No inclusion in multi level caches

by

Bharath Vasudevan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Akhilesh Tyagi (Major Professor)
Gyungho Lee
Zhao Zhang
Gurpur Prabhu

Iowa State University

Ames, Iowa

2003

Graduate College
Iowa State University

This is to certify that the master's thesis of

Bharath Vasudevan

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to all who helped me in various aspects of my research and writing this thesis. First and foremost I would like to thank Dr. Gyungho Lee for his encouragement and backing, and stimulating suggestions that made working on my thesis an exciting experience. His expert guidance helped me gain very useful insights into the area of Computer Architecture, which proved to be an invaluable contribution to my thesis. I would also like to thank Dr Akhilesh Tyagi for helping me with my doubts. I would like to thank my other research group members for participating in useful discussions and for their invaluable suggestions.

I would also like to thank my friends for standing by me in good and bad times during my stay at Iowa State University.

Finally I would like to thank my parents and sister for their support and understanding.

ABSTRACT

Inclusive property in multi-level cache has been the norm in most processor architectures. Nevertheless, recent trends in cache implementations call for a reexamination of this issue. This thesis analyzes and evaluates the traditional inclusive scheme, no-inclusion scheme and mutual exclusion scheme. Using a Simple Scalar-based simulation and the SPEC2000 benchmark, it is been shown that the no-inclusion scheme, one of the non-inclusion schemes, provides the best performance. Further the thesis proposes two techniques to optimize the no inclusion scheme by selectively writing back data from L1 to L2. The first optimization filters out stack data that are unlikely to be accessed again immediately, and the second one filters out non-stack data of poor temporal locality. The two techniques not only reduce the L1-L2 traffic but also improve the efficiency of L2 cache as a backup storage. The simulation results show that these optimizations may reduce the main memory accesses by up to 23% and improve the performance of the no-inclusion scheme by up to 9%.

1 INTRODUCTION

Inclusion property in multi-level caches [2], i.e. low level cache contents are a subset of higher level cache contents, has been the norm in most processor architectures. One major reason for it is that with the inclusion property, cache coherency control causes less interference to low-level cache activity in multiprocessor environment. But with logical and physical limitations imposed on low-level cache (L1) sizes, other approaches may be more appealing as they provide less pollution at high level caches (L2). Also, L2 caches are increasingly residing on chip along with L-1 cache and processor core. When both L1 and L2 caches are on-chip, the size limit imposed onto L2 cache makes duplication of L1 data in L2 a significant overhead. Furthermore, the high transfer rate between the on-chip L1 and L2 caches makes L1 miss penalty serviced by L2 less critical. Reducing off-chip access by a more effective L2 cache becomes appealing. Not enforcing the inclusion property may be able to offer significant reduction in off-chip accesses as it naturally provides more back up capacity in L2 cache.

There have been two approaches proposed for multi-level caches without inclusion property. In mutual-exclusion approach [9], the contents of L1 and L2 caches are maintained mutually exclusive. A miss in L1 followed by hit in L2 results in either swapping of blocks between L1 and L2 or loading of the block into L1 and invalidation of the block in L2. All blocks replaced from L1 are written back to L2. With no-inclusion approach [10], a miss in both L1 and L2 results in the block being loaded into L1, replacing a block from L1. A block

replaced from L1 has to be written (back) to L2 if it is dirty or not in L2. This requires a bit for each block in L1 to indicate if it is in L2 cache. On replacement from L2 a signal is sent to L1 so that if that block is there in L1 then the bit indicating if the block is in L2 be reset, which is also necessary for the inclusion property.

Without inclusion property enforced, to avoid interference of coherence traffic with L1 accesses, the non-inclusion schemes, no-inclusion and mutual-exclusion, has space overhead in terms of a duplicate copy of the L1 data tag array in the L2 cache interface. All accesses to the L2 and I/O write go through this interface. Or as in the Alpha EV6 [4] there can be a triplicate copy of L1 and L2 tags. This makes the major benefit of the inclusion property with little merit. In this thesis, performance potential of the three approaches in two level data cache environments is compared. With increasing clock speeds for pipelined processor core, the need for faster L1 access is increasing thereby making it difficult to employ many of the cache performance improvement techniques. Some popular choices for L1 cache design to achieve a very fast access times includes: 1) a small direct mapped cache with its size equal to a page size to avoid the address translation delay, 2) Relatively large sized blocks with virtual address indexed caches [6], which removes address translation and needs TLB access only on a cache miss, 3) pseudo associative cache to reduce conflicts with pipelined access. With these three design choices for L-1 cache, the performance potential of the no-inclusion and the mutual-exclusion approaches over the usual inclusion approach has been evaluated. In the experiments using the Simple Scalar [11] simulator running Spec2000 [14] benchmarks (results in chapter 4), no-inclusion approach is found to have the most potential, outperforming others, up to more than 20% in committed IPC (instructions per cycle). In the case of the mutual-exclusion, all blocks replaced from L1 cache have to be

written back to L2 cache and so the traffic between L1 and L2 caches is very high, thereby reducing the performance advantage obtained by greater capacity. The no-inclusion strikes a balance between the reduced L1-L2 traffic of the inclusion approach and greater on-chip area utilization of the mutual-exclusion approach. However, the no-inclusion approach also suffers from negative effects of traffic between L1 and L2 caches, especially if L1 cache size is small, which limits its potential. The copy back traffic contends for the bus with the data brought in for the program execution.

In chapter 5, a scheme to reduce the L1-L2 traffic and increase the backup space in L2 for useful data by selectively writing back data from L1 to L2 is been proposed. The scheme classifies the memory accesses based on the region accessed into stack and non-stack and based on the type of cache miss into conflict and capacity miss. The basic idea is that stack accesses have very good temporal locality and hence any data in the stack regions will be accessed over a small period of the program execution time only [12]. The scheme is enhanced by a simple hardware scheme that classifies misses as conflict and capacity misses. It is based on observation that data replaced from the L1 cache due to a capacity miss is less likely to be accessed in the immediate future [1]. These two aspects of program execution are used to selectively write back stack data form L1 to L2 in the no-inclusion approach. A load address table (LAT) that captures those loads that cause frequent misses is used to determine whether non-stack data has to be written back to L2 on replacement. The idea is that the blocks got into the cache by these frequently faulting instructions will have less locality and hence would not be accessed again immediately [7]. The scheme was implemented and the benchmarks simulated. The results presented in chapter 5 show that the number of accesses to the main memory decreases rather significantly (up to 23%) and hence the performance

increases. The experimental results with Spec2000 programs show that the no-inclusion approach improves significantly (up to 9%).

In chapter 2 the background work has been presented. In Chapter 3 the details about two-level cache contents management approaches inclusion, mutual-exclusion, and no-inclusion management are discussed. In Chapter 4, the reference model used for simulations and the results of the simulations for the different approaches are presented. New proposed scheme to improve the performance of the no-inclusion approach and the results are presented in Chapter 5, followed by conclusion and future work in Chapter 6.

2 BACKGROUND

Nowadays high-performance processors have employed two-level or three-level on-chip caches. For simplicity, we assume two-level cache in the following discussion. The L1 cache is usually small and fast so as to match the speed of processor core, while the L2 cache is relatively large so as to reduce the number of expansive DRAM accesses [3]. In a typical two-level cache [17], the L1 cache is virtually tagged and physically indexed to avoid TLB translation delay being added to L1 hit time, while the L2 cache is physically tagged to avoid the complicated synonym issue in virtual cache.

The inclusion property for a two-level cache states that any data cached in the L1 cache is also cached in the L2 cache. With this property, cache coherence and I/O traffic may only probe L2 cache in most frequent cases. If the data is not found in L2, or if the data is found but a bit in the block indicates the data is not in L1 cache, the L1 cache will be not interfered. The cache control has been considered to be less complex than that in a non-inclusive cache. Inclusion property was first used when L2 cache was put off-chip and was many folds larger than L1 cache, thus the duplication of L1 contents in L2 cache did not pose a serious issue in performance.

Baer and Wang [2] studied the necessary and sufficient conditions for enforcing the inclusion property for set-associative cache. Among others, they found that the associativity of L2 cache must be equal to or larger than the product of (1) the sum of the associativities of all L1 caches and (2) the ratio of L2 block size to L1 block size. As for replacement, the L1

cache needs to notify the L2 cache of the blocks being replaced, and the L2 cache should replace those blocks before others in the cache set. Wang et al [17] further found that with L1 virtual cache and L2 physical cache, the L2 associativity should be greater than the product of (1) the ratio of L1 size to page size and (2) the ratio of L2 block size to L1 block size.

Cache coherence may also be maintained without the enforcement of inclusion property. For example, Alpha 21264 processor employs separate, physically tagged tag storage for L1 data cache to help maintain cache coherence and to protect the L1 cache from being interfered by cache coherence and I/O traffic. This approach requires only small extra storage. Jouppi and Wilton [3] proposed a scheme called two-level exclusive caching and evaluate it on a single-issue pipelined processor with direct mapped L1 cache. They found it was consistently better than the inclusion scheme for SPEC89 programs. That scheme is similar to the no inclusion scheme. In this study, we examine several schemes on modern processors with a wider range of L1 cache configuration, and have proposed and evaluated two techniques to optimize the no inclusion scheme.

