A SPLIT DATA CACHE ORGANIZATION BASED ON RUN-TIME DATA LOCALITY ESTIMATION

A Dissertation

Submitted to the Department of Computer Science and Computer Engineering

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Engineering

By

Quazi Galib Samdani

MS in Applied Physics & Electronics,

University of Dhaka, 1990

MS in Electrical Engineering

University of Arkansas, 1996

April 2000

University of Arkansas

This Dissertation is approved for
recommendation to the
Graduate Council

Dissertation Director:

_____
Dr. Mitchell A. Thornton

Dissertation Co-Director:

_____
Dr. David L. Andrews

Dissertation Committee:

_____
Dr. Neil M. Schmitt

_____
Dr. J. Sherwood Charlton

_____
Dr. Terry W. Martin

Abstract

Cache memories are used extensively in modern computer organizations in order to reduce the performance gap between fast microprocessors and slower main memory. Cache memory hides the main memory access latency by exploiting the data locality present in pre-fetched memory blocks in the cache. Conventional pre-fetching policies used in traditional cache organizations have the potential to waste the available cache bandwidth and space by bringing non-usable data in the cache. Conventional caches cannot meet the different sized storage requirements of data that exhibit spatial or temporal locality characteristics when their address spaces vary and are non-overlapping. Data characterized by spatial or temporal locality could be more efficiently accommodated if caches with different line sizes based on the locality type could be used. The fixed line size of a conventional cache restricts this efficiency. To reduce the performance bottleneck of conventional caches, an alternative cache organization is explored in this research. The SPEC92 benchmarks as well as other standard benchmark programs are used to observe run-time data locality. Based on the locality analysis, a simple locality prediction technique was designed in hardware capable of estimating the data locality bias of the cache-resident data during run-time. This prediction hardware is used to design a split data cache that uses two sub-caches; spatial and temporal cache. This organization stores data in the respective sub-caches based on the dynamic locality estimation during run-time of the executing programs. The split data cache organization showed a considerable performance increase over a conventional unified data cache by

reducing the overall cache miss rate and bus data traffic. A better utilization of the cache space and bandwidth is possible using this new organization.

Key words and Phrases: Data Locality Cache, Computer Architecture, Memory system organization, High Speed Memory, Run-time Memory Access pattern.

Acknowledgements

I would like to express my sincerest thanks and gratitude to Dr. Mitchell A. Thornton for letting me pursuing this research, exchanging interesting ideas, solving problems and finally writing this dissertation. I would also like to thank Dr. David L. Andrews for his unique thoughts, support and accommodation for research in this department. Finally, I would like to thank first Dr. J. Sherwood Charlton for his silent support and encouragement and Dr. Neil M. Schmitt and Dr. Terry W. Martin for serving as valuable committee members.

Dedication

To Bimurto, Borshan, Juthee

and

In memory of Golam Samdani (father) and Carl Sagan (NASA)

Contents

List of Tables

List of Figures

CHAPTER 1

7 Introduction

One of the main bottlenecks of current computer architectures is the processor-memory interface. This bottleneck results in a mismatch between the speed of the CPU and memory, and is referred to as the processor-memory performance gap [1, 2, 3, 4]. Microprocessor performance is increasing at a faster rate than memory. Typical usage of off-chip memory units in computer architectures causes increases in access latencies and bandwidth limitations due to the processor-memory interface path and the finite number of pins that chip packaging allows [2,3]. Off-chip memory access times are higher than on-chip memory access times due to the relatively larger latency introduced during data propagation from the memory chip to the microprocessor through an external data path. Bandwidth is dependent on the data transfer rate between microprocessor and memory. Separate chip packaging limits the number of data I/O pins available. Thus, the data transfer rate is highly dependent on the number of pins available.

A common scenario is that programmers using state-of-the-art computers are increasingly demanding faster memory units in their computers to fully utilize the performance increase of the microprocessors. As faster memory units (specifically SRAM - static random access memory) are more expensive than slower dynamic RAM, designers must

mitigate this performance gap. Therefore, modern processors use a memory hierarchy composed of a combination of faster SRAM and slower DRAM. Figure 1.1 shows a typical memory hierarchy used by computers.

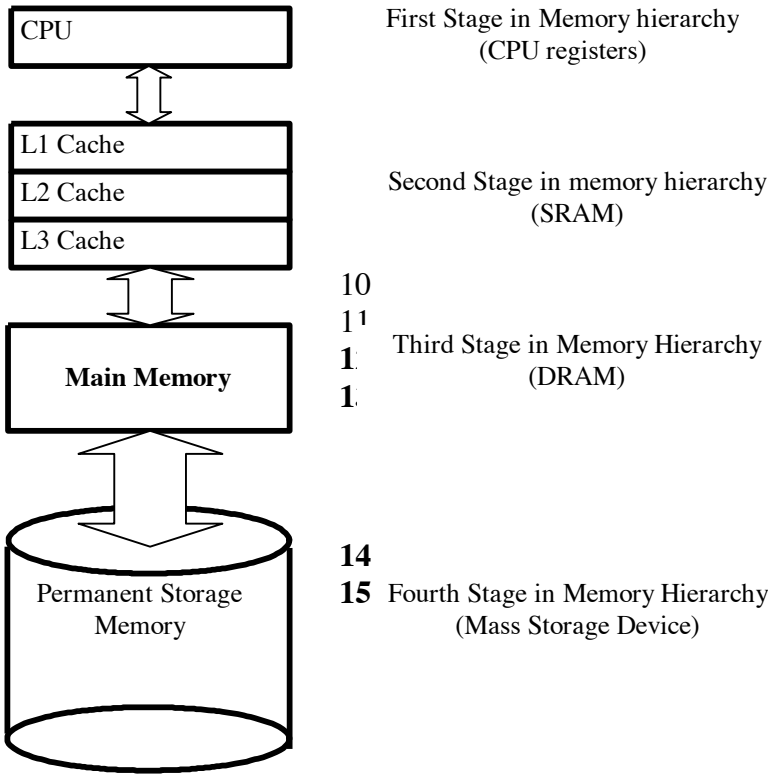


Figure 1.1. Typical memory

In this organization, the faster SRAM units serve as the second stage of memory hierarchy, and are organized as a cache memory. With the inclusion of a memory management unit (MMU), required data and instructions are fetched from the slower main memory (which is the third stage of the memory hierarchy) into the faster cache unit. The CPU can take advantage of the faster cache access times and the higher bandwidth available for using the cache organization. The performance benefit obtained

by using faster cache units depends on both caching policies that exploit data and instruction locality, and, on the physical organization of the cache.

The introduction of cache memories [5,6] allowed for significant performance increases in the early 1980's when the performance gap between the CPU and memory was not as large. CPU performance continues to increase at a tremendous rate every year, and cache organizations being used in an attempt to keep up with the faster data access demands. In the past 10 years, different cache organizations have been proposed to rectify this issue. In the 1980's, multi-level cache architectures were introduced. A multi level cache takes advantage of the extraordinary integration density offered by the current chip fabrication and packaging technologies, and may be integrated within the same die as the CPU. As an example, Intel's Pentium Pro® integrates the CPU, I-Cache, D-Cache and L2 cache onto the same die. Though multi-level caches help to reduce the memory access latency, they also introduce additional latency in cases where the memory lookup function has to traverse deeper into the memory hierarchy for cache misses. In some cases, the CPU can waste about 75% of its processing time due to the look-up penalty in the multi-level cache [45]. We are currently living in an 'information age' where almost all information is being kept in local and distributed databases, and, information is consistently being shared over the Internet and Intranet. Accessing huge databases is a very critical and time-consuming process. Database programs typically waste more than 50% their operation time in retrieving information in the memory hierarchy [8,9,12,30,31,32,39]. Proper caching of the requested data is fundamental for these applications.

To obtain a balanced CPU/Cache system we not only need the best architecture, but also an optimized caching strategy that will perform well in all general cases. The success of conventional caching policies depends on the locality present in the accessed data or instructions. The current policies used in caches are not always highly successful in caching data properly in order to gain maximum benefit from the varying spatial and temporal localities exhibited by the data. The cache resource can be polluted, in some cases up to 60%, due to the residence of unused data in the cache occurring to pre-fetching of non-usable data in the cache [57].

In the following subsections, basic cache organizations and concepts, such as locality, will be introduced. Then, the motivation for performing the current investigation, current research results performed by other researchers, and the split data cache subsystem design are presented. Finally, an overview of the contributions made by this research is presented.

1.1 Cache Basics:

Cache memories work based on the locality of the code segments of the program and accessed data. The types of locality are defined as follows [39]:

a. Spatial Locality (or locality in space): Given an access to a particular location in memory, there is a high probability that other accesses will be made in the neighboring locations within the lifetime of the program.

~~c.~~b. Temporal Locality (or locality in time): Given an access to a particular location, there is a high probability that references following that access will be made to the same location. If a program exhibits temporal locality, elements of the reference sequence will be accessed again during the lifetime of the program.

~~d.~~c. Sequentiality: Given that a reference has been made to a particular location *s*, it is likely that subsequent references will access the location of *s+1*. Sequentiality is a restricted type of spatial locality and can be regarded as a subset of it.

When a reference made by a processor is found in the cache, it is called a ***cache hit***. When the reference is not available in the cache, it is called a ***cache miss***. In the case of a cache miss, the cache control mechanism must fetch the missing data from the main memory and place it into the cache.

1.2 Cache Organization:

Basic cache organizations follow two fetching schemes. A "demand fetch" organization where memory contents are fetched based solely on cache misses. The second organization called "pre-fetching," fetches data depending on a priori anticipation of locality of references. This is also referred to as "speculation."

There are three basic types of cache organizations based on the 'main memory address mapping scheme' in the cache. They are a) Fully Associative, b) Direct Mapped, and c) Set Associative. Figure 1.2.

