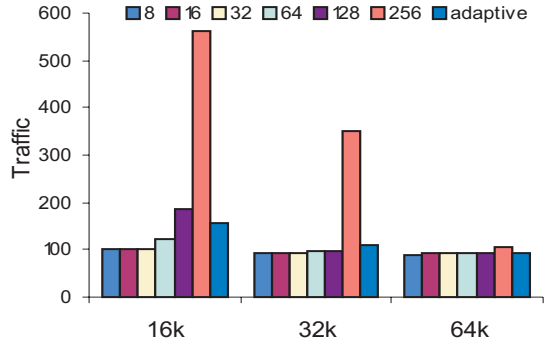

Figure 102: SWIM: L1 direct-mapped cache

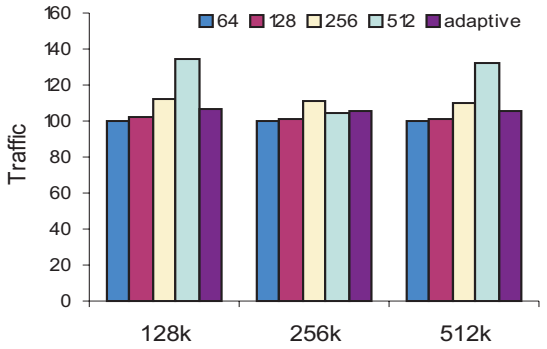

Figure 103: SWIM: L2 direct-mapped cache

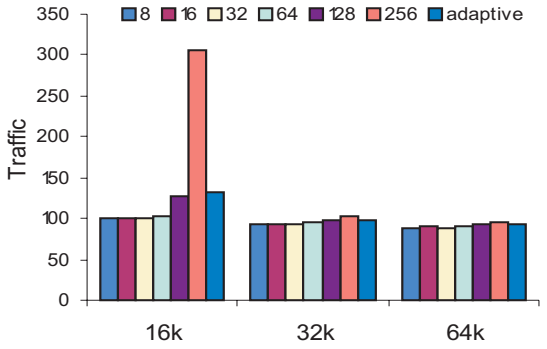

Figure 104: SWIM: L1 2-way set-associative cache

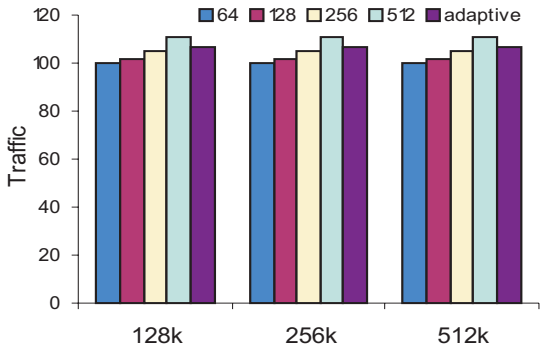

Figure 105: SWIM: L2 2-way set-associative cache

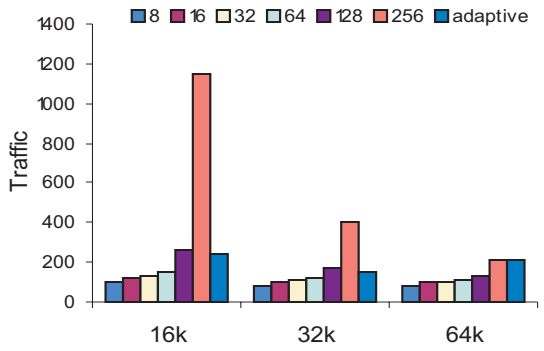

Figure 106: TOMCATV: L1 direct-mapped cache

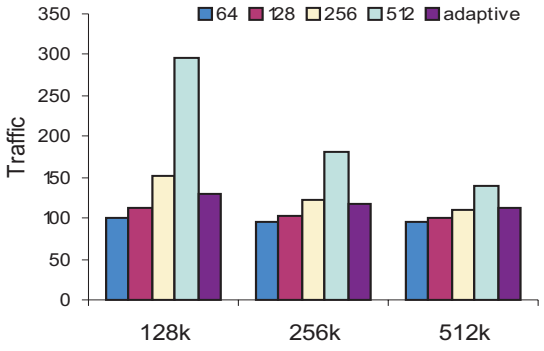

Figure 107: TOMCATV: L2 direct-mapped cache

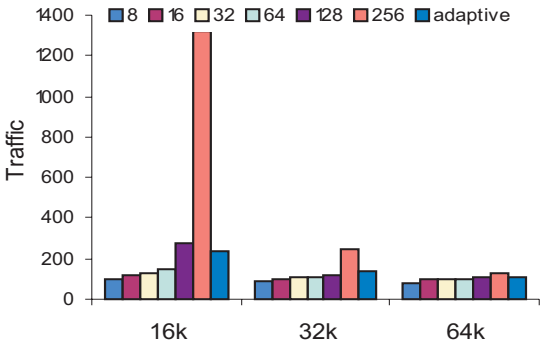

Figure 108: TOMCATV: L1 2-way set-associative cache