3 MULTI LEVEL CACHE CONTENTS

In most modern processors there is a larger second level cache added to the system to improve overall performance. The two-level cache improves performance by effectively lowering the first-level cache access time and miss penalty. Inclusion property [3], i.e. L1 holds a subset of L2 contents, has been traditionally enforced in multi-level data caches as it is assumed to provide a couple of benefits. First, the inclusion allows data access to L1 to proceed simultaneously with probing of cache contents from coherency traffic. Second, it is considered to require less complex control. Further, since L2 cache has been usually off chip and many folds (even tens of times) bigger than L1 cache, the overhead due to duplication of L1 at L2 does not pose a serious capacity issue. However, these advantages seem a claim not well based on facts. A separate copy of L1 tags at L2 interface can easily provide less interference of coherency traffic to L1 access. As seen in Alpha 21264, separate tag storage for additional copy of L1 and L2 tag requires not only little extra storage but also little extra control complexity (most of its complexity is due to virtual address indexed cache). Also, the inclusion does not allow less complex control as intuition may suggest. To enforce the inclusion property, L1 and L2 need to communicate each other to make it sure that L1 is a subset of L2. Also, it limits the choice of L1 and L2 design parameters [15]. L1 cache size is usually limited due to some physical and logical reasons. To avoid address translation delay, L1 size is often limited to (page size x set associativity). Furthermore, with increasing processor clock cycle, L1 size needs to be limited rather severely if L1 needs to be accessed

in one or two cycles. With smaller L1 size and ever increasing number of transistors available on-chip, a relatively modest sized L2 cache appears to be a good resource to be on a chip along with processor core and L1 cache. When both L1 and L2 caches are on-chip resources, transfer rate between them can be very high and also the modest size of L2 makes duplication of L1 data a significant overhead. Many data missing in L1 will also miss in L2. So in this case L2 may add more to the delay between L1 and off-chip access than reducing the number of off-chip accesses. Reducing off-chip access by more effective L2 cache design becomes more appealing. Not enforcing the inclusion property may be able to offer significant reduction in off-chip accesses as it naturally provides more back-up capacity in L2 cache.

While most cache related work assume the inclusion, there has been no comprehensive performance study comparing the inclusion with other possible approaches, mutual-exclusion and no-inclusion. In this thesis, the performance potential of the three approaches for multi-level data cache contents management has been presented in terms of

- Instructions Per Cycle (IPC) – Average number of instructions of the program executed every cycle.
- Local Miss Rate – local miss rate of a cache is the number of misses experienced by the cache divided by the number of incoming references.
- Global miss rates – It is the number of L2 misses divided by the number of references made by the processor. This is the primary measure of L2 cache. L2 cache is not measured using local miss rate because it may be difficult to determine the number of references made to it from L1.

To set up a reference for distinguishing one approach from another, the three approaches are described here.

3.1 Inclusion Management

Inclusion management corresponds to space inclusion between caches. L2 cache is a superset of L1 cache but need not be the most up-to-date copy. If a miss occurs in both L1 and L2, the missed block is copied from main memory into both L1 and L2. If L1 misses and L2 hits then data is got into L1 from L2. When we get a block into L1, an already existing block might have to be replaced. At a block replacement in L1, a dirty block is written back to L2 to update the L2 copy but a clean block not modified in L1 is discarded. Note while there are no replacement restrictions at L1, only blocks that are not in L1 can be replaced from L2 in order to maintain inclusion. In case there is no such block in the current set, we have to evict a block from L2 that is in L1. To maintain inclusion, the corresponding block is also evicted from the L1 cache. If the block is dirty, it is written back otherwise discarded. Note that there can be a chain of replacements triggered by a single cache misses.

The inclusion scheme requires a present-in-L1 bit in L2 cache for each block to indicate if that block is in L1 or not. This bit is set when we get data into L1 cache from L2 and reset when we write data back from L1 to L2. In case of using write buffers between L1 and L2, we will have to consider blocks in them for maintaining inclusion. Every time a block is replaced from L1 then a signal has to be sent to the L2 cache so that the present-in-L1 bit in the corresponding block in L2 is reset. Also all L1 references have to be passed to L2 so that it can update its LRU otherwise inclusion would not be possible. Also, as well known, the inclusion imposes some limitation in choosing block size and associativity. The crucial

requirement is that the number of L2 sets has to be greater than the number of L1 sets, irrespective of L2 associativity. If this were not true, multiple L1 sets would depend on a single L2 set for backing store. Another requirement is that the L2 associativity should not be less than that of L1 irrespective of the number of sets. If L1 associativity is greater than that of L2 then it is possible that several references to an L1 set exists that would hit in the same L2 entry. This would result in replacement of valid L1 lines before L1 starts to replace.

3.2 Non-inclusion Management

No inclusion essentially treats the L1 and L2 caches together as one large cache, with the advantage of maintaining the most recently used portion of the cached data in the L1 cache and thus closest to the processor. The contents in L1 and L2 depend on the access pattern at L1. In certain cache memory configurations, the contents of L1 and L2 will be mutually exclusive and in some it could be totally inclusive. A miss in both levels results in the block being loaded into L1 only, replacing a block from L1. If L1 misses and L2 hits, the block is got into L2 from L1. The block replaced from L1 has to be written back if it is dirty or not in L2. This requires a present-in-L2 bit for each block in L1 cache to indicate if it is in L2 cache. When a block is fetched into L1 from main memory the bit is reset and is set if the block is from L2 cache. Unlike in inclusion scheme, a write into L2 can result in a miss causing a block from L2 to be replaced. For both levels there are not any replacement restrictions. On replacement from L2, a signal is sent to L1 so that if that block is there in L1 then the bit indicating if the block is in L2 is reset. Unlike in inclusion scheme there is no need for signaling from L1 to L2 even when clean blocks are replaced.

The non-inclusion scheme has space overhead in terms of a duplicate copy of the L1 data cache tag array in the L2 cache interface. All accesses to the L2 and I/O writes go through this interface. This copy allows probing for addresses in the data cache without interrupting the load/store processing. Note that there are no replacement restrictions either in L1 or in L2, unlike in the inclusion and the mutual-exclusion, which allows the no-inclusion less complex control then for inclusion or mutual-exclusion. Simple no-inclusion requires that the L1 and L2 block sizes be the same but different block sizes for L1 and L2 can be supported by having extra valid bits at the granularity of the smaller block size (L1 block size will be normally smaller). The cache design then becomes similar to that of sectored caches. In sectored caches, each line is broken into transfer units (a sub-line that represents one access from the cache to the memory). When a miss occurs only the transfer unit is brought into the tag array though the missed line is entered into the directory. Valid bits indicate the status of the sub-lines. When a subsequent access is made to another sub-line in the newly loaded line, it is brought into the cache. To support no-inclusion, L2 will be organized as a sectored cache and data will be brought in at the granularity of L1's block size.

3.3 Mutual Exclusion

Mutual Exclusion maintains complete exclusion between the contents of L1 and L2, thereby achieving more capacity altogether. Useful information in the caches equals their sum of the sizes. In this approach, L2 cache can be considered a victim cache for L1 cache: Data replaced from L1 are written back to the victim buffer (L2). When data misses in the L1 cache, first the victim buffer (L2) is accessed and if data is there it is brought into L1. A miss in both levels results in the block being loaded into L1 alone. A block that misses in L1 but

hits in L2 would result in swapping of the blocks between L1 and L2. The block that misses is swapped with the block that is chosen for replacement from the L1 cache. A miss in both levels would result in the missing block being loaded into the L1 cache alone. In case of L2 hit, the missed block and the block chosen for replacement from L1 cache may not map into the same L2 set. In this case, to maintain exclusion the data is brought into L1 from L2 and the L2 entry is invalidated. Invalid blocks would be formed in a L2 cache whenever the missed and the replaced block do not map to the same set. So the next time a block is written back into L2 we can use the invalid block, if there is any in the set to which the block written back maps. The replacement algorithm is modified to consider the valid bit associated with all blocks in L2 cache. This would in turn avoid write-backs from L2 to main memory, thereby decreasing the overall latency of the data fetch operation. The L2 incoming block can also cause another replacement from L2. In [9], it is claimed that the miss rate of the second-level cache is reduced with swapping and a system with swapping has a performance greater than that of a two-way set-associative cache because of its combination of miss rates and lower cache access time on a hit. Mutual exclusion has been considered mostly in the context of providing a small victim buffer between L1 and L2 caches. While extra L1 tag will allow probing of addresses for cache coherency protocol to proceed without interrupting L1 accesses as in inclusion, mutual exclusion has replacements restrictions, not only in L2 but also at L1. Also mutual exclusion requires extra control in the form of swap tag lines; swap data lines and swap/read lines connecting the L1 and L2 caches.

The simplest exclusion like in no inclusion method requires that the line size of the first and second level caches be the same. If different line sizes at different cache levels have to be supported then valid bits must be provided on the granularity of the smallest cache line.

When data is swapped from a larger cache line to a smaller cache line, the excess data is discarded. (If the cache is write-back and the data is dirty, the discarded dirty data must be queued to be transferred to the next lower level in the memory hierarchy off-chip.) When swapping from a smaller line to a larger line, the valid bits in the larger line corresponding to the data not provided from the smaller line must be turned off.

In the next chapter the performance potential of the three schemes in two level data cache environments is explored under different configurations. All experiments were carried out on the Simple scalar simulator running Spec2000 benchmarks.

4 PERFORMANCE POTENTIAL

As we review the details of the three approaches for managing multilevel cache contents in the previous section, the traditional inclusion is found to provide little advantage over other approaches without inclusion. Interference to L1 access due to coherency traffic can be avoided as in the inclusion by having extra tag copies, and the inclusion requires as complex control, if not more than, as other approaches need. In this section, we consider the performance potential of the three approaches.