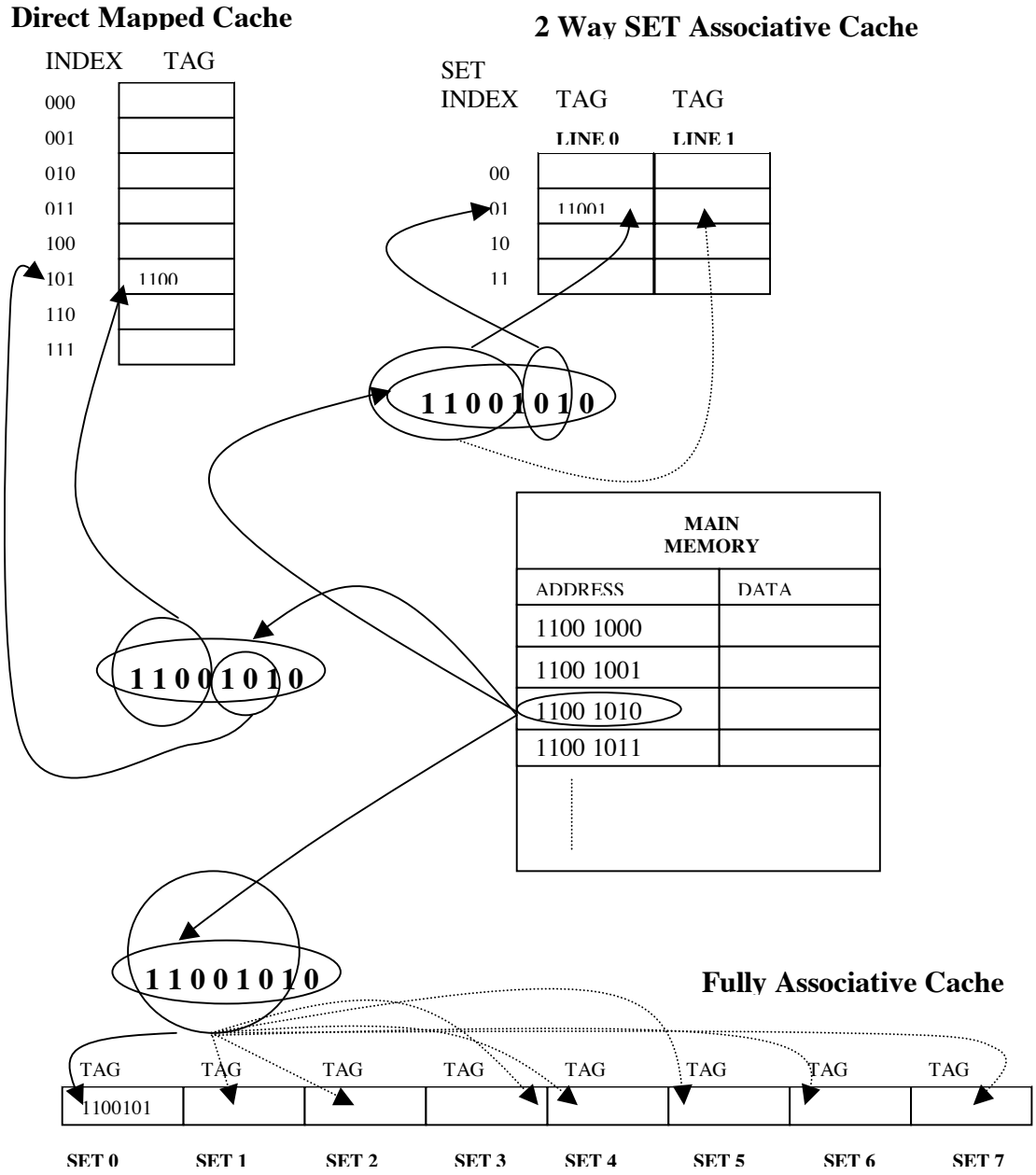


Figure 1.2 Illustration of an 8-line cache address-mapping process for Direct mapped, Set Associative and Fully Associative organizations (Cache Data Block size is two bytes here).

~~b.~~a. Direct mapped: Memory reference blocks are mapped to a specific block location in the cache. As an example, the address 0xCA in the illustration can be mapped only in the line with index 101.

~~c.~~b. Set associative: Memory references are placed into a restricted subset of blocks in the cache. A particular memory reference block can only be placed in a specific set, but the block can be placed anywhere within that set. In Figure 1.2, the address 0xCA can be mapped in either 'line 0' or 'line 1' with set index '01' in the 2-way set associative cache in the illustration.

c. Fully associative: Memory references block may be placed anywhere in the cache. The address 0xCA can be mapped anywhere between set or line 0 to 7 in the shown illustration in the Figure 1.2.

1.3.1 Cache read process:

During a read operation, the cache looks for a match between the address and the stored address in the cache. The cache stores the 'address tag' for each block in the cache in a tag RAM. An additional bit is also stored in the tag RAM for each block to indicate if the tag entry is a valid data-block in the cache. During a tag match, the cache control circuitry checks this bit, referred as the valid bit 'V'. If the valid bit is 'set' during a tag match, then control circuitry makes a data transfer from that cache block to the CPU register. In a direct mapped cache, only a single cache block entry is selected by the mapping process to be searched for a match. In associative caches, all the cache blocks, decided by the degree of associativity, are searched in parallel. Figure 1.3 shows how the

memory address field is partitioned to derive the address tag, cache index, and block offset fields for the cache. First, it is divided into a block address and block offset. The block address is further divided into tag and index fields. The block-offset field selects the desired data from the block; the index field selects the set, and the tag field is compared against the stored address in that set for a hit.

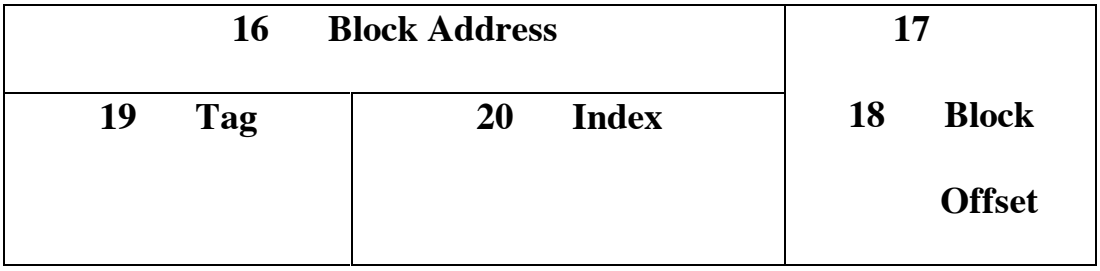


Figure 1.3 The three portions of an address in a set-associative or direct mapped cache.

1.3.2 Cache update process:

During a cache miss, the control circuitry must update the cache with the missed memory block. In performing this operation, a valid cache block may need evicted to accommodate the requested data by the CPU. In a direct mapped cache, new memory block is placed into a single location independent of whether the existing entry is valid or invalid. For a set associative or fully associative cache, there can be multiple cache blocks available for replacement. The cache controller must decide which block of the cache should be replaced. There are several cache replacement policies available to make this decision. These policies have their own merits and de-merits. Two common “cache replacement policies” [3] that are used are as follows:

a. Random: In this strategy a block for replacement is selected in random fashion, which is believed to provide uniform spread of allocation. This policy may provide poor performance in average cases.

~~c.~~b. Least Recently Used (LRU): This strategy allows for replacing blocks based on their aging or least recent usage. Some aging counters are used to track the recent usage of each block in this strategy.

The hardware cost and complexity is less for implementing the random replacement policy, however the cache performance suffers in the average case. Alternatively, the LRU policy yields good cache performance but the hardware cost and complexity increases linearly with increase of the associativity of the cache.

1.4 Cache Write Process:

There are two types of write policies generally used for cache writes; a) Write-through and b) Write-back. In a write-through policy, both cache blocks and lower order memory structures are updated with the data at the same time during a write operation. Whereas, in a write-back policy, only the block in the cache is updated. This makes the write-back process faster than the write-through process since it can be done at cache speed. However, additional difficulties can arise due to inconsistencies that can occur in the cache versus main memory content. This difficulty is referred to as the "cache coherency" problem.

According to [38], cache reads occur more frequently than cache writes. Usually about 25% of the cache bandwidth is utilized for a write cycle, whereas approximately 75% is utilized for a read cycle. Thus, in any cache design, optimization for the read cycles should receive more importance.

1.5 Cache Performance:

Cache performance is measured in terms of the miss rate. This is the probability that a requested reference is not available in the cache. The miss rate times the miss time measures the "delay penalty" due to a cache miss. In most processor designs, the processor ceases activity and must stall when a cache miss is encountered. Thus, a cache miss behaves in much the same way as a pipeline break.

1.5.1 Cache performance improvement process:

The average memory access time provides a metric for optimizing the cache for improved performance [38]:

Average memory access time = Hit time + (Miss rate x Miss penalty)

Thus, cache optimization could be accomplished by reducing any of the following factors, which are directly contribute to the overall cache performance:

a. Miss rate

b. Miss penalty

~~d.~~c. Average hit-time in the cache.

Two simple and classic techniques for reducing miss rate are using larger block sizes and higher associativities in the cache memory (for set-associative organizations). Improving one aspect of the average memory access time comes at the expense of another. Larger block sizes take advantage of spatial locality, but at the same time can cause an increase in the miss penalty. Similarly, greater associativity reduces the miss rate at the expense of higher hit time.

1.5.2 Split Instruction & Data Cache:

A split Instruction (I) and Data (D) cache provides the designer with the possibility of significantly increased cache bandwidth, potentially doubling the access capability in the cache. Split I- and D- caches are particularly useful when the instruction bandwidth is higher than data bandwidth. Split I- and D- caches come at the expense of having higher miss rates than unified caches. This is due to two main reasons, a) relative cache sizes, and b) adaptation to the changing ratio of instruction and data elements of a running program. An 8kB unified cache can provide more flexibility in instruction and data storage requirements when compared to a divided 4-KB Instruction and 4-KB Data cache. In a unified cache, the cache replacement process intelligently adapts the cache for the changing ratio of instruction and data elements during the execution of a program. However, such an adaptation is not possible in a split I- and D-Cache. In modern processor design achieving higher memory access bandwidth is more desirable and the fabrication of separate I- and D- cache with considerable size to avoid potential miss rate increase is possible. Most modern processors now employ separate I- and D- caches in their organizations.

1.5.3 Two or Multi -level caches:

Another useful technique for reducing the miss rate is to use a two or multi-level cache organization. In the case of a two level cache, a small, fast on-chip cache is used as primary or level one (L1) cache, and a separate second cache (usually larger than L1) is used as a secondary, or a level two (L2) cache. Miss rates can be reduced by up to 10% by carefully tailoring the L1 and L2 cache sizes [5]. This type of organization may be expanded into a multi-level cache by using additional cache levels. The potential problem of using multi-level caches is the look-up penalty that results in cases where the required data is not present in any level of the cache. The lookup-time could increase significantly with the increase of the depth of the cache. Figure 1.4 shows the typical lookup penalty that arises with the depth of a multi-level cache.

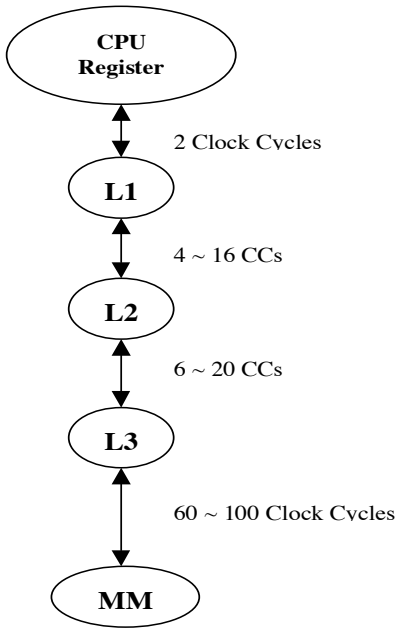


Figure 1.4 Lookup penalty with the depth of the levels in a multi-level cache

The intuitive argument for adding multiple caches in the cache hierarchy is that the increasing performance gap between the processor and main memory can be reduced by using several smaller accesses to main memory.

1.5.4 Blocking & non-blocking Cache:

In a blocking cache, the processor halts processing on a miss until the missed line is brought in the cache. This can result in frequent stalls. In a non-blocking cache, the processor is allowed to continue instruction execution without stalling if no true data dependency exists between instructions and data. A non-blocking cache organization pre-fetches data to avoid frequent misses. Proper anticipation of the required data plays a vital role in non-blocking cache performance.