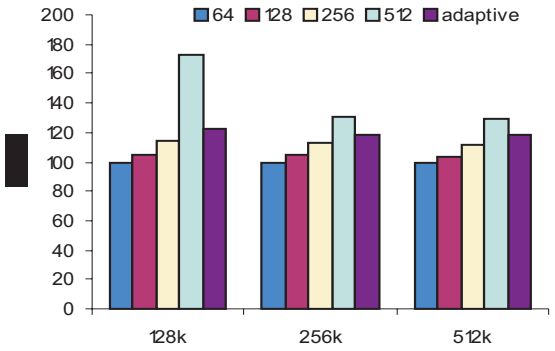

Figure 109: TOMCATV: L2 2-way set-associative cache

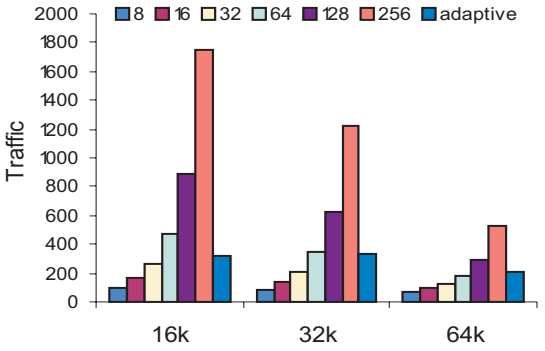

Figure 110: WAVE: L1 direct-mapped cache

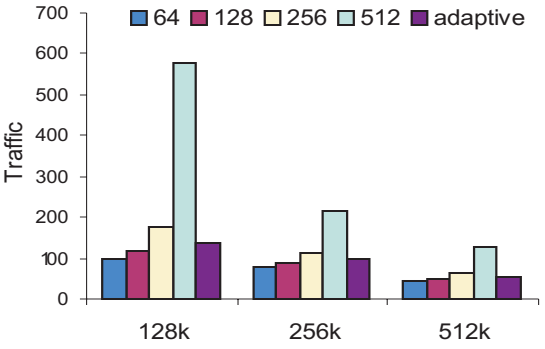

Figure 111: WAVE: L2 direct-mapped cache

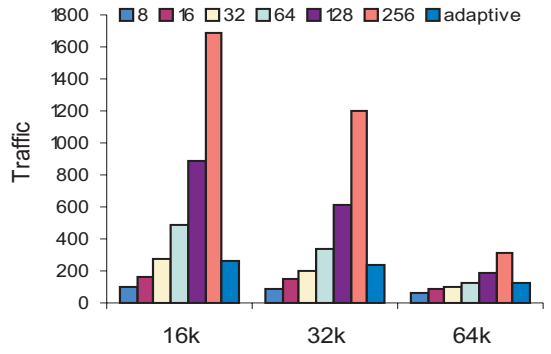

Figure 112: WAVE: L1 2-way set-associative cache

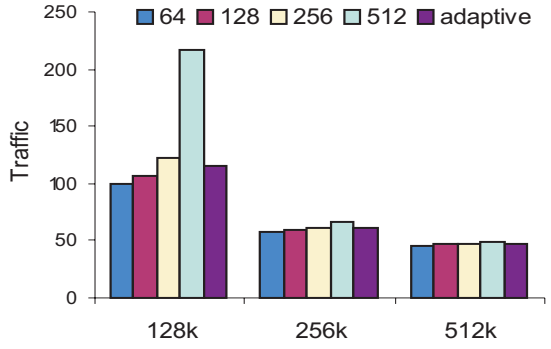

Figure 113: WAVE: L2 2-way set-associative cache

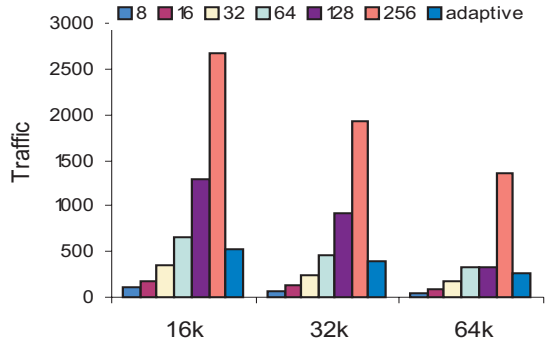

Figure 114: COMPRESS: L1 direct-mapped cache

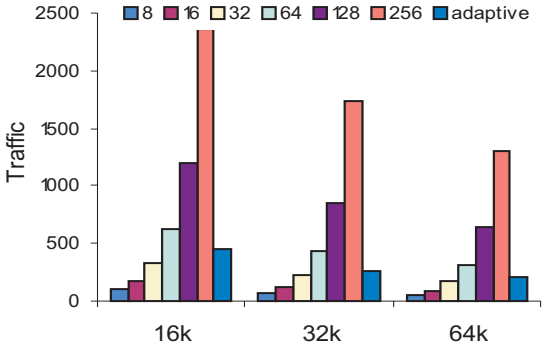