Table 4-1: Simulation Parameters

Issue Width	4-way
RUU	32-entry
LSQ	32-entry
L1 Instruction	32Kb 2 way 64byte block
L1 data, L2 unified	Varied in the simulations
MSHR	8 entries
Write back Buffer	8 entries
Hit latencies	L1=3, L2=12, TLB hit –1 cycle, memory-80 cycles
TLB miss latency	30 cycle
Back Bus L1-L2	16 bytes every 1.5 CPU cycles
Memory Bus bandwidth	8 bytes every 1.5 CPU cycles
Branch predictor	Bi Modal 2048 entries
Branch mis-prediction latency	3 cycles

The Simple scalar simulation environment was used. The simulator was modified to resemble modern day microprocessors [4] by incorporating common features like deep pipelines, load store forwarding, a system bus between L2 and main memory and a back bus between L1 and L2, write buffers (we assume caches use write back policy) between

different levels in memory hierarchy. The presence of a backside bus allows copy back and L2 access from L1 operations to proceed without interference from memory accesses from L2. The data cache was made non-blocking with up to 8 outstanding misses. This requires a Miss Status Hold Register (MSHR). Both read and writes are non-blocking. Consistency problems arising as a result of a later read being needed before a previous write is buffered is resolved by checking the MSHR and write buffer. Also when writing a block to the write buffer, other read /write requests can proceed provided they are not a miss. To compare the different content relationship schemes we take the reference model shown in Figure 4-1 into which the schemes can be embedded. The first level of the memory hierarchy consists of an L1 data cache and L1 instruction cache. Separate instruction and data caches are common in modern processors as they offer the designer the possibility of significantly increased cache bandwidth potentially doubling the access capability. The L1 data cache is virtually indexed. There is an unified L2 cache at the second level. In the unified cache, the ratio of instruction to data working set elements changes during the execution of the program and is adapted to by the replacement policy. At the next level is the main memory. Buffers are provided to speed up inter level transfers. There are write back buffers between L1 and L2 and between L2 and main memory. The size of these buffers is generally small (8 or 16 entries). Separate TLB's are used for instruction and data. Table 4-1 gives the parameters used. Unless noted otherwise, we assume 3 cycle latency for L1 data cache and 12 cycle latency for L2 cache with 64 byte block size for both caches, the caches being direct mapped. The bimodal branch predictor with 2K entries was used.

4.1 Handling Cache Misses

Whenever the address being referenced is a miss in the L1 cache two actions are taken. One is the missed line is fetched from the higher levels in the memory hierarchy and the other being one of the current cache lines is replaced by the currently accessed line. In case of write back caches, the simplest way of handling cache misses is to first select the line to be replaced. If the line is dirty, it is written back to the next level and then the missing line is brought into the cache. Processing is resumed once the entire line has been brought into the cache. Write buffers are used to speed up the process. The way cache misses are handled in our simulations is by using a non-blocking cache. The cache has the extra hardware to allow the cache miss to be handled while the processor continues to execute. This allows the processor to continue executing instructions that do not depend on the line being currently brought in to the cache. The non-blocking data cache in our simulations can handle up to 8 outstanding misses. A 8 entry miss status holding registers (MSHR) which reserves space for outstanding cache misses is used. On every cache miss a entry is made in the miss status register. If there are no empty slots then the pipeline is stalled. Processing resumes only after an earlier memory request completes creating an empty slot in the miss register for the new cache miss. The MSHR supports coalescing so that multiple misses to the same line do not initiate multiple requests to lower levels of the memory hierarchy. We do not include such coalesced requests when calculating miss counts for our analysis. Also we assume that fetch bypass or wraparound load [5] is used, so the execution begins as soon as the first word has been accessed.

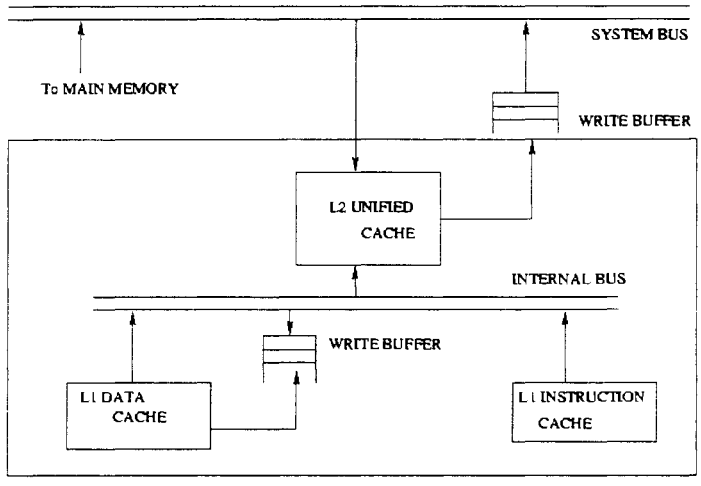

Figure 4-1: Simulation Model of the Super Scalar Processor Used

4.2 Performance Evaluation for Varying Cache Sizes

With increasing clock speeds for pipelined processor core, the need for faster L1 access is increasing thereby making it difficult to employ many of the cache performance improvement techniques. Some popular choices for L1 cache design to achieve a very fast access times includes: 1) a small direct mapped cache with its size equal to a page size to avoid the address translation delay, 2) direct mapped with relatively large sized blocks and virtual address indexed caches [6], which removes address translation and needs TLB access only on a cache miss, 3) pseudo associative cache to reduce conflicts with pipelined access.

The no-inclusion outperforms the other two for most benchmarks as shown in the results in Figure 4-2 for small caches. The performance of the exclusion scheme is worse than no-inclusion for most benchmarks but better than inclusion for some of the benchmarks.

For compute intensive benchmarks like bzip2 and art, the performance of the exclusion scheme is good as it has a better on chip coverage of the address space. Twolf and Vpr spend very less amount of the execution time in memory accesses as indicated by their low global miss rates [13]. So the advantage from increased coverage of address space in case of exclusion as well as no inclusion is less. The increased traffic between L1 and L2 is responsible for the low

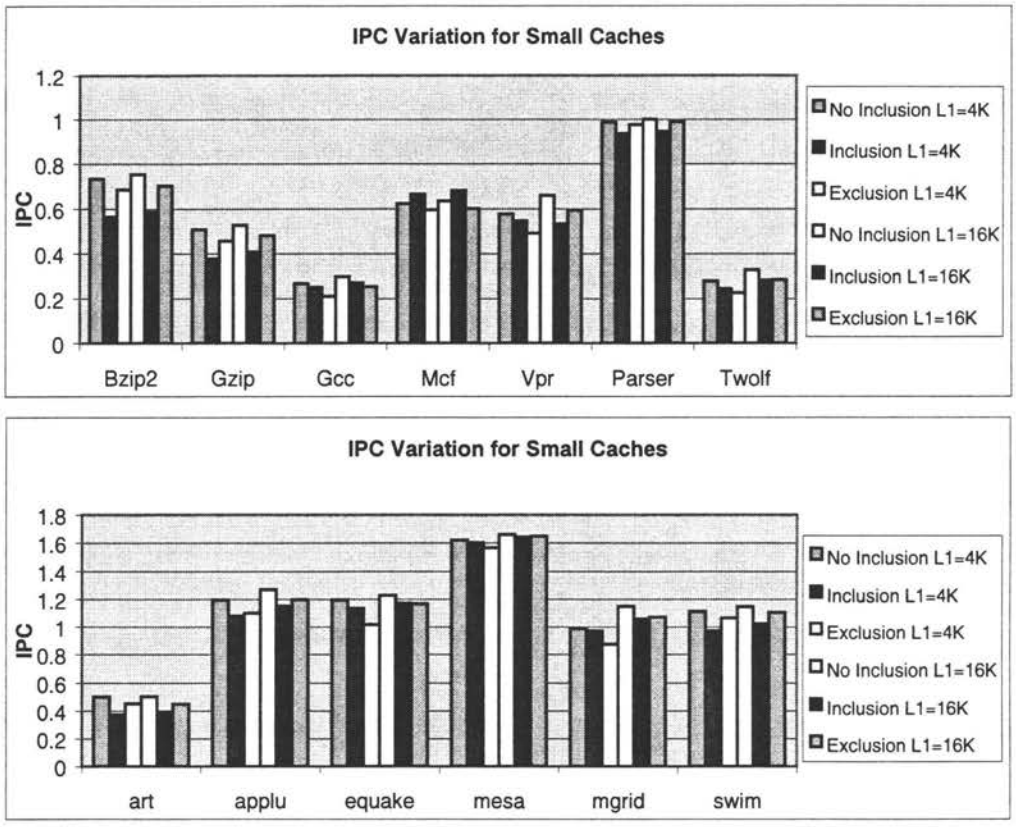

Figure 4-2: Variation of IPC for inclusion, exclusion and no-inclusion schemes for small direct mapped L1 caches with L2 size=64K direct mapped

performance of the exclusion scheme. In the exclusion scheme unlike in no-inclusion, all the blocks replaced from L1 have to be written back into L2. For larger L1 caches (16K), the no-inclusion scheme outperforms inclusion by up to 20% as in the case of bzip2 and art while in

case of benchmarks like twolf and Vpr, the difference is about 5%. Only in case of mcf, the inclusion scheme outperforms no-inclusion by 10%.