1.5.5. Pseudo-associative Cache:

A pseudo-associative cache is used with a direct mapped or set-associative cache to increase the hit-speed and reduce the miss rate respectively. In this approach, before going to the next lower level of memory during a miss, another cache entry is checked in the pseudo set for a hit. The address of the pseudo set is calculated by inverting the MSB (most significant bit) of the index field of the cache address. This approach provides a variable hit time, and reduces the average memory access time as compared to using a direct mapped or set associative cache organization. Although this is an attractive process, it is not preferred for practical implementation due to the complications that arise in the design of a pipelined processor.

1.5.4 Victim Cache:

One recent technique of reducing miss rates is to use a victim cache. Figure 1.5 shows the organization of a CPU architecture with a victim cache. A victim cache is typically a small, fully associative cache located between a main cache and the refill path, and contains the blocks that are discarded from the main cache due to a miss. These victim blocks of data are checked during a miss to determine whether they contain the desired data before going to the next level of memory. If the data is found in the victim cache, then the victim cache block and main cache block are swapped. While fully associative caches are expensive to build in terms of logic, the size of this very small supplemental cache makes it feasible to implement on chip, along side the main level one (L1) cache.

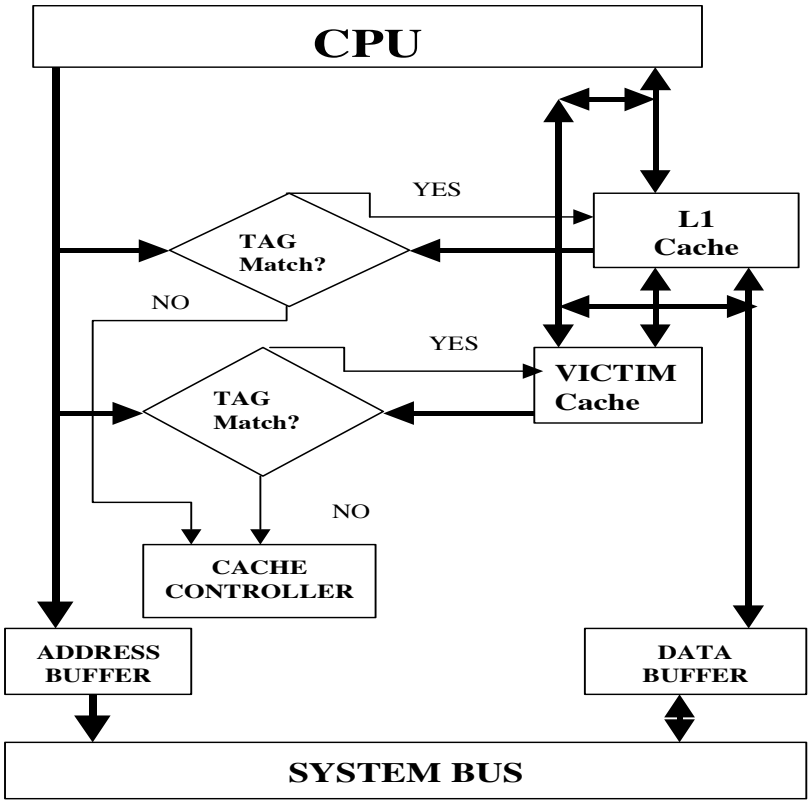


Figure 1.5 Victim cache placement in the memory hierarchy

Because it is not as fast as a direct-mapped or set-associative cache, the victim cache is not placed in the critical path of the processor. This means there is still some additional penalty associated with satisfying a reference via the victim cache rather than the main L1 cache. However, the penalty is generally on the order of 1 cycle instead of the 4-16 cycles often required for accessing off-chip L2 caches. A four-entry victim cache can remove 20% to 95% of the miss rate in a 4-KB direct mapped data cache [38].

1.5.5 Write Buffer:

On a write operation in a write-through cache, the cache suffers from the slow main memory write cycle time to finish a write operation. To hide the main memory write cycle delay from the cache and allow the cache to continue its operation at cache speed, a small write buffer can be used to temporarily store the data. Write buffers are very effective for improving the write cycle time of the cache. However, buffering the data can create memory consistency problems when the buffered data is not yet written into the main memory while the cache is updating the same location on a read miss from the main memory. To eliminate this problem, commercial processors implement the write buffer as a few-entry fully associative cache. On a miss, it makes an associative search in the cache and main memory. If the data is still in the write buffer, than it supplies the data directly from the write buffer to the microprocessor. This scheme is similar to a ***'victim cache'*** as described in the previous section.

1.5.6 Multimedia Cache:

Stream data processing is becoming a common factor in modern computer architectures due to the heavy usage of the Internet and the popularity of multi-media applications. In multimedia system designs, it is common to separate the data and control paths to simplify and optimize the hardware and software in order to handle the large volume of data traffic. Often in these systems, the video information passes directly from the network interface to the display unit without intervention by the CPU. This mechanism is highly effective at providing a support mechanism for multimedia applications without the high bandwidth data streams consuming CPU time. Following this strategy incurs the disadvantage of precluding the processor from accessing the multimedia data. This eliminates an interesting range of applications where processor intervention is necessary. A balanced architecture would keep data away from the processor when not necessary, but still enable high-speed access by the CPU when the application demands it.

It is particularly helpful to use a special type of cache to address this situation, referred to as a "stream-cache" (S-cache). When processing a data stream, it is likely that the data will be accessed in order of arrival. Hence, an S-cache holds the most recent data from the stream, and new data is written over the oldest data. In S-cache architectures, data arriving from the stream is placed directly into the cache, not passing through the main memory. This avoids unnecessary buffering. A section of the cache effectively becomes a circular buffer holding the latest stream information. Update of the S-cache content is asynchronous rather than triggered by a CPU cache miss. When the CPU attempts to access the stream data, there are three possible outcomes:

1. The data has recently arrived in the stream. In this case, the item is found in the cache and processing continues.

~~3.~~2. The data has not arrived yet. This is treated as a cache miss and the CPU may be blocked until the required item arrives. If the data does not arrive for some (long) period, the operating system may choose to reschedule the CPU.

~~4.~~3. The required data is far in the past accessing order that the buffer can hold. This case should be flagged as an exception to the operating system and represents "staleness" in terms of temporal locality.

The stream data will need to be addressed in some way. Most streams include a frame or temporal structure used by the application. This can be conveniently mapped into a range of processes in the virtual address space. Thus, the process may access the stream as an array indexed by frame number as shown in Figure 1.6.

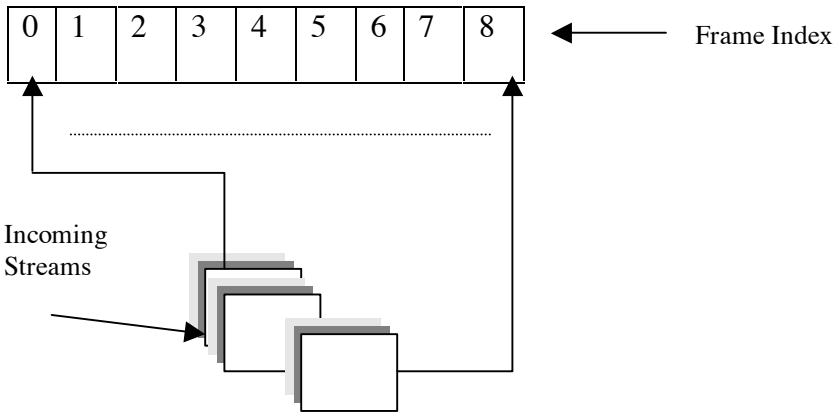


Figure 1.6 Addressing a stream

There are two major advantages to a stream cache system. First, the data from the stream is placed where it is going to be used, namely in the CPU cache. Hence, even if data is

not accessed in a strict temporal order, recent information can still be found in the cache. Second, the hardware manages fine grain resynchronization with the stream, imposing no overhead other than the necessity to wait for the data to arrive. The stream cache could be optimized for improved performance by carefully designing the cache size and architecture.

1.6 Cache Policies:

Cache policies are the rules of operation of the cache and are used to answer the following questions. During which cycles can data be read from the cache instead of main memory? Where does the cache fit into the system? How associative is the cache? What happens during write cycles? Cache policies are chosen for a single motive [6,11]; the designer wants to get the most performance for the lowest cost. Two variables play into this tradeoff: 1. Which is more important, to save engineering time or to save overall system parts cost? 2. Is the cache to be integrated or constructed from discrete components?

Cache policies may be chosen in a number of ways, depending on the generality of the system and the amount of resources available to improve the design. In the best case, the hardware and software of the system are designed together (referred to as "hardware/software co-design"). In this approach, the hardware can be optimized to a very good degree based on a large amount of empirical results on the effects of different caching policies on the intended software's performance. In the worst case, the hardware designer is asked to design a cache without any knowledge of the software that will run

on the system, no empirical data, nor any chance to develop any, and with very little knowledge about the tradeoffs of various cache policies [6].

There are different caching policies and organizations being utilized in different computer architectures, and efforts are being made towards their improvement. Though a tremendous amount of research is ongoing for achieving an optimum performance cache organization, an optimal solution has not been found. It is has been shown [48] that most programs require considerably less cache memory than what is available in a typical superscalar processor.

1.7 Cache Pollution and Bandwidth Waste:

Current caching policies in use by most computers result in cache pollution and memory bandwidth waste. This is due to pre-fetching a memory block or data cluster into the cache when a cache miss occurs without performing any data locality analysis. The pollution is due to the placement in cache of a non-reusable block whereas the memory bandwidth waste is caused by the additional data brought from a L2 cache to a L1 cache in the same block as the requested data. To cope with this issue, some microprocessors provide memory reference instructions that can bypass the cache [10].

Blind caching policies can also create similar cache pollution and memory bandwidth waste problems when the data references exhibit temporal locality. In the case of temporal locality, only one data element is being referenced from each block of data. Thus, the cache becomes full with unusable data elements. It has been reported in recent

investigations [48, 49, 50] that a large percentage of references exhibit temporal locality, and a significant percentage of references do not show any type of locality.

1.8 Motivation for this Research:

The research presented here is motivated by finding an alternative cache organization that will be able to use the cache resources more efficiently. A split data cache organization is proposed that exploit the full benefit of different types of locality references in a running program. Based on this motivation, the goal of this research is to design, simulate and investigate the performance benefits obtainable using the above mentioned cache organization in the hardware abstraction level which will perform dynamic locality prediction during runtime using only hardware resources. Before presenting the contribution made through this investigation, a brief survey of other efforts is presented in the next section.

1.9 A review of the current results:

Several investigations have proposed different schemes for instruction and data cache organizations to reduce overall memory access latency. These include a *transient value cache* (TVC) [37], lockup-free cache [20,47], cache-conscious load scheduling [27], hardware and software pre-fetching [16,17,18,19,20,21,22,29,41,42,43] and multithreading [28]. TVC uses a small data cache in addition to a L1 cache to provide support for large fraction of parallel loads in a massive parallel-processing environment. The mechanism proposed in [28] identifies non-cacheable data by means of profiling. The scheme proposed in [56] is based on a run-time managed history table of the most

recent load/store instructions. In [45], a pre-fetch engine is used which relies on software or hardware optimized *Deterministic Prediction Approach* (DPA) in order to pre-fetch data that is estimated to be referenced in the future.