Figure 115: COMPRESS: L1 2-way set-associative cache

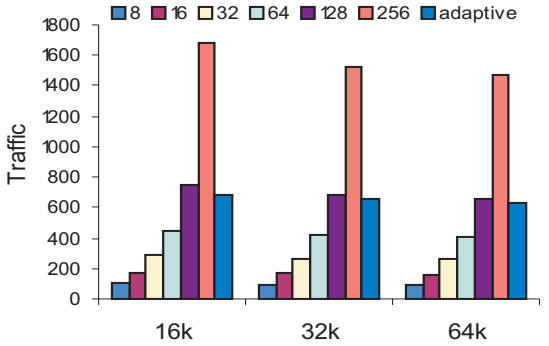

Figure 116: FPPPP: L1 direct-mapped cache

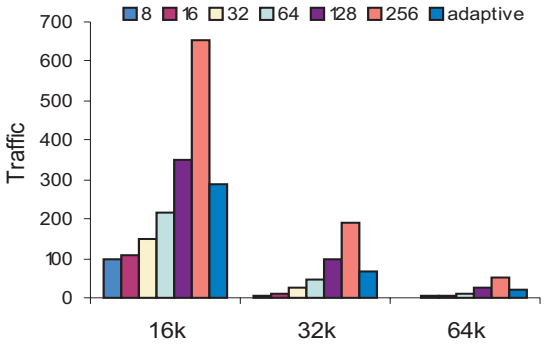

Figure 117: FPPPP: L1 2-way set-associative cache

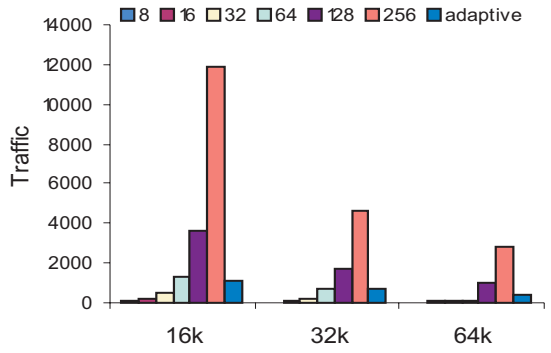

Figure 118: GO: L1 direct-mapped cache

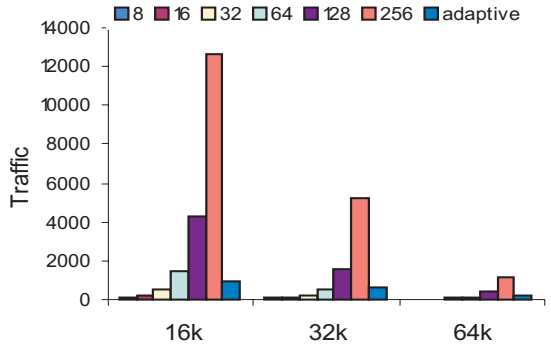

Figure 119: GO: L1 2-way set-associative cache

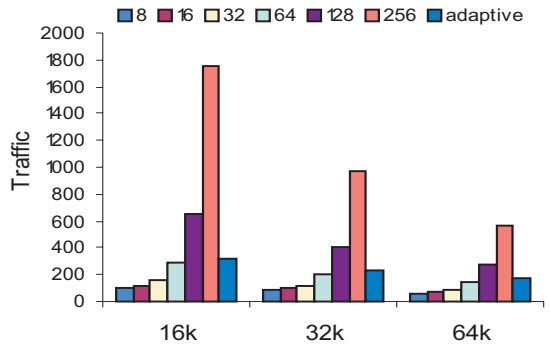

Figure 120: IJPEG: L1 direct-mapped cache

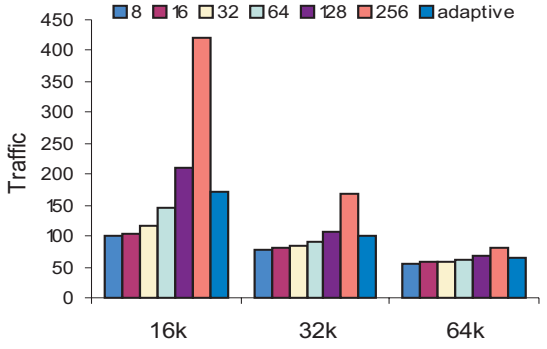

Figure 121: IJPEG: L1 2-way set-associative cache