The reason for no-inclusion scheme performing better is that L1 and L2 put together cover a greater portion of the address space as compared to the inclusion case. Another reason could be that a block not in L2 cache would be written back even if it is not dirty

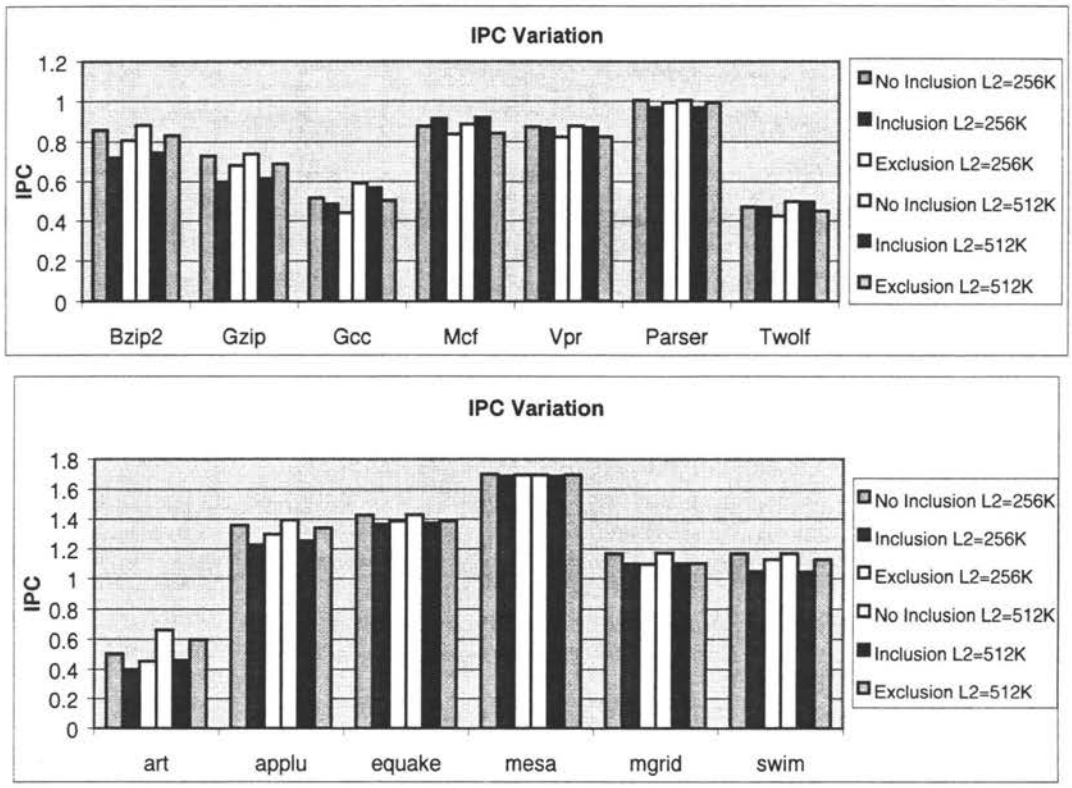

Figure 4-3: Variation of IPC for inclusion and no-inclusion schemes for large direct mapped L1 (64K) cache

and hence if accessed immediately would most likely be in the write buffer. Thus no inclusion may increase the effective set associativity [3]. Also in our simulations we use non-blocking caches. The value of a non-blocking cache depends on the effectiveness of the prefetch. The caches could be filled with anticipated data and less available for current requirements. This cache pollution problem is more profound in inclusion than in no

inclusion as data is brought into both cache levels on a miss. For smaller L1 caches (4K), the increased write back rates between L1 and L2 affects the performance of no inclusion as in case of Vpr and mgrid. Figures 4-4 and 4-5 show the global and local miss rates for inclusion and no inclusion. We see that the L1 cache miss rates are almost similar while the global miss rate in case of no-inclusion is lesser then in case of inclusion and exclusion. In case of mcf, the L1 as well as global miss rates are higher in case of no-inclusion and hence the lower performance.

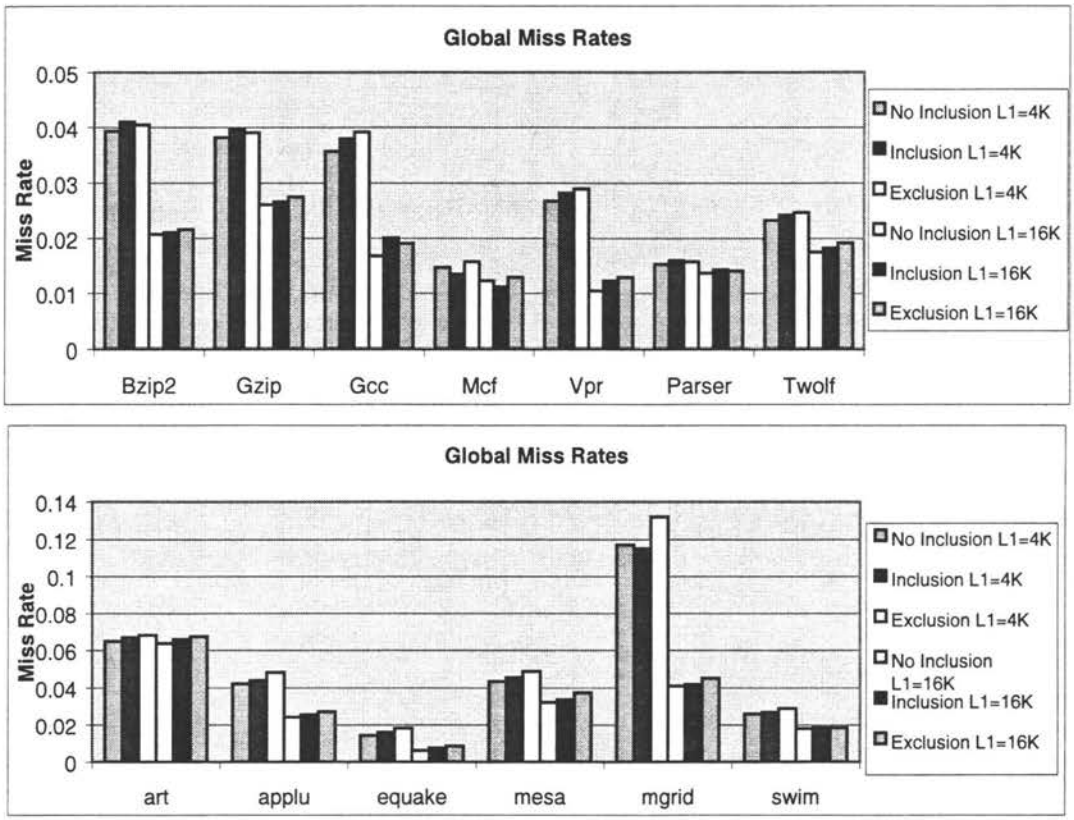

Figure 4-4: Variation of L1 miss rates for inclusion no inclusion and exclusion schemes with L2 size=64K direct mapped

Figure 4-3 shows the results of the simulation for large virtually indexed L1 caches. The performance of the no-inclusion scheme is better for almost all the benchmarks other than

mcf for large caches. The performance difference is as high as 15% in case of bzip2. The difference in performance in case of large caches is less because the property of the no-inclusion scheme to cover a greater portion of the address space is less effective when the cache size is big. The negative effect of increased write back rates in case of no-inclusion

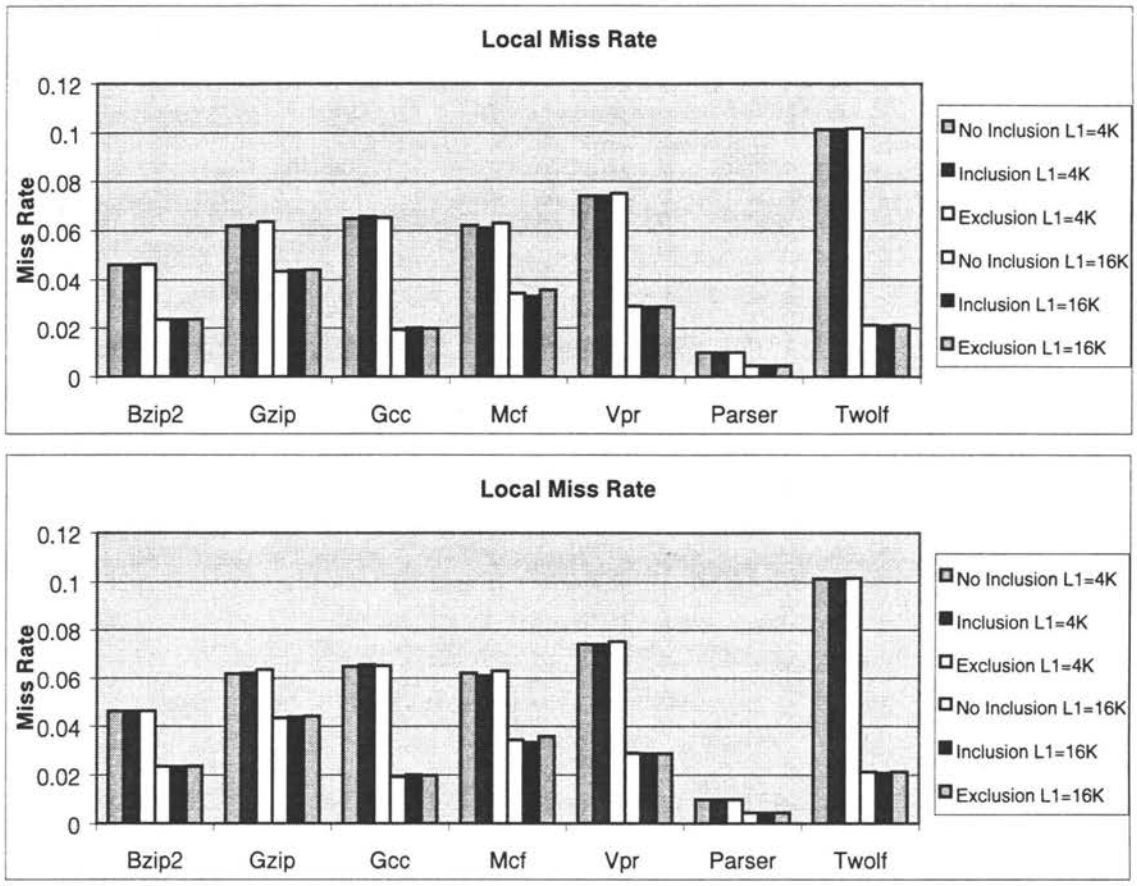

Figure 4-5: Variation of L1 miss rates for inclusion, exclusion and no-inclusion schemes with L2 size=64K direct mapped

is lesser, as with larger L1 size the write back rate from L1 decreases and hence the no-inclusion scheme outperforms inclusion scheme for almost all benchmarks except mcf. Mcf is highly memory intensive and hence the higher write back rates badly affect the

performance of the no-inclusion. Again exclusion performs well only for compute intensive benchmarks.