Compiler assisted optimization of the cache data locality is proposed in [23,24,25,26,27,32,33,34,35]. Compiler based optimizations are based mainly on improved algorithms which use several techniques to identify locality in loops in scientific codes, and perform data layout transformation to provide optimum locality for better cache performance.

Combined compile-time and run-time caching policies as proposed in [46] use memory access detection, and automatic data caching based on compiler provided analysis of run-time memory access requirements. This is considered as an efficient approach in a shared memory parallel computing on distributed memory machines. In this approach, if the compiler analysis fails entirely, then the run-time maintenance of the shared memory is done with the hardware resources. Therefore, the complexity and limitations of compilers that directly target message passing [44,46] can be avoided.

Run-time memory performance feedback and memory layout optimization is proposed in [55,56]. In [55], the processor is informed about the memory operation by using the cache outcome condition code and cache miss traps so that the processor can tackle the performance requirements by using in-built hardware supports. This approach is based on the observation that modern in-order-issue and out-order-issue superscalar processors

already contain the bulk of the necessary hardware support. In [46], a system is proposed that uses a memory layout oriented approach to exploit cache locality for parallel loops at run-time on Symmetric Multi-Processor (SMP) systems using application dependent hints and the targeted cache architecture.

In [41], a programmable pre-fetch engine is used in the on-chip cache. As more chip area is available due to the tremendous advancement of the VLSI technology, designers can take advantage of using such programmable chips to hide the main memory access latency. This pre-fetch engine can pre-fetch data without any compiler intervention during run-time. The pre-fetch engine is programmable by software, allowing the designer to optimize the cache performance by using improved software algorithms to program this pre-fetch engine. Though pre-fetching always increases data traffic in the bus, the proposed scheme claims that additional data traffic can be significantly reduced by using the programmable approach of the hardware, and benefits from both software and hardware.

The selective caching policy proposed in [47] leads to an organization similar to a conventional cache in which all memory instructions have an additional bit set (or reset) by the compiler. During a cache miss, this bit controls whether a new block should be retrieved from the L2 cache and placed in L1 cache, or if the requested data should be retrieved from the L2 cache directly without updating the L1 cache.

A cache organization with both temporal and spatial subsystems has been proposed in [48,49,50,52]. This organization uses a very simple heuristic based on the data type which can be changed by dynamic or pre-runtime profiling. Selective caching is a feature of current microprocessors such as that being used in the PowerPC. The HP PA-7200 [51] uses a software-managed data caching policy. Every memory instruction used by the HP PA-7200 includes a "hint bit" indicating that spatial locality is used to predict if the data referenced by that instruction shows only spatial locality characteristics and not temporal locality. The HP PA-7200 consists of two cache modules; the on-chip fully associative assist cache and a large direct-mapped off-chip cache. The assist cache holds data related to all memory references for which hint bits are explicitly set indicating spatial locality. The off-chip main cache holds all data in which the hint bit is not set indicating the lack of spatial locality.

To avoid cache pollution, intelligent spatial pre-fetching schemes have been proposed [36,57]. In [36], a *Spatial FootPrints* (SFP) table is maintained by using specialized hardware. Depending on the content of the SFP table, the predictor mechanism fetches a smaller or larger number of blocks when misses occur in the cache. Also, in [57] a somewhat similar strategy based on a *Spatial Locality Detection Table* (SLDT) is used to prefetch multiple data blocks or less in order to reduce memory access latency during runtime.

In [12,13], considerable performance improvement was shown by using a stream cache unit with a conventional cache. In this strategy, hardware based reordering of stream data was used to improve cache performance. The logic behind this approach is that, the

performance of most memory systems is dependent upon the order of the requests presented to it. Access ordering refers to any technique that changes the order of memory requests to increase bandwidth. Stream data, such as vector (scientific) computations, multi-media (de)compression, encryption, signal processing, text searching, etc., are affected more by bandwidth than by latency.

Table 1 [38] shows the comparative performance benefits obtainable from different cache scheme compiled in [38].

Table 1[38]. Summary of different cache optimization schemes. In the table + indicates it improves the factor, -indicates it hurts the factor. Hardware complexity factor 0 indicates easy to implement, and 3 indicate more complex to implement.

Technique	Miss rate	Miss penalty	Hit time	Hardware Complexity	Comment
Larger block size	+	-		0	Trivial; RS/6000 550 uses 128
Higher associativity	+		-	1	e.g., MIPS R10000 is 4-way
Victim caches	+			2	e.g., HP 7200
Pseudo-associative	+			2	Used in L2 of MIPS R10000
Hardware prefetching of instruction and data	+			2	Data are harder to prefetch; Alpha 21064
Compiler controlled prefetching	+			3	Needs nonblocking cache too
Compiler technique to reduce misses	+			0	Software is challenge
Giving priority to read misses over writes		+		1	Trivial for uniprocessor, and widely used
Subblock placement		+		1	Used primarily to reduce tags
Early restart and critical word first		+		2	Used in MIPS R10000, IBM 620
Nonblocking caches		+		3	Used in Alpha 21064
Second-level caches		+		2	Costly hardware; widely used
Small and simple caches	-		+	0	Trivial; widely used
Avoiding address translation during indexing of the cache			+	2	Trivial if small cache; used in Alpha 21064
Pipelining writes for fast write hits			+	1	Used in Alpha 21064

1.10 Research Contributions:

The research contributions made are documented in this dissertation and in [63,64,65]. The contributions presented in this dissertation are:

a. The performance bottleneck of the caching scheme used by most current microprocessor architectures is identified, and reviews of different caching technologies suggested and implemented by various researchers are highlighted.

b. Run-time cache resident data locality analysis of the memory access patterns of a wide variety of application programs defined in SPEC92 benchmark suite [62] are presented using the results of a simulation of real-time data cache access. This analysis presents a clear understanding of the data locality behavior of the common application programs. The average cache resource requirements for the spatial and temporal address spaces used by the programs are also identified. This contribution provides valuable information for the designer of a cache subsystem.

c. A locality estimation technique and subsequent circuit is designed which can perform run-time prediction of the data memory access locality shown by the programs [64]. The novelty of this prediction circuit is that it is simple enough to reduce penalties due to increased hardware complexity, yet it can also provide better performance in all cases of the SPEC92 benchmarks. In addition, this technique does not require any compiler assistance and is independent of any particular computer platform.

d. The design of a new split data cache organization is presented. This organization uses two sub-caches termed as the Spatial Sub-Cache and temporal Sub-Cache. It stores data with appropriate locality as predicted by the locality estimation circuitry into these sub-caches for efficient cache management and thereby improves the overall memory access efficiency of the microprocessor.

e. A detailed evaluation of the locality estimation circuitry and the split data cache subsystem. A simulation prototype is written for the split data cache model using the *C* programming language and is implemented and tested on a *UNIX* platform using memory address traces of data for load/store instructions of the *SPEC92* benchmark suite.

1.11 Dissertation outline

The remainder of this dissertation is organized as follows. Chapter 2 presents the analysis of the memory access behaviors of different benchmark programs during their runtime residency in the cache. Chapter 3 presents a hardware scheme used to predict the possible locality behavior of the accessed data during the execution of a program. Chapter 4 presents the organizational design of a split data cache along with its implementation strategy. Chapter 5 presents the simulation of the designed split cache model and an evaluation of its performance. Finally, chapter 6 summarizes the research findings with conclusion and future research direction.

CHAPTER 2

Cache Resident Data Locality Analysis

The organization of the data cache can significantly affect overall data access latencies during program executing. The cache performance depends on the locality characteristics of the data being processed in a program as well as the underlying architecture. A typical program has a data access profile that exhibits both temporal and spatial locality characteristics. Since most processors contain single data caches at a given level, and a single data cache cannot be optimized for purely spatial nor purely temporal locality data accesses, cache space pollution and inefficient usage of cache resources can occur. In the worst case, these phenomena can actually introduce additional data access latencies through repeated line fills. Here an analysis and modeling scheme is presented that describes the runtime data access behavior of several benchmark programs in a typical, unified data cache. The motivation for the development of this model is to produce

information that may aid in the design of a split data cache with one side optimized for temporal locality accesses and the other for spatial locality accesses.

2.0 Introduction

A cache memory subsystem pre-fetches additional memory data during a miss along with the requested data word by the processor. The amount of pre-fetched data depends on the line size of the cache. With data pre-fetching, a cache memory can hide data access latencies by exploiting the locality characteristics of the running programs in the pre-fetched lines. Pre-fetching a greater amount of data helps to hide latency rather than reducing the latency. The main problem is that pre-fetching can aid in cache performance only when additional memory bandwidth is available. This is because pre-fetching does not decrease the number of memory accesses; it simply tries to perform them over a shorter period. The available cache space and bandwidth may be polluted and misused by pre-fetching when a large amount of non-usable data is resident in the cache. About 60 percent of available space in a cache can be polluted in some extreme cases due to this phenomenon [57]. In cases where the program is already memory-bandwidth limited, it becomes impossible for pre-fetching to improve performance. Alternatively, locality optimizations such as cache blocking [29] can actually decrease the total number of accesses to memory, thereby reducing both latency and required bandwidth.

A good knowledge of memory access behavior characterized by the locality of references can lead to efficient cache memory subsystem designs. Locality analysis of different types of programs during runtime aid in defining an optimized cache subsystem

organization. The locality behavior of a program is categorized as either, spatial or temporal. Most current research projects [36,57,58] are investigating the spatial reuse of data and strive to find a means to exploit this spatial reuse of data.

Past research efforts [27,31,32,33] have sought to optimize program loop nest localities. Different models and reorganizations of loops have been proposed using tiling, compound transformations consisting of loop permutation, loop fusion, loop distribution and loop reversal [31] to increase temporal and spatial locality in loop-nests. These techniques are primarily compiler-based approaches. Programs must be compiled using the target machine's compiler to gain the optimization benefits.

Cache organizations based on compiler optimizations or based on identifying spatial reuse may produce poor performance when running a variety of different application programs. This poor performance is due to the particular bias of optimizations for a specific subset of the application programs. Usually, the average data access by the programs are both from spatial and temporal locality. Analysis of the run-time memory-data access plays a critical role in this respect.

The question still remains as to why use a run-time locality analysis model to design a data cache when compilers exploit detailed information from applications to optimize locality. Compiler based locality optimization can perform very effectively to improve the performance of those applications to which they can be applied. In reality, many dynamic data access patterns of the applications cannot be analyzed during compile time.

For example [56], consider the sparse matrix multiplication program shown in Figure 2.1-1. In the innermost loop, the array elements A[k] and B[r] are indirectly determined by the data in the arrays Arow, Acol, Bcol and Brow. These indirect data accesses cannot be determined by the compiler since the compiler has no idea about what kind of data the program is going to process during run-time. Therefore, if we want to optimize the data access locality for such a case, only run-time locality analysis can optimize cache performance.

```
double  A[X], B[Y], C[M][M];
int Arow[M+1], Acol[X], Bcol[M+1], Brow[Y];

sparse-mm()
{
        int i=0, j=0, k, r, start, end;
        register double d;
        for(;i<M;i++)
                for(;j<M;j++)
                {
                        d=0;
                        start=Bcol[j];
                        end =Bcol[j+1];

                        for(k=Arow[i];k<Arow[i+1];k++)
                                for(r=start;r<end;r++)                    → task t(i,j)
                                        if(Acol[k] == Brow[r])
                                        {
                                                d+=A[k]*B[r];
                                                start = r+1;
                                                break;
                                        }
                        C[i][j] =d;
                }
}
```

Figure 2.1-1 Code block of a Sparse Matrix Multiplication, which has a dynamic data-access pattern and an irregular computation pattern

A cache sub-system, that targets the true run-time data access of the program can improve the cache performance significantly. The advantage of using such a scheme is that since it is not designed for specific program sets nor does it depend on compiler assistance, in many cases can provide better performance.