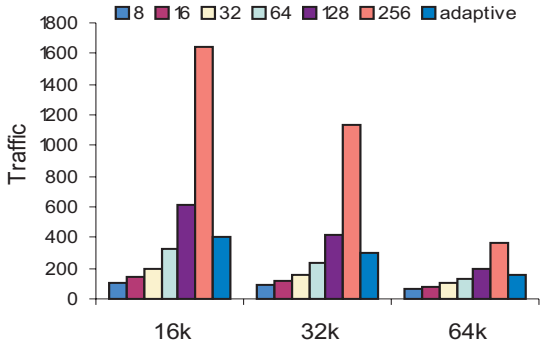

Figure 122: LI: L1 direct-mapped cache

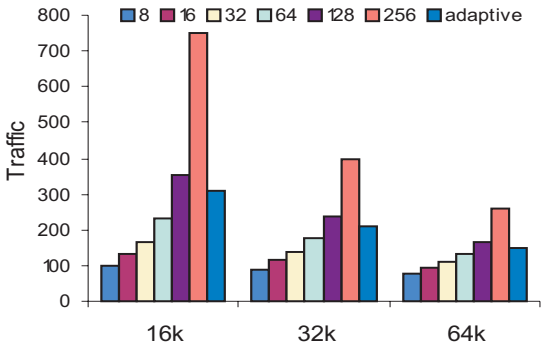

Figure 123: LI: L1 2-way set-associative cache

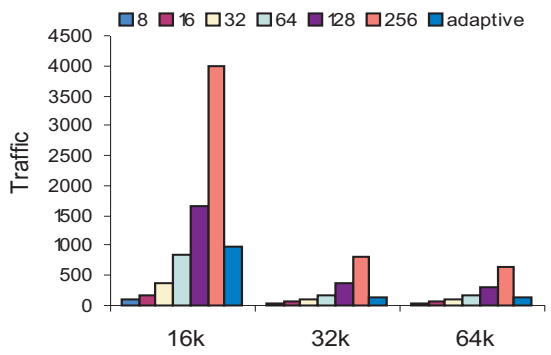

Figure 124: M88SIM: L1 direct-mapped cache

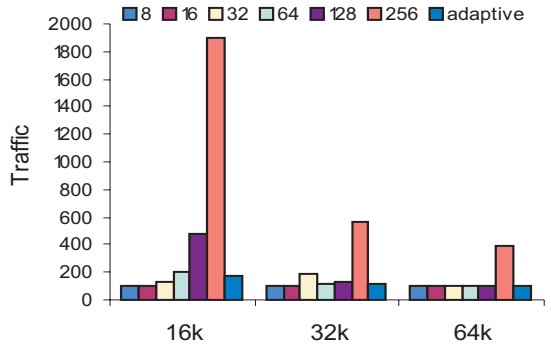

Figure 125: M88SIM: L1 2-way set-associative cache

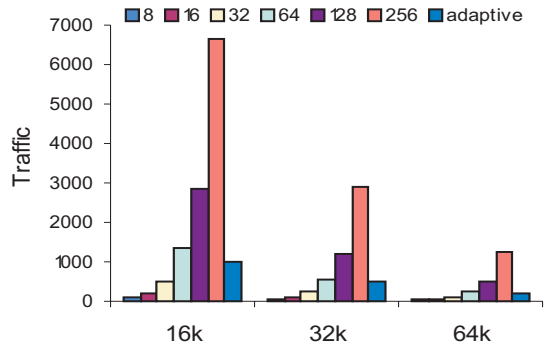

Figure 126: PERL: L1 direct-mapped cache

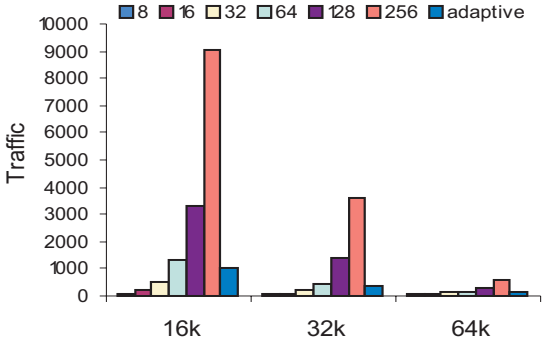

Figure 127: PERL: L1 2-way set-associative cache

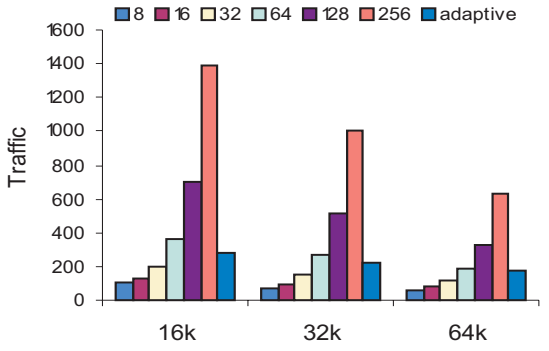

Figure 128: TURB3D: L1 direct-mapped cache