4.3 Pseudo Associative caches

One approach to achieve the hit speeds of the direct mapped cache is pseudo associative cache. Pseudo associative caches also try to achieve the miss rates of set associative caches. In this section, we see how with inclusion and with no-inclusion scheme react to pseudo associative caches. The level 1 data cache was organized as a pseudo associative cache. In a pseudo associative cache, when a block is replaced it is put in the pseudo set (pseudo set of a block is obtained from the cache set to which it maps, one simple scheme would be to invert the most significant bit of the set to get the pseudo set). The block is written back when it is replaced from the pseudo set. A cache access will proceed just as in the direct mapped cache for a hit. On a cache miss, the pseudo set is checked and if it is also a miss the next level is accessed. So there are two kinds of hit, one is the fast hit (regular hit) and other is the slow hit (hit in the pseudo set). The penalty associated with a pseudo hit will be 2 or 3 cycles greater than the L1 cache latency (2 in our simulations). The disadvantage could be that many of the hits might become slow hits. To avoid too many pseudo hits, on a pseudo hit a swap of the contents of the block is done. The block hit in the pseudo set is swapped with the block to which it would be normally mapped. The Simple scalar L1 data cache was modified to be a pseudo-associative cache and the benchmarks simulated. The results are shown in Figure 4-6. The No-inclusion scheme responds better than the inclusion scheme for both small and large cache sizes for most of the benchmarks including mcf with performance difference greater for larger cache sizes. In case of bzip2 the no-inclusion

outperforms inclusion by 20%. In case of parser there is not much difference between the schemes as the number of L1 misses and also conflict misses is less.

4.4 Effect of Prefetching

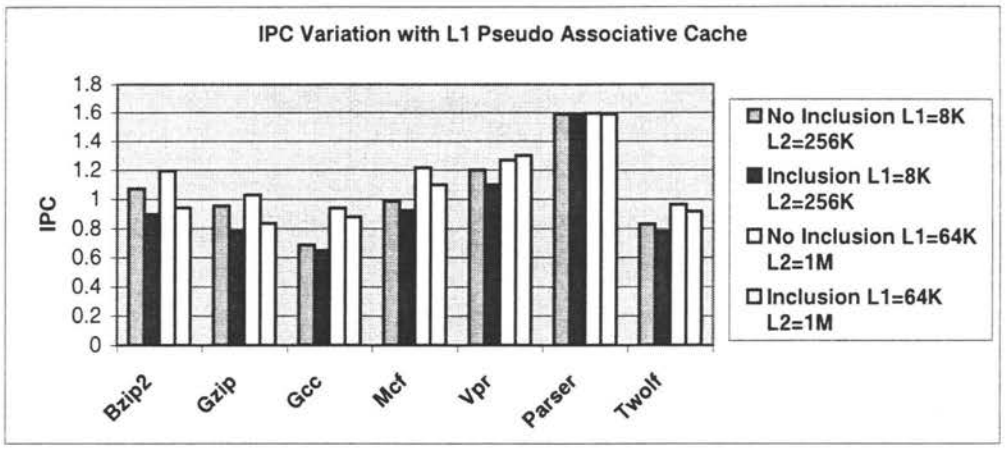

Figure 4-6: Variation of IPC for inclusion and no-inclusion schemes for pseudo AssociativeL1 caches

As L2 in mutual-exclusion can be considered as a victim buffer, the no-inclusion scheme can be considered as a large cache (L2) with a prefetch buffer (L1). So we consider the effects of prefetching. The prefetching scheme used is Next-Line prefetching [3]. Every time there is a L1 miss the next block is also fetched. This is based on the spatial locality and sequentiality principles of programs. Spatial locality says that given an access to a particular location in memory there is a high probability that other accesses will be made to either that or neighboring locations. The sequentiality principle says that if a reference has been made to a particular location 's', it is likely that within the next several references a reference to the location of 's+1' will be made.

The benchmarks were simulated for both the schemes with next-block prefetching scheme. The results indicate that the no-inclusion case responds better to prefetching than the

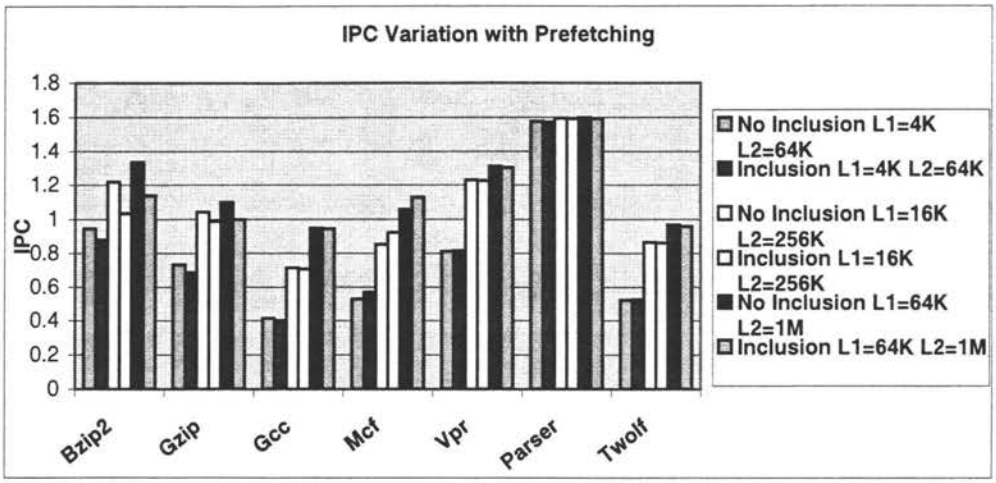

Figure 4-7: Variation of IPC for inclusion and no-inclusion schemes with next line prefetching

with inclusion case. Figure 4-7 gives the variation in IPC for the two schemes with next-line prefetching. For small caches the no-inclusion scheme outperforms inclusion by a maximum of 20% in the case of bzip2 and for most of the benchmarks other than mcf.

Prefetching could result in a frequently used block getting replaced. The block being prefetched might conflict with a frequently used block. The reason for no-inclusion responding better to prefetching could be because non-dirty blocks not in L2 are written back and would be a hit in the write buffer if accessed immediately. It would be a short L1 miss. In some sense the write buffers act as victim cache to the L1 cache. In case of inclusion, the block is written back only if it is dirty. If the access pattern is such that the two blocks mapping to the same cache block are accessed alternately then there night be a large number of cache misses. In case of no inclusion, the replaced block might be there in the write buffer.

So each of these would be a short miss rather than a long miss involving access to the L2 cache. In most programs a significant number of the misses are due to conflicts.

In general the no inclusion scheme would perform better than with inclusion scheme if the prefetching scheme results in the replacement of large number of frequently used blocks and their access in the immediate future. One more reason would be the amount of useful information in the caches. In case of inclusion, the prefetched block would have to be brought into L1 and L2, resulting in greater cache pollution in case the prefetched block is not used in the immediate future. The no-inclusion scheme also has this cache pollution problem but to a lesser extent. As the cache size increases the performance difference decreases. This could be because of lesser extent of cache pollution in case of large caches due to prefetching.

4.5 Performance Variation with Line size:

Till now we have considered systems where L1 and L2 block sizes are the same. The simplest form of no inclusion has this requirement. In this section, we consider a complex form of no inclusion were the L2 block size is greater than that of L1. The L1 and L2 block sizes were made different and no-inclusion and inclusion schemes simulated. When L2 size is made larger than the L1 size, the inclusion scheme starts outperforming the no inclusion scheme. In case of inclusion, when the main memory is accessed an L2 block sized chunk of data is got into the L2 cache. In the case of no inclusion, data is brought into L1 directly. If data is brought in chunks of L1's block size then the number of main memory accesses will be very high. So a prefetch buffer that can hold one/two L2 blocks is used. When the main memory is accessed a L2 block sized chunk is brought into the prefetch buffer. The prefetch