In this chapter, a model determining the locality behavior exhibited by several benchmarks programs executing in a load/store based uniprocessor with a typical unified data cache is presented. Locality analysis results using this model are also presented. The motivation for performing this analysis is to determine the data locality behavior of different programs, and to use the results to design an efficient cache organization that will not suffer from the inability to exploit varying data locality behaviors over a variety of executing programs.

The subsequent sections of this chapter are organized as follows. Section 2.1 presents an overview of locality and the need for runtime locality analysis and modeling of executing programs. In section 2.2, the model used for the locality analysis is presented. Next, experimental results of the cache access behavior by different SPEC integer and floating point programs are presented and discussed. Finally, section 2.4 provides the conclusions based on the experimental data.

2.1.1 Principle of Locality

To hide memory access latency due to fast processors with relatively slower main memory, a cache subsystem is used to attempt to store data, which will be accessed in the

future by the processor. This is accomplished by loading additional data other than that being requested by the processor during a cache line fill. A typical way of doing this is to retrieve additional data from the neighboring address space of the requested data. The purpose of writing neighboring data into the cache is to exploit the principle of spatial locality. Spatial locality exists due to the empirical observation that "data tends to be accessed that is close (in address space) to previously accessed data". Figure 2.1-1 shows an example of this type of locality. Data block B is requested by the processor and the resulting cache miss causes a line fill to occur that loads blocks A through D. Thus, any consecutive memory blocks requested by the CPU within this spatial region will result in a cache hit with the access time equal to the (faster) cache access time..

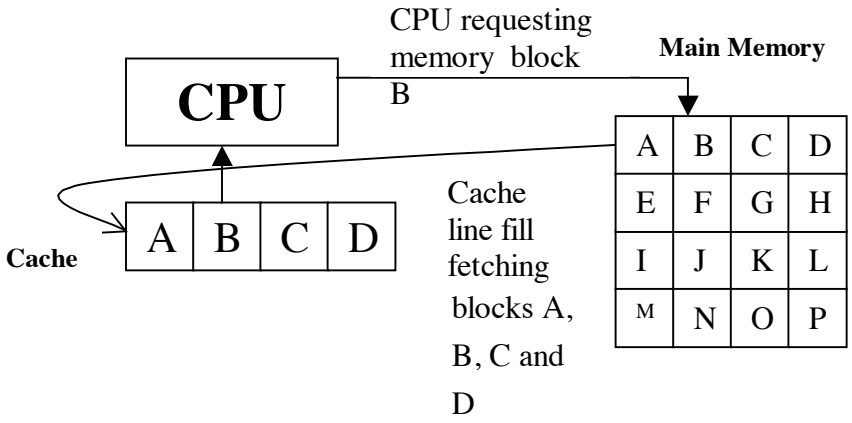


Figure 2.1-1: Cache line fill illustrating the "spatial access"

Whenever the data access pattern is largely spatial in nature, the inclusion of large cache lines that contain more neighboring data can reduce the overall memory access latencies drastically. For strictly spatial data access patterns the reduction in memory access latency depends mainly on the cache line size.

Another type of locality is "temporal" locality or locality in time. This type of locality is characterized by certain locations in memory being accessed repeatedly in time. For example, this occurs when a CPU requests data blocks in the order B, G, M, B, G, M repeatedly during the execution of a program. The illustration shown in Figure 2.1-2 depicts this type of access pattern. In this case, the cache line fills are bringing additional memory blocks in each cache line that is not used by the processor.

2.1.2 Motivation for Locality Analysis

In past work on data cache optimization, mainly numeric (scientific) programs have been considered for analysis of the data locality pattern. Since most numeric codes contain a large amount of nested loops, a significant amount of research has been attributed to the incorporation of more spatial reuse through different compiler optimization techniques such as unimodular transformations, loop fusion and distribution and tiling [66]. Some assertions of the spatial reuse of data have been made without doing any intra-loop reuse analysis [31]. Some computer architectures, such as the HP-7200 [51] do not use any detailed program locality information and depend only on spatial reuse of data.

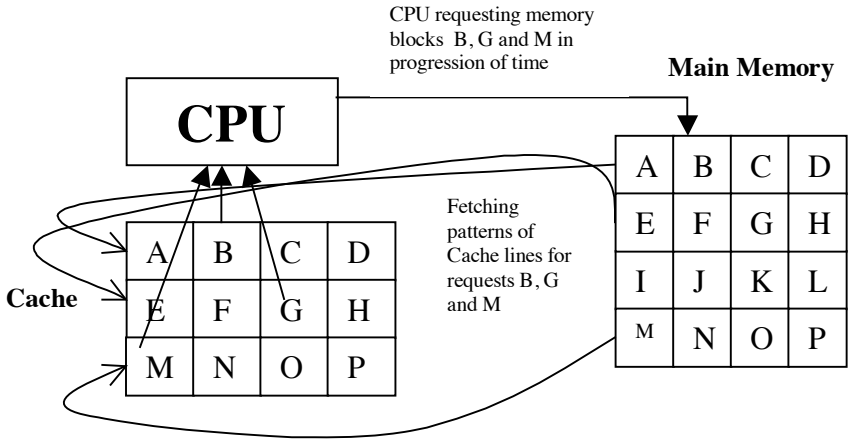


Figure 2.1-2: Cache line fill illustrating the "temporal access"

To fully take advantage of the spatial locality present in a program's data access patterns and to also benefit from the temporal locality that is also present, a data cache may be organized with multiword line sizes. In Figure 2.1-1 it is seen that for the spatial access pattern B, C, D and A the access penalty is one cache miss since the next three consecutive accesses result in a cache hit. Thus, the effective miss rate is 25 percent and cache space utilization is 100 percent for this case. From Figure 2.1-2, if the access pattern is B, G, M, B, G, M then the effective miss rate is increased to 50 percent and cache space utilization is reduced to 25 percent. This clearly indicates that the relatively large line size used for taking advantage of spatial locality results in the pollution of the cache and also increases the memory access bandwidth. About 40% cache capacity waste is typical[57].

The depicted scenario indicates that the same cache organization will not perform equally well in all cases. To optimize performance, the cache organization must be tuned to benefit from both spatial and temporal data access behaviors. The tradeoff arises because increasing cache line size to exploit more spatial locality causes more cache pollution and wasted bandwidth when temporal accesses are requested. Alternatively, decreasing the line size and adding more lines to a cache can result in inefficient usage when the accesses are largely spatial in nature. Further, as is demonstrated later in this dissertation, the data access behavior varies largely from program to program. Data access behavior can be purely spatial, purely temporal or (more typically) a combination of both. It is possible to optimize a cache organization to provide optimum performance for a particular program. However, it is a very difficult task (if not impossible) to provide optimum performance for all types of program data access behavior. A reasonable choice in this case is to design a cache subsystem that will perform well on average. Analysis and modeling of the program data access behavior over a number of different programs can provide estimates of average-case behavior. This motivates us to carefully study and analyze the data access behavior of the programs that cover a wide range of applications. The SPEC benchmark suite has been used as a representative sample of different types of application programs.

2.2 Locality Analysis Method

The data locality behavior of different application programs is analyzed during runtime in order to observe the characteristics of interest. In the results presented here, parameters of interest are generated through the accumulation of statistics based on data access patterns

in a general cache during program execution. In this approach, specific cache architectures are considered and runtime data access profiles of different *SPEC92* benchmark programs are stored. Initially, different cache sizes with varying line sizes were modeled. Among these, a four-way set associative 32 KB cache with 128 byte (32-bit words) line size was considered as the baseline organization to analyze and model cache data locality in terms of miss rates, and a wide window width to capture both spatial and temporal locality. This target cache architecture was simulated using the ***C*** language and complied using the **`Unix cc`** compiler. Input to the program consists of memory traces gathered during the execution of the *SPEC92* benchmarks.

The memory traces of the *SPEC92* benchmarks used in this investigation are those available from the anonymous ftp site of the New Mexico State University Trace Database [62]. The traces contain the addresses of the memory references and a field indicating whether it is instruction address or data read/write address. Since the main interest is data caching, a filter program was written that extracted only the data load/store related addresses. The cache simulator then used the data load/store related traces as input and generated the analysis results after simulating the cache.

For locality profiling purposes, the simulator keeps track of the number of accesses in each line of the cache as well as the average time difference of each word being accessed in a line over successive hits, or the "temporal stride". Although the term "stride" is generally used to refer to the absolute distance between different memory addresses, here

it is used in a temporal sense to refer the relative time difference in terms of the processor clock cycles. The analysis tool records the number of hits for each word in a line. Analyzing the runtime behavior of the *SPEC92* benchmark programs' memory traces allows the data access locality characteristics of these programs to be noted.

For the locality analysis, the line hit-rate and strides of the words in the lines as well as word-hit frequency is used. Usually, for spatial locality, the strides of the words in a line should be similar or should have a fixed difference with an equal or close number of hits. For temporal locality behavior, the number of accesses to a line should become very high and we may expect that the strides of the words and word-hit frequencies will vary greatly. Figure 2.2-1 and 2.2-2 shows the typical nature of the strides for temporal and spatial locality in a cache line for two benchmark programs used in this test bench.