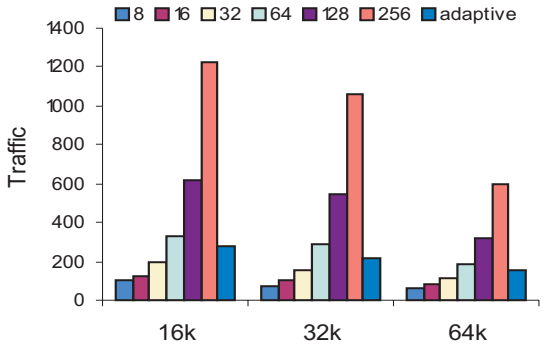

Figure 129: TURB3D: L1 2-way set-associative cache

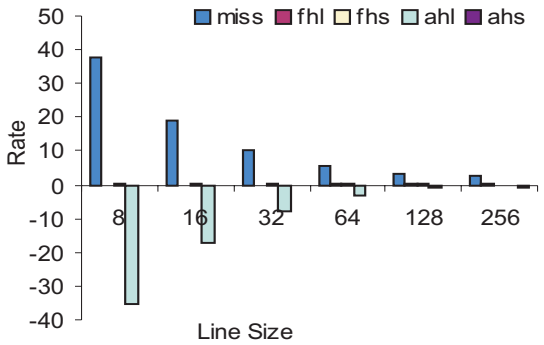

Figure 130: APPLU: L1 32K performance analysis graph

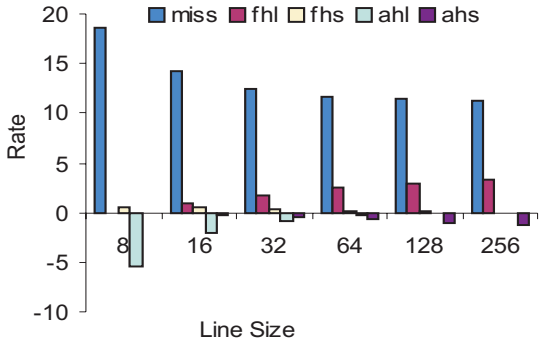

Figure 131: APSI: L1 32K performance analysis graph

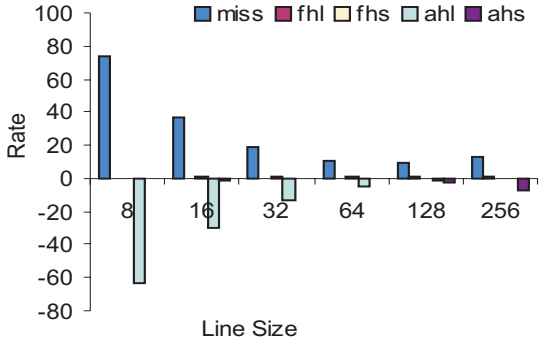

Figure 132: HYDRO2D: L1 32K performance analysis graph

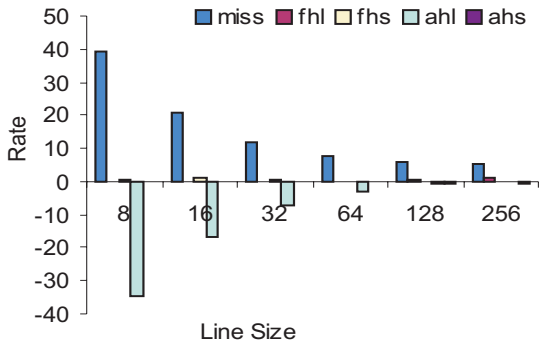

Figure 133: MGRID: L1 32K performance analysis graph

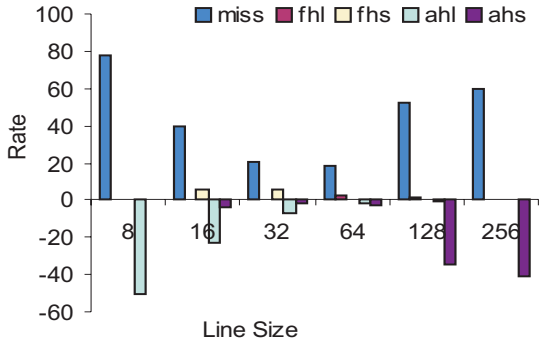

Figure 134: SU2COR: L1 32K performance analysis graph

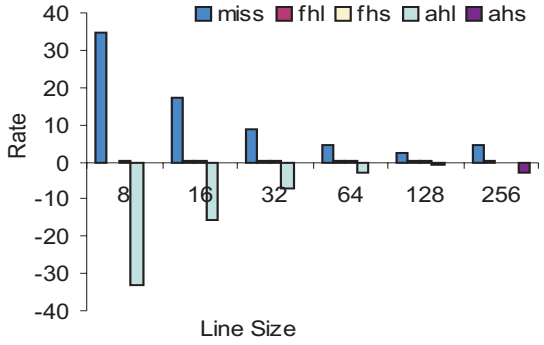