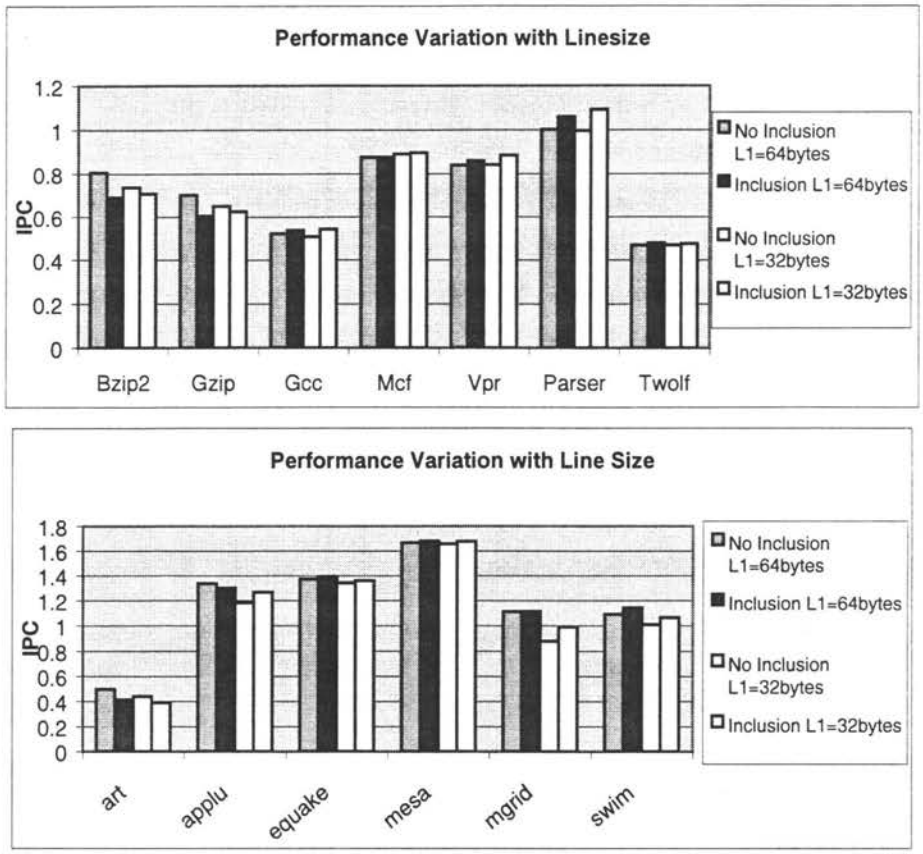

Figure 4-8: Variation of IPC with Line size for no-inclusion and no inclusion L1=32K L2=256K, 128 bytes line size direct map

buffer is accessed in parallel with the L1 cache. If the data is a hit in the prefetch buffer it is copied into L1. On replacement from the prefetch buffer the data is discarded. Also at the L2 cache, we need (L2 block-size/ L1 block-size) bits to indicate whether that portion of the data has been brought into the cache system in case of no-inclusion. The inclusion scheme does not have this overhead. Despite the presence of prefetch buffer, the number of main memory accesses is going to be higher for no inclusion compared to inclusion. The results are shown in Figure 4-8. The L2 block size is fixed at 128 bytes. The inclusion scheme outperforms the no-inclusion scheme by as high as 15% in case of mcf when the L1 size is 32 bytes. The

performance is better for inclusion for most of the benchmarks other than bzip2, art, applu and gzip. This is because in the case of these benchmarks the difference in the number of memory accesses between the schemes is lesser when compared to other benchmarks. In Exclusion scheme also data is brought in at the granularity of L1 block size and hence its performance also will be lower.

4.6 Variation with L1 associativity:

Till now in all our simulations we considered direct mapped caches. Direct mapped caches have very small access times. But the dis-advantage with direct mapped caches is that two blocks mapping to the same cache block would result in a miss every time if they were accessed alternately. A significant portion of the cache misses is conflict misses. The L1 cache was made set associative and the schemes simulated. The L1 cache was made 2-way and 4-way associative and results are shown in Figures 4-9 and 4-10. As the associativity increases the performance increases for both the schemes for most benchmarks. In the case of inclusion, the performance increase is more significant for few benchmarks like bzip2 and

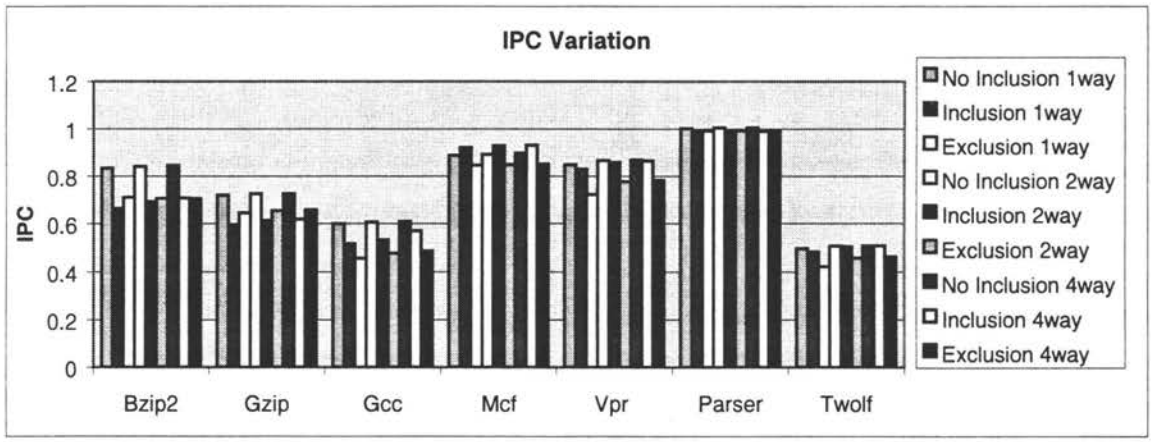

Figure 4-9: IPC Variation with L1 associativity L1=32K L2=256K 4way

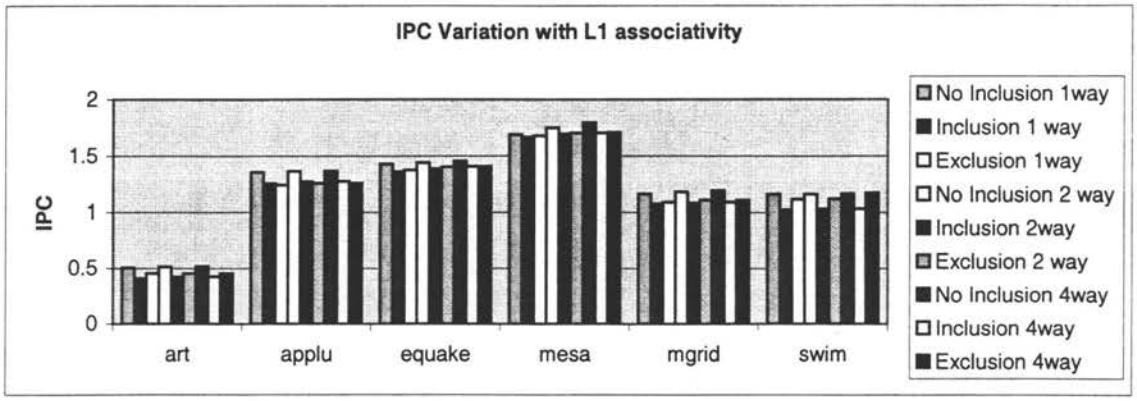

Figure 4-10: IPC Variation with L1 associativity L1=32K L2=256K 4way

gzip. This is because the global miss rate decreases by a more significant fraction in the case of with inclusion for these benchmarks. In case of bzip2, the performance increases by almost 10% when the L1 cache is made 2-way associative rather than direct mapped. For most of the other benchmarks the performance increases is comparable for both the schemes. But the overall performance is still better in the case of no-inclusion.

The presence of write buffer is a greater advantage to the no inclusion scheme as the write buffer acts as a victim cache as well. In the case of inclusion, it helps in write combining alone. In case of no-inclusion, the non-dirty blocks not in L2 and hence written back enter the write buffer. So if these blocks are accessed immediately, they will be a hit in the write buffer. The performance doesn't vary much beyond a 4-entry write buffer.

4.7 Blocking Caches

Non-blocking caches are the fastest approach for handling cache misses. But this requires extra control hardware. In this section we consider a slow approach in which the processor waits for the missed line to be brought into the cache before proceeding with the

execution. If the processor has to wait till the entire cache line is brought in it is called fully blocked. If fetch bypass is used then processing begins once the first word is accessed. These are called partially blocked caches. In this section, partially blocked caches are considered. The inclusion and no-inclusion schemes were simulated with a blocking L1 cache and the results are shown in Figure 4-11. The no-inclusion scheme performs better than inclusion for all the benchmarks for small caches. Even in case of mcf, the performance difference between the schemes is very minimal. For large caches however the no inclusion scheme outperforms inclusion except in case of *mcf* alone. The performance difference is as high as 30% in case of *bzip2* for small caches.

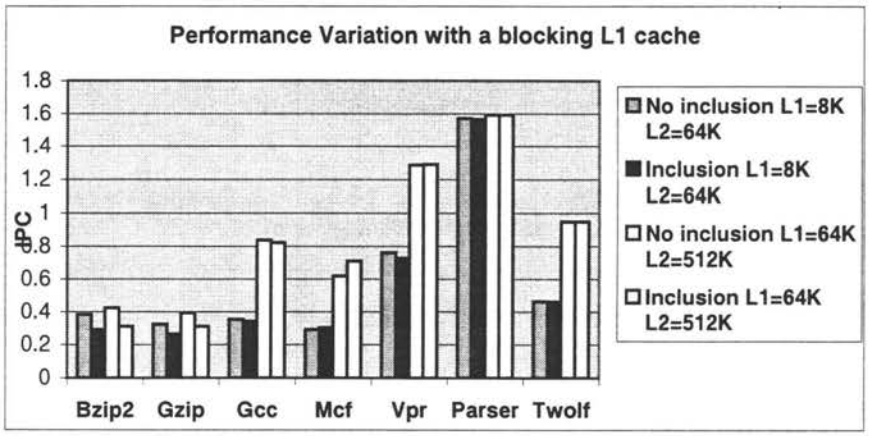

Figure 4-11: Variation of IPC for inclusion and no-inclusion schemes with L1 blocking cache.

In this chapter, the performance potential of three approaches in 2-level data cache environments was discussed and no inclusion is found to have the most potential with Spec2000 benchmarks running on Simple Scalar simulator, the no inclusion outperformed inclusion under most of the situations. In the next chapter, two optimizations to improve the performance of no inclusion are discussed.