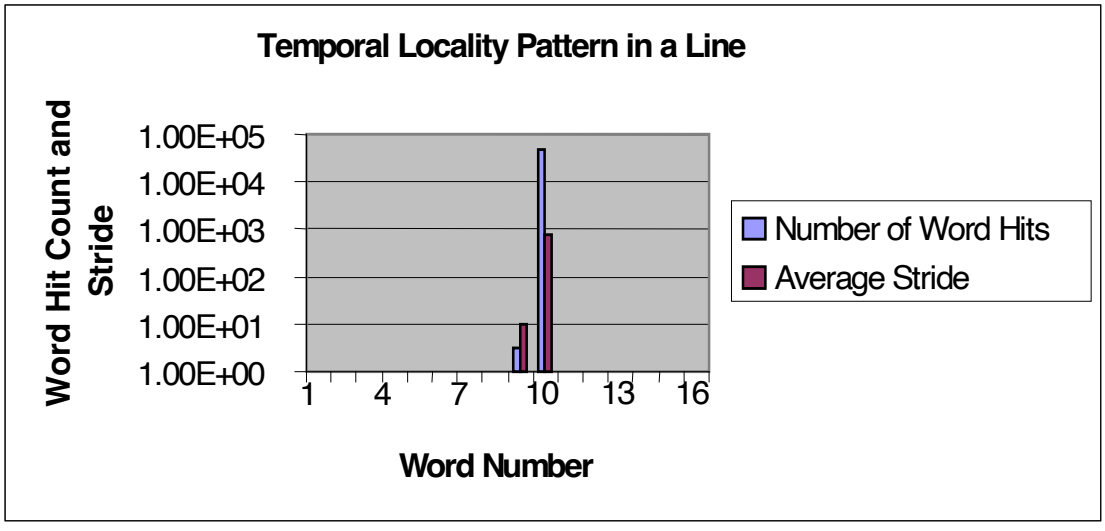


Figure 2.2-1: Temporal access pattern in a cache line

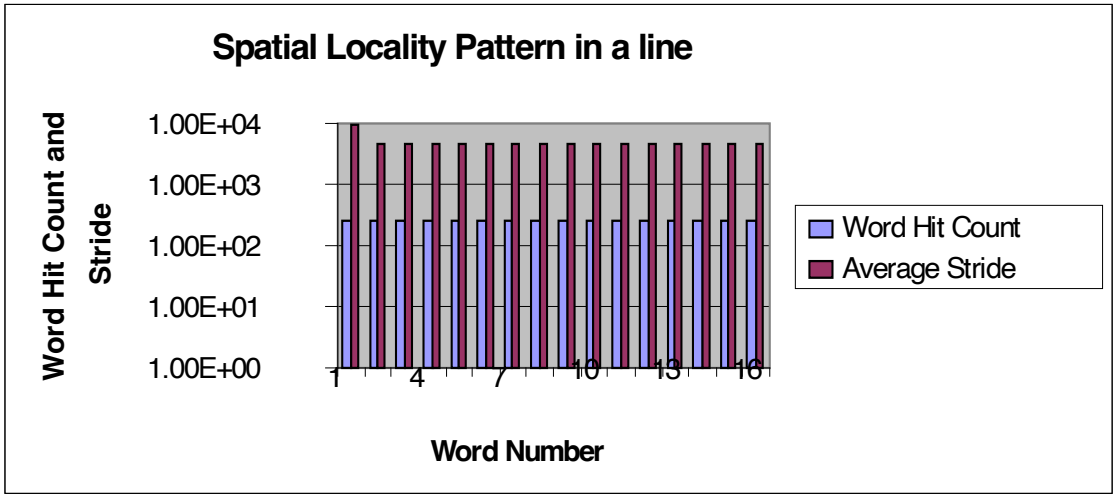


Figure 2.2-2: Spatial access pattern in a cache line

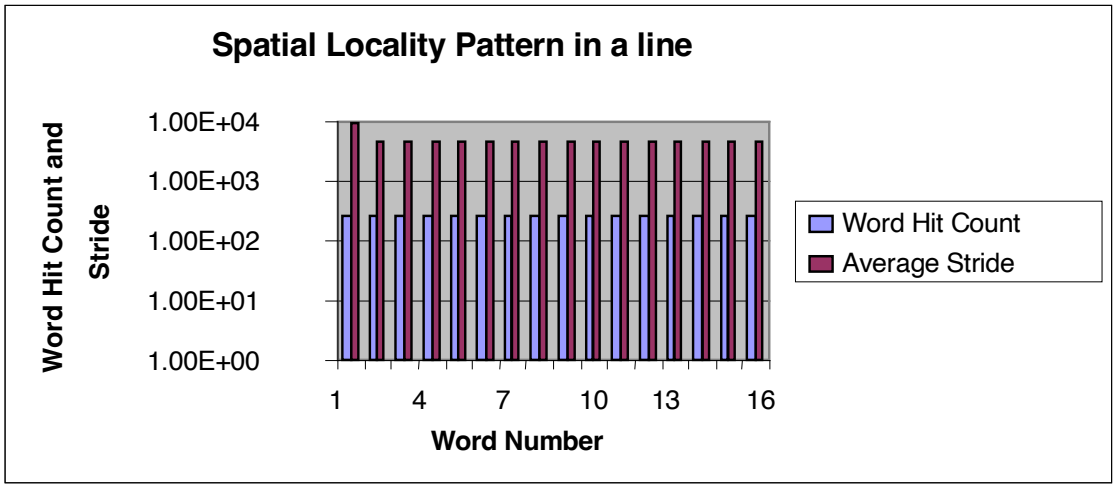


Figure 2.2-2: Spatial access pattern in a cache line

The following equation for the estimation of hit rate (in percentage) for spatial or temporal locality was used:

$$EHit_{Spatial/Temporal} = \left(\frac{100}{TWHC} \right) \sum_{i=1}^{N_{Spatial/Temporal}} WHC_i$$

Where:

$EHit$ = Estimated percent of Hits due to spatial or temporal Locality

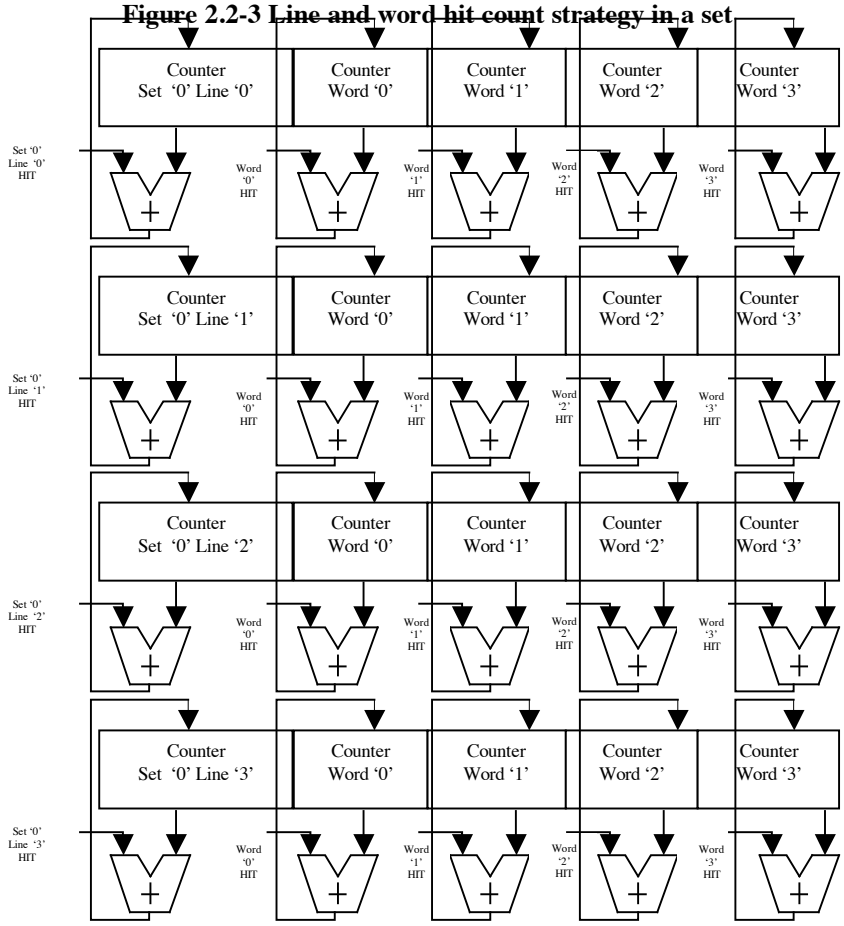
WHC_i = i^{th} Word Hit Count due to spatial or temporal Locality

$N_{Spatial/Temporal}$ = Number of Word Hits due to spatial or temporal Locality

$TWHC$ = Total Number of Word Hit Count in the cache

To facilitate this estimation process, the model uses counters for each line of each set in the cache and for all corresponding words in the lines. Two-dimensional unsigned integer array variables are used to store the count values. The mapping process of a 4-way set associative cache is used to gather the array indexes of the counter variables in a manner similar to hashing, where the hash function is actually the cache mapping function. These counters are used to maintain the hit counts for each word in each line of the sets. For each respective word in the cache, the average time between successive hits is also maintained in another variable in terms of memory access cycles that we refer to as stride (in this case, temporal stride) in the plots. Figure 2.2-3 illustrates this basic strategy of counting the hits for a single 4-way set that contains 4 words per line. Figure 2.2-4 contains a code fragment that shows how to calculate temporal stride values for successive word hits.

Figure 2.2-3 Line and word hit count strategy in a set




```
// Initially, before any memory load store operation the index variables are set to zero, so,
// Access_Cycle[Set & Line Index][Word Index] = 0
// Avg_Stride[Set & Line Index][Word Index] = 0
// Cum_Stride[Set & Line Index][Word Index] = 0
// Code fragment below showing the method of calculating average time stride calculation on HITs on
// words in the cache lines

Current_Access_Cycle = Mem_Access_Cycle;

if(MatchFound)
{
        Cum_Stride[Set_Line_Index][Word_Index] =
        Cum_Stride[Set_Line_Index][Word_Index] + (Current_Access_Cycle  -
                                                Access_Cycle[Set_Line_Index][Word_Index]);

        Access_Cycle[Set_Line_Index][Word_Index] = Current_Access_Cycle;
        Word_Hit_Count[Set_Line Index][Word_Index]++;
}

……
……
if(feof(Memory_Trace_File_Pointer))
{
   for(I=0; I<Number_of_Sets;I++)
        for(J=0; J<4; J++)
        {
        Set_Line_Index = ((I<<2)|J);
        for(Word_Index=0; Word_Index<Max_Word_Count_Per_Line;Word_Index++)
        if(Word_Hit_Count[Set_Line_Index][Word_Index]!=0)
        {
        Avg_Stride[Set_Line Index][Word Index] =
        Cum_Stride[Set_Line Index][Word Index]/ Word_Hit_Count[Set_Line Index][Word Index];
        }
        }
}
```

Figure 3.1-4 Code fragment for average time-stride calculation on Hits

As input, the analysis program uses memory traces obtained through the simulated execution of the *SPEC92* benchmarks assuming a load/store CPU with the cache structure described above. After processing the hit rate and average stride of all words in the cache, the portion of the cache hits due to spatial and temporal accesses is determined. This determination is based on the 'hit count' and 'average stride' values for each word in the cache, and is compared with the other words' hit count and stride values. For spatial accesses, the hit count and stride should be similar in value for each word in relation to the other words in a specific line of the cache. This observation forms the basis of how spatial locality is detected. The spatial accesses are isolated by simple relative comparisons of both the word and total line hit count values. For temporal accesses, the words with large differences in stride and hit count as compared with other words in the line are considered and their cumulative counts are recorded for each line. Following the same process for all of the lines in the cache, a combined set of statistics based on spatial, temporal and unused word counts are obtained to calculate the percentage of cache hits due to spatial versus temporal locality. Figure 2.2-5 shows a flow diagram illustrating the major steps of the analysis method.