Figure 135: SWIM: L1 32K performance analysis graph

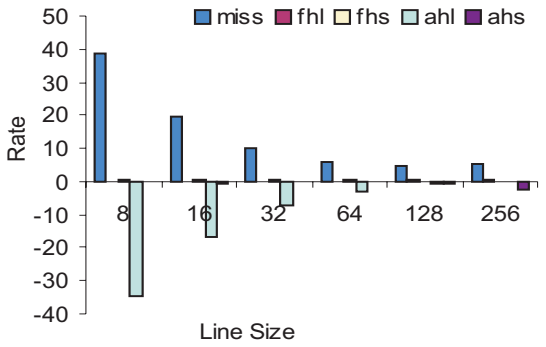

Figure 136: TOMCATV: L1 32K performance analysis graph

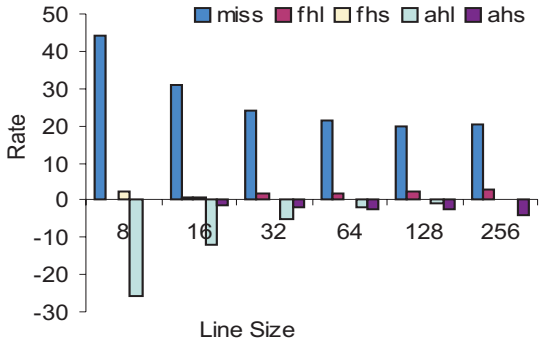

Figure 137: WAVE: L1 32K performance analysis graph

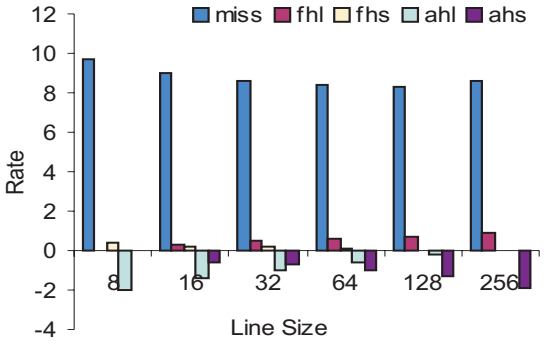

Figure 138: COMPRESS: L1 32K performance analysis graph

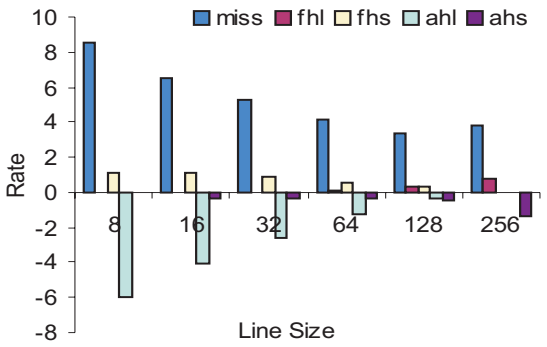

Figure 139: FPPPP: L1 32K performance analysis graph

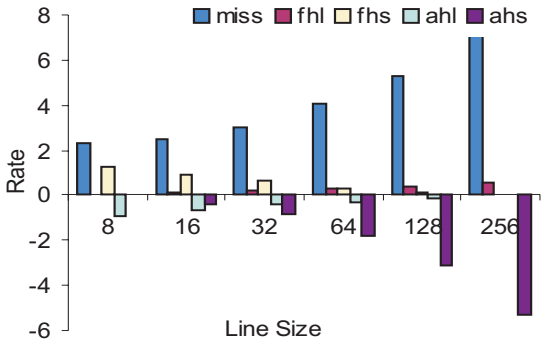

Figure 140: GO: L1 32K performance analysis graph

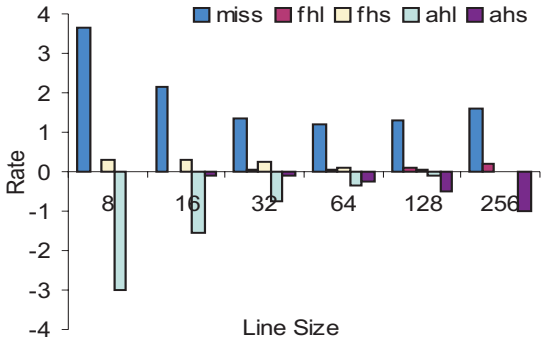