5 IMPROVING NO INCLUSION

Although the no-inclusion is found to have better performance potential than the inclusion, it can be made to be even better. With the no-inclusion, the write-back rates from L1 to L2 are high compared to with the inclusion, especially when L1 size is small. This becomes a performance bottleneck especially for memory intensive programs. The write back traffic contends for bus access with data that has to be fetched into the cache for program execution. If the information as to whether a particular block will be used again in the immediate future is known, we need not write back data that will not be referenced again in the near future. We consider improving the no-inclusion with a scheme to determine whether to write back replaced data (not in L2 and not dirty) from L1 into L2. We selectively write back data from L1 to L2 and by this we try to reduce the data traffic between L1 and L2. This reduces congestion in the backside bus and also increases the backup space for useful data replaced from L1 cache in L2. As an initial scheme we consider the following three well-known aspects of data access behavior in programs:

- Memory access pattern for stack data generally have very good spatial and temporal locality since the data occupies a small, contiguous area of the memory near the top of the stack. The stack is heavily used in modern programs compiled from high level languages for parameter passing in function calls, allocating local variables used in the function body and for register spilling. For data in the stack region the lifetime will be generally less than the duration time of that block's L1 tour. In [16], it has

been shown that despite the large number of stack accesses, their reference space tends to be very small.

- Cache misses are classified into conflict and capacity misses. Conflict misses happen if two different memory blocks map to the same cache block while capacity misses happen if the cache cannot contain all the blocks needed during the execution of program. A block replaced from L1 because of a capacity miss will be less likely accessed again. In [1], it has been shown that the performance of the architecture can be improved by having a bias against capacity misses during replacement.
- A very small number of load instructions are responsible for a large number of misses in program executions. In [7], it has been shown that fewer than 10 instructions account for half the misses for most of the benchmarks. So the blocks got into the cache by these frequently faulting instructions will have less locality.

Initially the advantage from not writing back those stack blocks that are not expected to be used again in the immediate future is explored. The data blocks are classified based on their access region into stack and non-stack as in [5]. Each L1 cache block has a bit associated with it to indicate whether it belongs to the stack or non-stack region and is set when the block is brought into the L1 cache. Also each cache miss is classified as either capacity-miss or conflict-miss. Another bit in each cache block classifies it as being brought in due to a capacity or conflict miss. A simple mechanism proposed in [1] is utilized to classify each cache miss as a conflict miss or a capacity miss. The scheme uses a hardware table that maintains an entry for each cache set. The tag of the last block replaced from that set is stored in the entry corresponding to the set. If the next miss to the set has the same tag then it is a conflict miss else it is a capacity miss. The basis behind this classification is that if

the cache had been slightly more associative the line may have been a hit in the cache. The entire tag need not stored but more the tag bits the greater will be the accuracy of the classification. We can also store the tags of the last n blocks replaced from each set. Every cache block has a bit associated to indicate the cache miss type and is set if that block is brought in due to a conflict miss. In [1], it has been claimed that this scheme classifies 85% of the cache misses correctly and full accuracy can be got by only keeping 8 bits of the tag.

On replacement, in addition to dirty stack blocks, a non-dirty stack block that is not in L2 (recall the presentin-L2 bit associated with each L1 block) but is marked as a block brought in on a conflict miss is written back to the L2. Other non-dirty blocks for stack data are written into a victim cache [8] and on replacement form the victim cache are simply discarded. This is because non-dirty stack blocks brought in due to capacity miss are less likely to be accessed again. Considering only this aspect did not yield significant improvement in the performance, as the number of non-dirty stack blocks written back from L1 to L2 is low for some benchmarks. For instance, with L1=8K and L2=128K, we found that for the benchmark *gzip* only 2% of the blocks written back were non-dirty stack blocks though a large fraction of the accesses are to the stack region. So the performance improvement from selectively writing back only non-dirty stack blocks is very low for some benchmarks. Furthermore even amongst non-stack blocks, a large number of non-stack blocks written back to L2 might not be accessed again. Abraham et al. have showed that a few percentages of loads are responsible for a large number of misses in [7]. For non-stack data, we try to capture a small set of instructions that causes majority of L1 cache misses. The idea is that the cache blocks brought in by these frequently faulting instructions have less locality and hence are less likely to be accessed again in the immediate future.

We maintain a "load address table" (LAT) containing the program counter (PC) values of some load instructions and a 3-bit saturation counter for each entry. With every L1 miss, the counter value of the LAT entry corresponding to the current load instruction that caused that miss is incremented. Each entry has a sticky bit associated with it. Every access to that entry refreshes the sticky bit to '1' value. On replacement if the counter is saturated we reset the sticky bit and do not replace the entry in the LAT. We also do not update LAT with the new load entry. If, before the next miss, the entry is not accessed, then it is replaced in the LAT. This is to prevent a rarely faulting instruction from replacing a frequently faulting instruction. Also when an entry is made for an instruction in the LAT the counter value is reset. With the LAT, a block brought into L1 by an instruction whose counter is saturated is expected not to be used again and hence is not written back on replacement. A bit indicating whether to discard or not is associated with every L1 block and is set or reset when the block is brought into the cache based on whether the PC of the instruction accessing the block is a hit in the LAT. Again as in case of stack blocks, non-stack blocks discarded are written into the victim cache and thrown on replacement from the victim cache. A 3-bit saturation counter was used because experimental results showed that a 2-bit counter caused too many L2 misses thereby affecting the performance, while beyond 3 bits the performance did not vary much. This is because the set of instructions captured by three or more bit counters remained the same.

The above-described technique of identifying useful blocks to be written back might involve a lot of mis-predictions. We might discard blocks that would be accessed immediately. To reduce the effect of mis-predictions, a simple scheme is proposed that switches the architecture into a "no-discard" mode when the L1 miss rate increases. The

increase in miss rate might have been caused by the access pattern of the program rather than due to the discarding of L1 blocks on replacement, but by switching to the "no-discard" mode we prevent further increase in the miss rates. The miss rate is calculated over a fixed period say 10,000 cycles. An n-bit miss register is used. If the miss rate increases between successive periods, a '1' is shifted into the register or else a '0' is shifted. The architecture shifts to "no-discard" mode if the miss rate has been increasing for the last n-periods. The architecture gets back into the "discard" mode once the miss rate decreases. The performance was best for a 4-bit miss register.

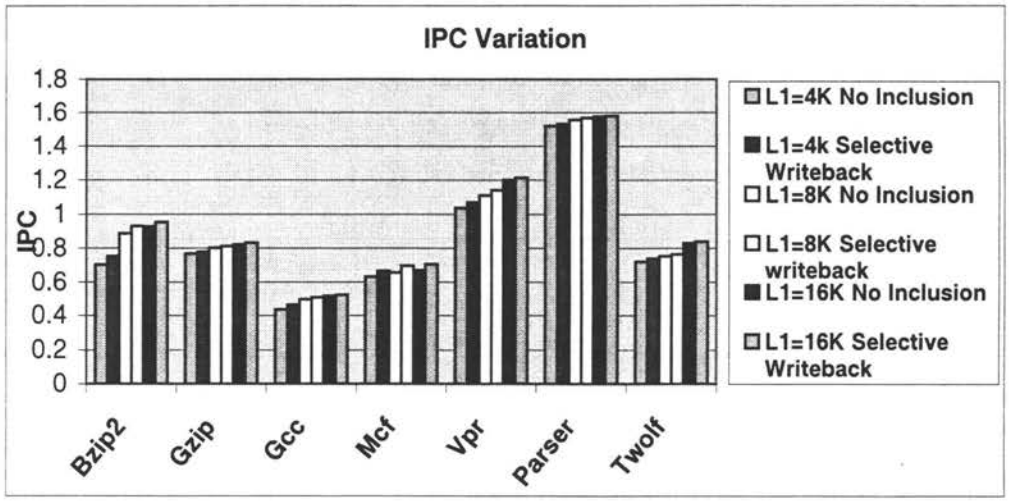

Figure 5-1: Variation of IPC for no-inclusion and no inclusion with selective write back

The scheme was implemented and the benchmarks simulated. The IPC results are shown in Figures 5-1 and 5-2 and the percentage of decrease in the memory accesses in Figures 5-3 and 5-4. The performance increase is as high as 9% in case of bzip2 and mcf. Only in the case of parser the performance improvement is less than 1%. On average the performance of most benchmarks increases by up to 5%. The number of main memory accesses decreases up

to 23% as in the case of Vpr and 17% for applu with selective write back. The reason why this difference does not translate directly into performance is because of the increased traffic on the internal bus.

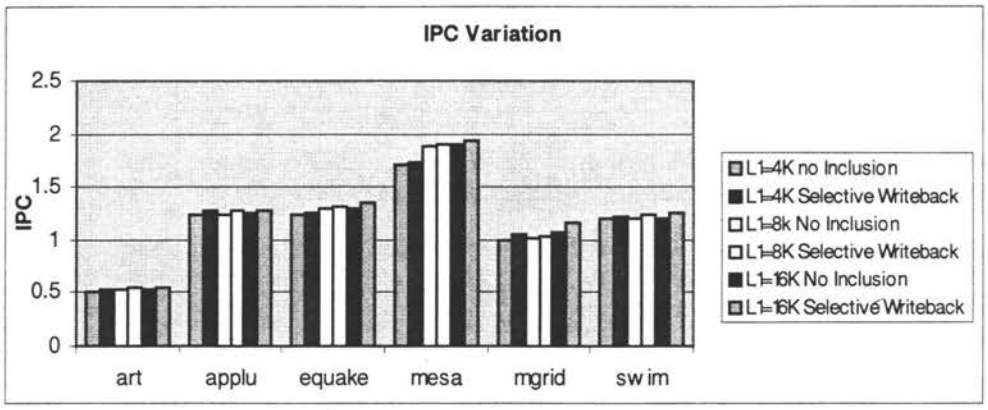

Figure 5-2: Variation of IPC for no-inclusion and no inclusion with selective write back

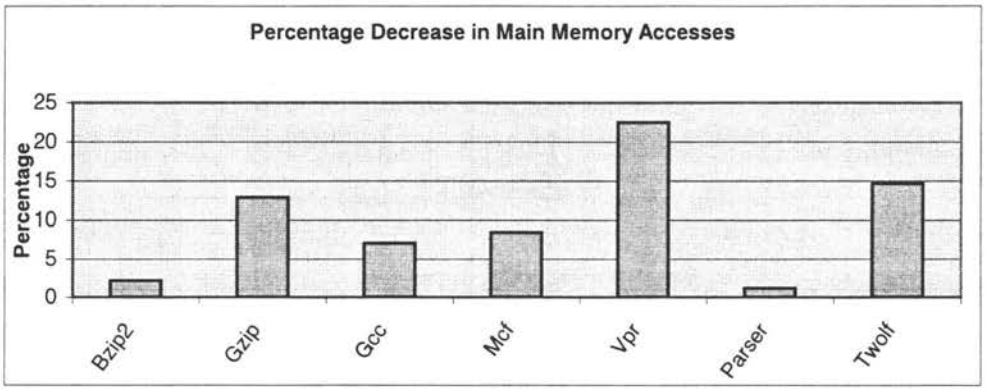

Figure 5-3: Percentage decrease in memory accesses for no-inclusion scheme with selective write back.