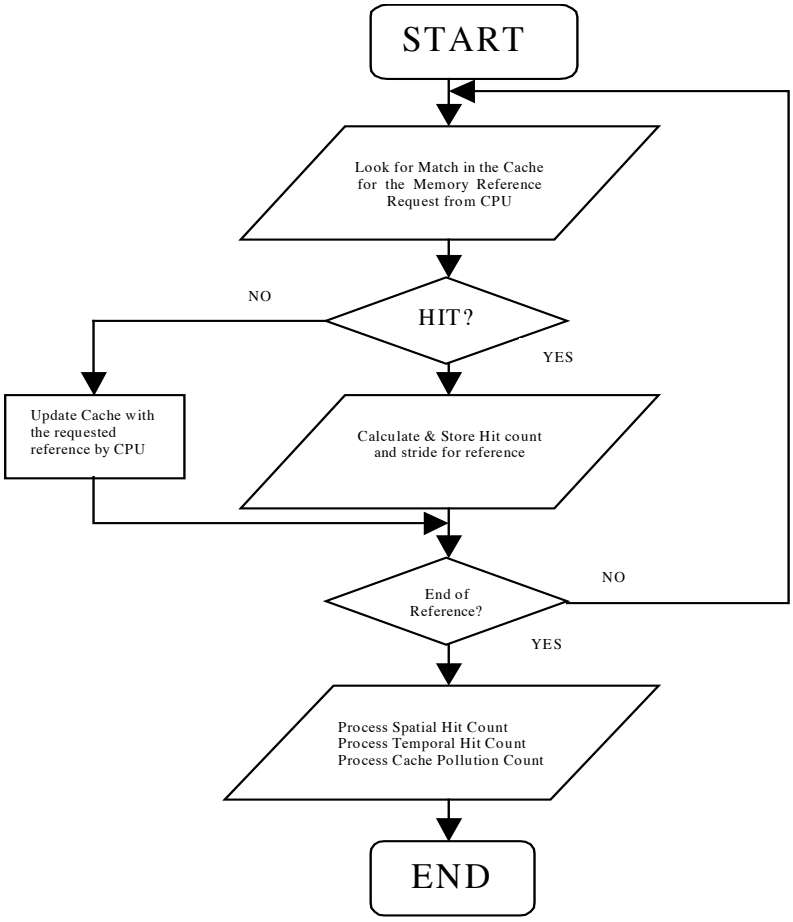

START
Look for Match in the Cache for the Memory Reference Request from CPU
HIT?
NO
YES
Update Cache with the requested reference by CPU
Calculate & Store Hit count and stride for reference
End of Reference?
NO
YES
Process Spatial Hit Count
Process Temporal Hit Count
Process Cache Pollution Count
END

Figure 2.2-5: Diagram illustrating the cache data analysis

2.3 Data Locality Analysis Results

Data locality behavior of several *SPEC92* integer and floating point programs is shown in Table 2.3-1. From the data locality behavior of the benchmark programs, it is apparent that the data access patterns do not show purely spatial or temporal locality in any case. The ratio of spatial versus temporal locality varies from program to program. These results indicate that the ***spice2g6, gcc*** and ***doduc*** benchmarks have a bias toward temporal locality. Table 2.3-1 also indicates that most of the benchmark programs possess a significant amount of temporal locality. The average spatial locality is 68 percent and average temporal locality is 32 percent for the SPEC benchmark programs used in this study.

Table 2.3-1: Locality behavior of some *SPEC* Benchmark Programs

Benchmark	Spatial Reuse	Average Spatial Reuse	Temporal Reuse	Average temporal Reuse	Cache Space Pollution	Average Space Pollution
Espresso	0.54		0.46		0.34	
spice2g6	0.38		0.62		0.25	
Doduc	0.45		0.55		0.01	
Li	0.54		0.46		0.07	
Eqntott	0.67		0.33		0.18	
Compress	0.63		0.37		0.01	
mdljdp2	0.64		0.36		0.28	
wave5	0.63	0.68	0.37	0.32	0.62	0.23
Tomcatv	0.99		0.01		0.14	
Ora	0.90		0.10		0.61	
Alvinn	0.79		0.21		0.15	
Ear	0.81		0.19		0.10	

Sc	0.55		0.45		0.40			
mdljsp2	0.49		0.51		0.31			
swm256	0.96		0.04		0.10			
Gcc	0.44		0.56		0.10			
su2cor	0.87		0.13		0.01			
nasa7	0.99		0.01		0.38			

The spatial and temporal locality distributions of the SPEC benchmarks are shown in Figures 2.3-1 and 2.3-2.

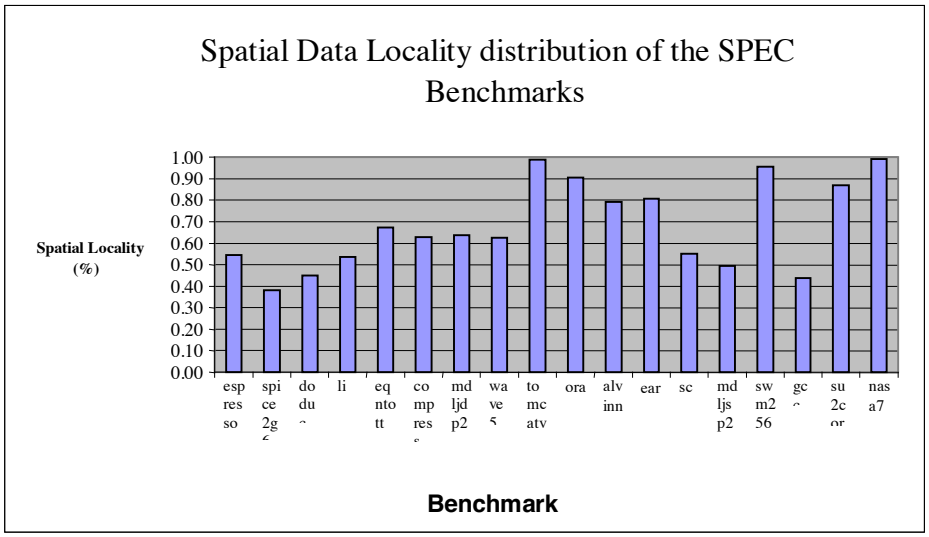


Figure 2.3-1: Graph showing spatial reuse patterns of the cache space by *SPEC* benchmarks

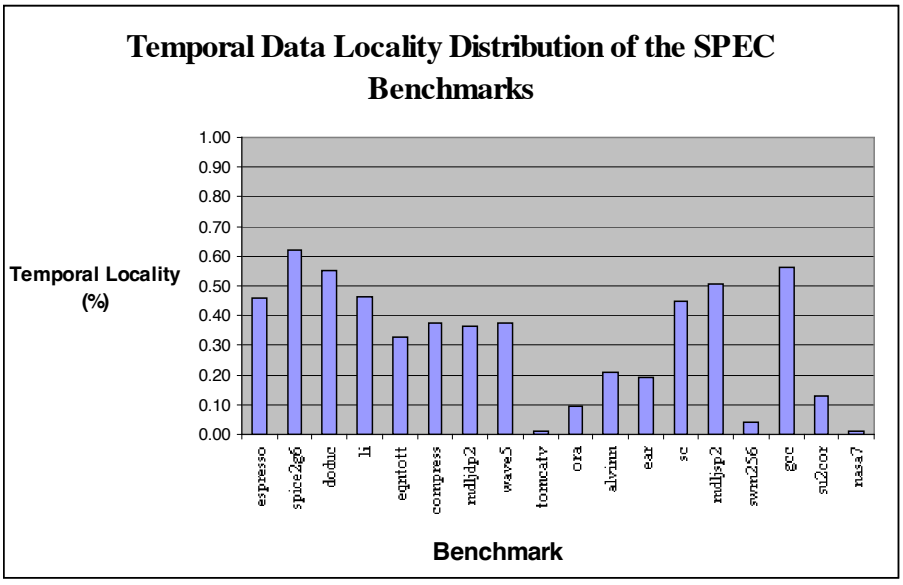


Figure 2.3-3 shows the pollution of cache space due to spatial fetching of data in the cache lines. The results suggest that on average, 23% of the available cache space be polluted by the spatial pre-fetching of data. In an extreme case the pollution was 62% (***wave5***).

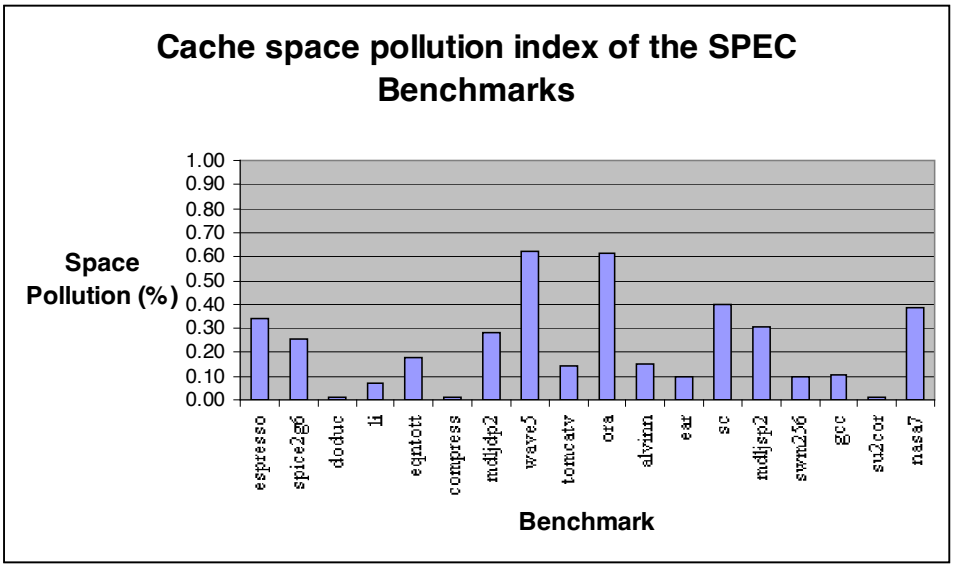


Figure 2.3-3: Cache space pollution for spatial fetching of data into cache lines by *SPEC* benchmarks

Figure 2.3-4 shows a 3-D plot of the portion of the cache space usage by the benchmark ***espresso***. This plot indicates that even when the spatial reuse component is dominant, the

reuse surface is not very uniform. The reuse frequency is very high in some lines. However, in most of the lines, spatial reuse is minimal.

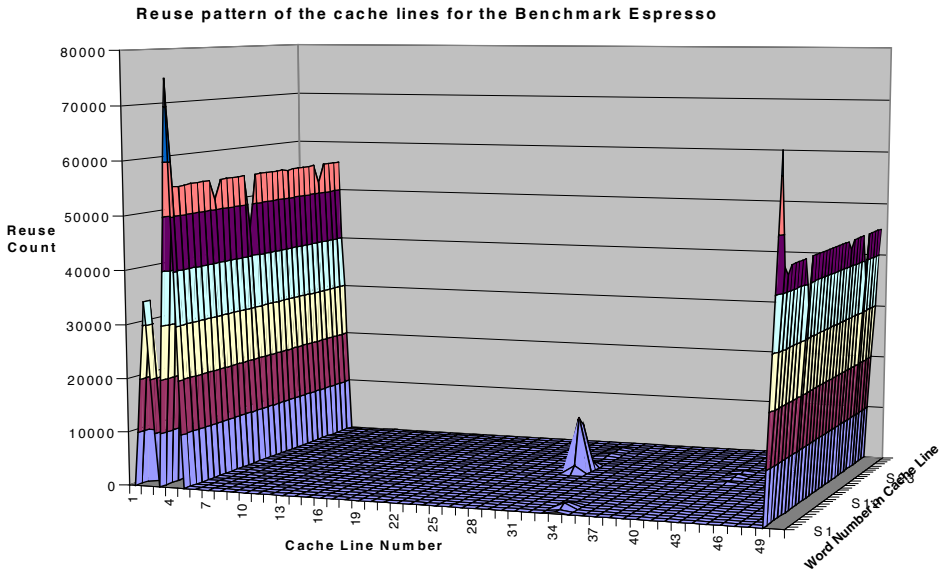


Figure 2.3-4: 3-D plot of the reuse pattern of the portion of cache space by the benchmark espresso

Careful analysis of the results suggests that the address space of the memory references could be pre-dominantly spatial, pre-dominantly temporal or a combination of each. This is illustrated in Figure 2.3-5 where set A represents accesses that exhibit spatial locality and set B indicates those with temporal locality. The results indicate that programs typically contain a subset of accesses that have characteristics of both sets A and B. The intersection of these two classes of memory access types is indicated by set C in Figure 2.3-5. As an example, consider a program that consists of several consecutive loops,

each of which accesses an array of data sequentially. Clearly, the accesses within a single loop are spatial in nature, however examining the access pattern of a single array element is temporal in nature due to the existence of multiple loops, and hence, multiple accesses of the same element.