Figure 141: IJPEG: L1 32K performance analysis graph

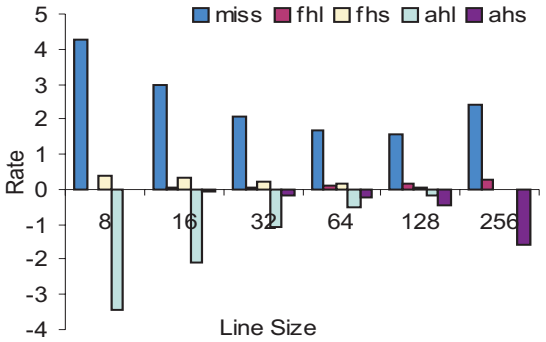

Figure 142: LI: L1 32K performance analysis graph

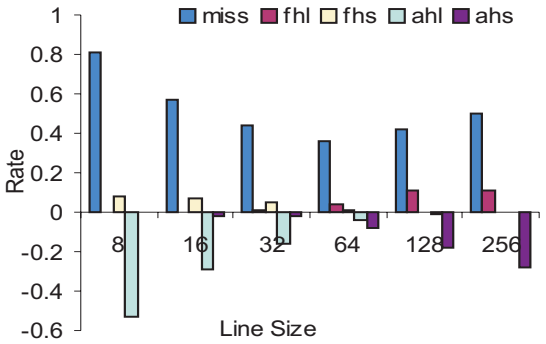

Figure 143: M88SIM: L1 32K performance analysis graph

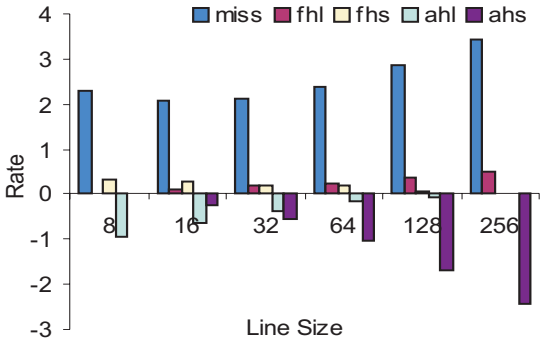

Figure 144: PERL: L1 32K performance analysis graph

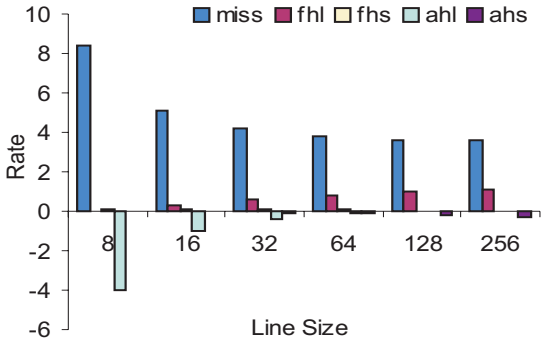

Figure 145: TURB3D: L1 32K performance analysis graph

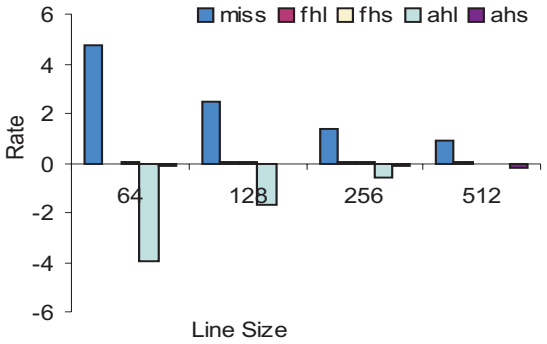

Figure 146: APPLU: L2 256K performance analysis graph

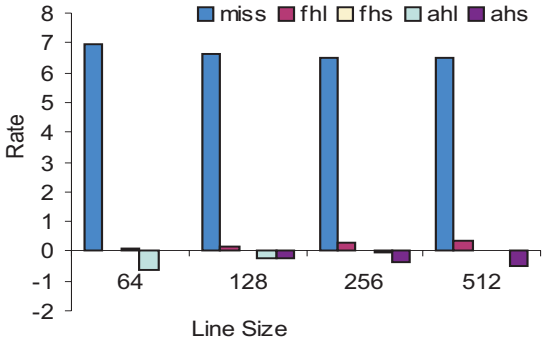

Figure 147: APSI: L2 256K performance analysis graph