5.1 Hardware and Timing Complexity of the scheme

Each L1 cache block requires two more bits, one for indicating if it is a stack block and other for type of miss that caused the block to be brought into the cache. The classification of the cache blocks into stack and non-stack requires no hardware. If the block address is

between the stack base register and stack pointer, the block is classified as a stack cache block. Classification of the miss type requires that each cache set store the tag of the last replaced block. This information is determined when the block is fetched on a cache miss. In [1], it has been claimed that full accuracy can be got by only keeping 8 bits of the tag. Thus each cache set requires 10 extra bits. The Load Address Table is updated on a cache miss and the information as to whether to discard the block being fetched into the L1 cache currently can be discarded on future replacement is determined during the L1 cache fill. The miss register unit also operates in parallel with the cache accesses. The LAT stores the PC value of the load instruction and a 3-bit value. A 16-entry LAT takes 16 *(32+3) bits assuming a 32 bit PC value. The miss register unit requires an n-bit register, a register for storing the miss rate during the lasts period, a couple of registers for calculating the miss rate during the current period and a counter for tracking the periods. None of these come in the critical path of execution.

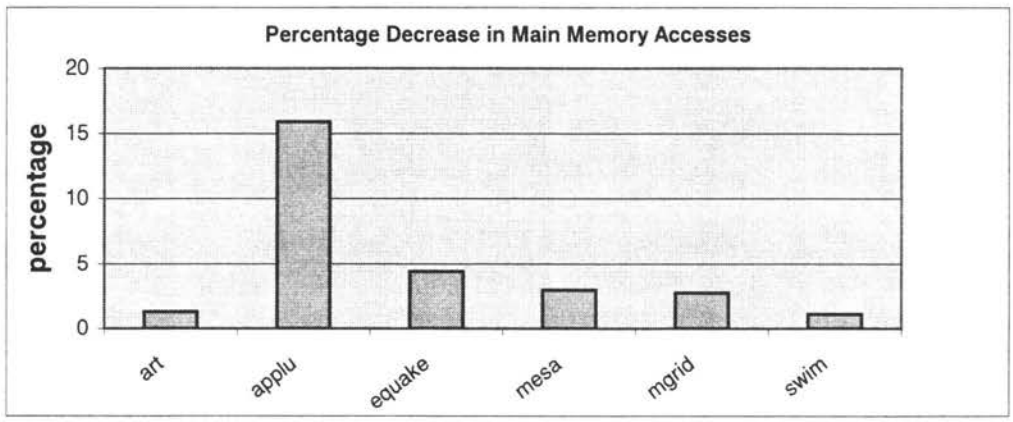

Figure 5-4: Percentage decrease in memory accesses for no-inclusion scheme with selective write back.

6 CONCLUSIONS

Inclusion property has been traditionally enforced in multi-level data caches as it is assumed to provide a couple of benefits. First, the inclusion allows data access to L1 to proceed simultaneously with probing of cache contents from coherency traffic. Second, it is considered to require less complex control. Further, since L2 cache has been usually off chip and many folds bigger than L1 cache, the overhead due to duplication of L1 at L2 does not pose a serious capacity issue. However, these advantages seem a claim not well based on facts. Even without the inclusion, a separate copy of L1 tags at L2 interface can easily provide less interference of coherency traffic to L1 access. As seen in Alpha 21264 and AMD Athlon, separate tag storage for additional copy of L1 and L2 tag requires not only little extra storage but also little extra control complexity. Also, inclusion does not allow less complex control as intuition may suggest. Further, now L2 caches are increasingly being on-chip along with L1. With increasing processor clock cycle and ever increasing number of transistors available on-chip, a relatively modest sized L2 cache appears to be a good resource to be on a chip along with processor core and L1 cache. When both L1 and L2 caches are on-chip resources, transfer rate between them can be very high and also the modest size of L2 makes duplication of L1 data a significant overhead. Many data missing in L1 will also miss in L2. So in this case L2 may add more to the delay between L1 and off-chip access than reducing the number of off-chip accesses. Not enforcing the inclusion

property may be able to offer significant reduction in off-chip accesses as it naturally provides more back-up capacity in L2 cache.

This thesis studied performance potential of the three approaches in two-level data cache environments, and no-inclusion approach is found to have the most potential. In case of the mutual-exclusion all blocks replaced from L1 cache have to be written back to L2 cache and so the traffic between L1 and L2 caches is very high thereby reducing the performance advantage obtained by greater capacity. With Spec2000 benchmarks running on the Simple scalar simulator modified to behave like Alpha 21264; the no-inclusion outperformed the inclusion under most of the situations simulated, although, for small L1 sizes (4KB and 2KB), the with inclusion scheme performed better for some benchmarks. No inclusion combines the advantages of mutual exclusion (better usage of on-chip area) and inclusion (less write back traffic between L1 and L2). Also, the no-inclusion is found to be a better choice when L1 is a pseudo associative cache.

But the no-inclusion approach suffers from negative effects of traffic between L1 and L2 caches, especially if the L1 cache size is small, which limit its potential. In this thesis, a scheme to reduce the traffic between L1 and L2 has been proposed. The scheme reduces congestion on the back bus by selectively writing back data from L1 to L2. This also increases the backup space in L2 for useful data replaced from L1. In the proposed scheme there might be quite a few mis-predictions. Mis-predictions might lead to immediately used data being discarded from L1. Future work would involve developing other techniques for presenting a clear view about the re-usability of data to the architecture so that mis-predictions would not cause unnecessary main memory accesses.

BIBLIOGRAPHY

[1]. Jamison D.Collins, Dean M. Tullsen, "Hardware identification of cache conflict misses", In proc. 32nd International Symposium of Microarchitecture, November 1999.

[2]. J.-L.Baer and W.-H. Wang, "On the inclusion properties of multi-level cache hierarchies", In proc 15th Annual International Symposium on computer Architecture, pages 73-80, May 30-June 2, 1988.

[3]. N. P. Jouppi and S. J. Wilton, "Tradeoffs in two-level on-chip caching", In Proceedings of the 21th Annual International Symposium on Computer Architecture, pages 34-45, April. 1994.

[4]. R.E,Kessler, "The Alpha 21264 microprocessor", IEEE Micro, 19(2): 24-36, March/April 1999.

[5]. Michael J.Flynn, "Computer Architecture: Pipelined and Parallel Processor Design", Boston: Jones and Barlett, 1995.

[6]. M.Cekeleov, M.Dubois, "Virtually Addressed Caches Part-1: Problems and Solutions in Uniprocessors", IEEE Micro Sep-Oct 1997.

[7]. Santosh G. Abraham, Rabin A.S, Daniel Windheiser, B.R. Rau and Rajiv Gupta, "Predictability of Load/Store Instruction Latencies", Proceedings of the 26th Annual International Symposium on Microarchitecture, pages 139-152, December 1993.

[8]. N.P Jouppi, "Improving direct mapped cache performance by the addition of a small fully associative cache and prefetch buffers", Proceedings of 17th Annual symposium on computer Architecture, pages 364-373, 1990.

[9]. United States Patent 5,386,547 Jouppi, January 31,1995, "System and Method for exclusive two level caching".

[10]. United States Patent 5,564,035, Lai, October 8, 1996 "Exclusive and/or partially inclusive extension cache system and method to minimize swapping there in".

[11].D.C.Burger and T.M.Austin, "The Simplescalar tool set", version 2.0 Computer Architecture News, 25(3): 13-25, June 1997.

[12].Sangyeun Cho, Pen-Chung Yew and Gyungho Lee, "Access Region Locality for High bandwidth Processor Memory System Design", 32nd Annual International Symposium on Microarchitecture, Nov 16-18, 1999.

[13].Joshua J. Yi, Resit Sendag and David J.Lilja, "The Spatial Characteristics of Load Instructions", Laboratary for Advanced Research in Computing Technology and Compilers, Technical Report ARCTiC 02-10, October 2002.

[14].SPEC, "SPEC2000 CPU Benchmark", http://www.spec.org, 2000.

[15]. J.L.Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach", New York: Morgan-Kaufmann, 3rd ed., 2002.

[16] S. Cho, P. Yew, and Gyungho Lee, "Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor", Proc. of the 26th International Symposium on Computer Architecture, Atlanta, GA., May 1999.

[17] W. H. Wang , J.-L. Baer , H. M. Levy, " Organization and performance of a two-level virtual-real cache hierarchy", ACM SIGARCH Computer Architecture News, v.17 n.3, pages 140-148, June 1989.