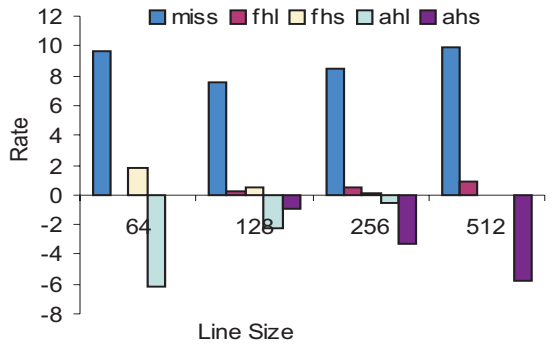

Figure 148: HYDRO2D: L2 256K performance analysis graph

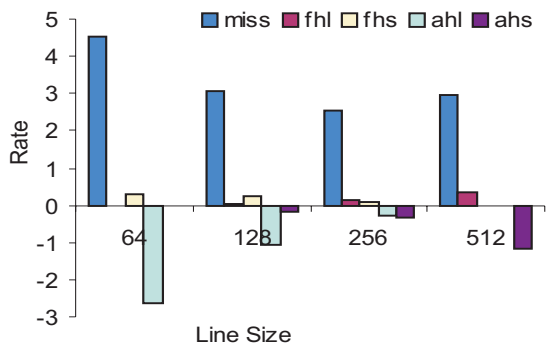

Figure 149: MGRID: L2 256K performance analysis graph

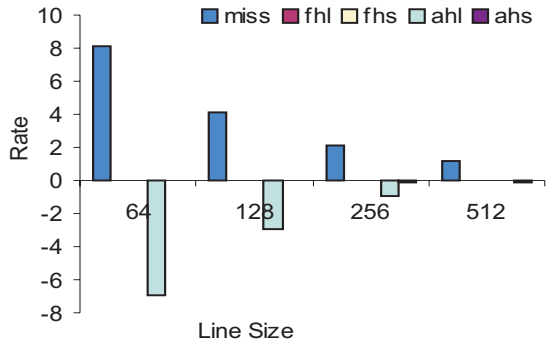

Figure 150: SU2COR: L2 256K performance analysis graph

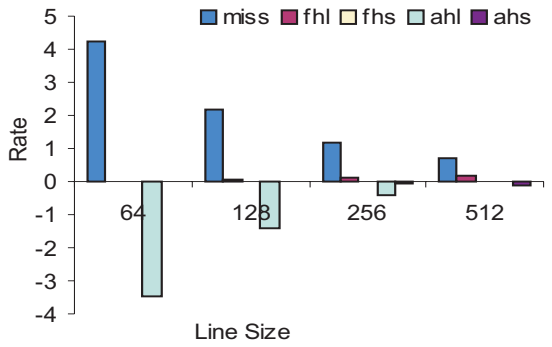

Figure 151: SWIM: L2 256K performance analysis graph

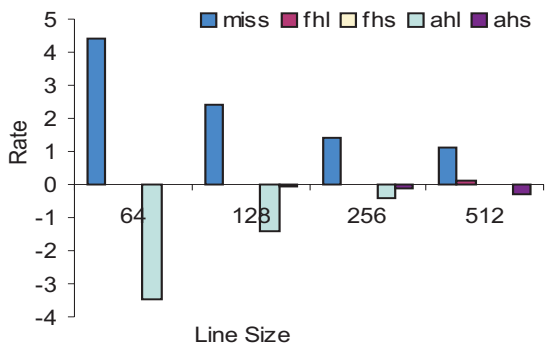

Figure 152: TOMCATV: L2 256K performance analysis graph

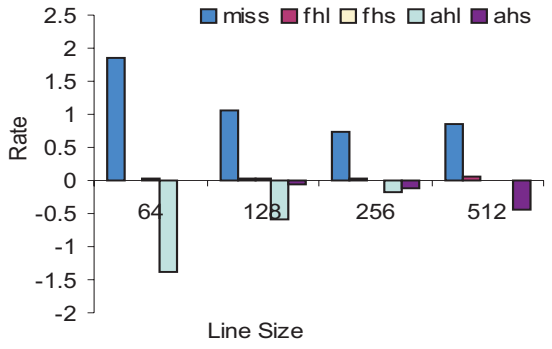

Figure 153: WAVE: L2 256K performance analysis graph