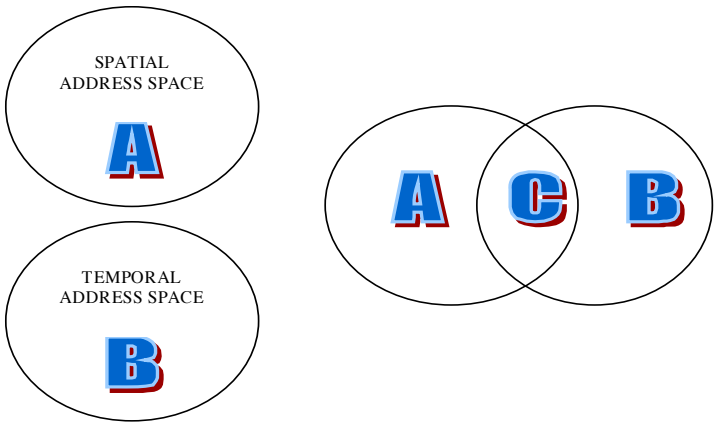


Figure 2.3-5: Diagram of overlapping spatial and temporal locality characteristics

2.4 Conclusions

Based on the locality analysis presented above, the following conclusions are made:

1. Run-time data access behavior of different programs needs to be supported. Thus, both spatial and temporal locality data should be cached. Therefore, a split data cache is justified to facilitate both types of locality.
2. A unified data cache can perform poorly in some cases by wasting valuable cache capacity.

3. The data that should be cached in a spatial cache are whose reuse frequency is good enough to allow for future cache hits. Otherwise, their accesses can be bypassed in the cache.
4. Since spatial reuse is dominant in most of the cases, a relatively larger spatial cache with bigger line sizes should be used as compared to the temporal cache in the split data cache.

CHAPTER 3

Dynamic Data Locality Estimation Circuit

A split data-cache architecture with separate caches for data accesses classified as predominately spatial or temporal requires specialized hardware or software to predict these characteristics. This chapter presents a locality estimation circuit that operates dynamically as the program executes. The technique is developed based on an analysis of the locality behavior of several benchmark programs as described in the previous chapter. The split data cache organization is then described and simulated. Experimental results obtained from the simulations are preserved. These results are of use in

determining the effectiveness of the dynamic locality-estimation circuit and the relative line sizes that should be used for the two caches.

3.0 Introduction

A data locality cache requires specialized hardware to predict the data access locality, and to determine in which cache the data should be stored. Run-time access behavior could show a random variation of locality of data from program to program. Performing compiler assisted profiling of locality before execution of the program is much easier in this case. Accomplishing the same result with a hardware scheme is more difficult due to the finite size of the hardware. The design of the prediction hardware should be simple and effective in any case to avoid complexity and minimize the additional hardware resources required. Complex locality-estimation hardware may provide best the locality estimation but the overall organization may introduce additional 'in-cache' locality computation time that effects the cache access time. With this in mind, a locality prediction hardware unit is designed which does not require any complex hardware scheme and uses only a simple protocol to estimate the data access locality.

The subsequent sections of this chapter are organized as follows. Section 3.1 presents a guideline to predict data locality analysis done in chapter 2. Section 3.2 describes a simple 'locality-estimation-circuit' to be included in the cache controller for dynamic prediction. Next, the performance of the locality prediction circuit when used in a split data cache organization as compared to the locality prediction with the statistical analysis is discussed. Finally, in section 3.4, conclusions based on the experimental data are

presented.

3.1 Data Locality Prediction Guideline

The locality analysis presented in chapter 2 provided insight to the overall data access behavior of the programs during run-time. This analysis model can be used effectively to define the guidelines for designing a locality prediction circuit. It has been seen that the data access behaviors exhibit uniform access and equal strides in most of the spatial accesses in a cache line. For temporal references, the access frequencies are quite high in some memory locations. Some temporal accesses are within very limited zones of the cache lines. It has been also observed that overlapped spatial and temporal accesses exist in some lines of the cache. Table 3.1 illustrates the spatial and temporal locality distribution of a few more benchmark programs in addition to that presented in Chapter 2.

Table 3.1: Estimation of locality type for Benchmark programs

Benchmark	Estimated Hit Rate (%) due to spatial Locality	Estimated Hit Rate (%) due to temporal Locality
LINPACK	35.36	64.64
MATMULT64	13.85	86.15
QSORT	50.03	49.97
WORDFREQ	20.48	79.52
CELLAUTO	62.51	37.49
QUEENS	0.01	99.99

From the locality analysis presented in Chapter 2, a guideline that the data cache needs support for storing data in two different sub-caches according to the locality bias can be inferred. These analysis results are used to propose a simple hardware solution for a split spatial and temporal data cache that allows for an overall improvement in caching efficiency. The approach followed is to implement a solution in hardware using dynamic locality estimation. This poses the problem of which cache to store the data in during cold-start accesses. At the cold-start point, no prior information is known about the data and an estimate of the locality would simply be a guess. Furthermore, the results in Table 3.1 indicate that depending on the functionality of the program, some exhibit predominately temporal locality while others exhibit spatial locality. The second guideline is that an estimation circuit should be designed to estimate the data access locality during run-time and then store those data in the proper locality caches. The next section describes the organization and working principle of a locality estimation circuit considered in this research for its simplicity and effectiveness.

3.2.1 Split data locality cache

The functional blocks of a generic split data cache is in Figure 3.1. This cache organization contains a dynamic locality-estimation circuit that controls the runtime caching policies for the whole organization. The dynamic locality estimation circuitry analyzes the locality pattern of recently accessed data in the cache and directs the next line-fill to the appropriate cache. This is accomplished by runtime locality analysis on hits occurring after the cold start of the cache.

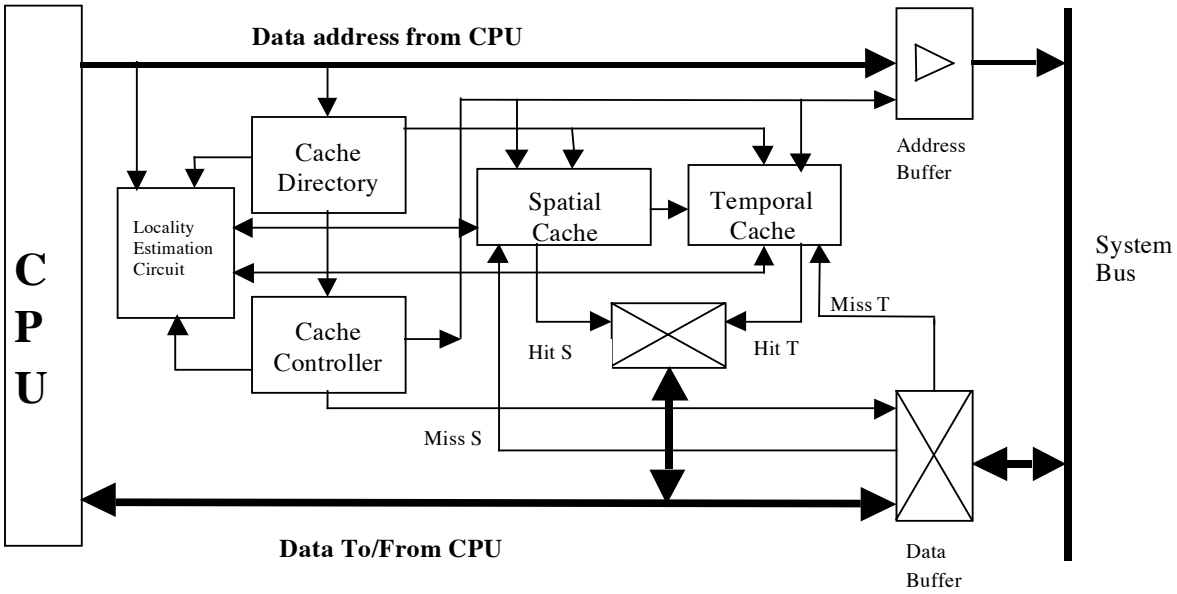


Figure 3.1 The functional blocks of the split data cache organization

3.2.2 Dynamic locality estimation scheme

Locality prediction hardware must estimate run-time data access patterns. This can be accomplished with the knowledge of the run-time data access pattern in the cache blocks. To store the access pattern information, we need to keep a pattern table in hardware. To maintain a separate run-time data prediction-pattern table is both expensive and difficult. Instead of using a separate locality prediction table, we can use the cache line structure for both spatial and temporal data caches as shown in Figure 3.2. This requires some additional storage space in the spatial cache. The fields in this cache line are typical for any set-associative cache with the exception of the inclusion of a single bit, L, which is referred to as the "locality" bit and a “reuse” bit, R. The V field is used to indicate cache line invalidation for write-through operations, the LRU bits are included for the implementation of the replacement policy, the tag bits will serve as inputs to the address circuitry to determine matches or hits, the DATA field contains the actual cache data.

Although the line sizes differ in the temporal and spatial data caches, the structure is the same.

V	L	R	LRU	TAG	**LINE OFFSET**	DATA

L = Locality Information, '0' for all cache lines initially, and SET to '1' when data shows locality.

V = Valid Bit.

TAG = Tag bits of the address.

LRU = Least Recently Used bits (Number of Bits depends on the number of sets in the cache).

Figure 3.2: Cache line entries for the spatial and temporal caches

The locality bit is used to indicate that the cache line has an estimated spatial locality behavior while in the spatial cache, or exhibits temporal locality behavior while residing in the temporal data cache when it is set. During the cold start execution phase, data is brought into the spatial cache initially. During cold start, bringing data in the spatial cache is advantageous because we cannot do any prior anticipation of data locality before they are accesses by the program. Thereby, if data is brought into the temporal cache with an anticipation of temporal hits then the cache might face multiples misses if the prediction is wrong. Copying data from the spatial cache to the temporal cache will not increase miss rate and the release of the spatial cache space is possible in case the hit in a particular spatial cache line is found temporal. The strategy for doing this spatial cache to temporal cache transfer is described in the next paragraph.

During a hit in the spatial cache, if the hit occurs due to the same word for which that line

was originally brought from a lower level of memory to the cache, the temporal reuse bit is set to "1". Otherwise, the spatial locality bit is set to "1" to indicate that spatial locality of references is present in the line. The match of the line offset of the new memory reference with line OFFSET field of the spatial cache maintained in cache directory will be used to infer the locality information.

3.2.3 Dynamic locality estimation hardware

The simple hardware scheme for the locality estimation circuit is shown in Figure 3.3. This scheme sets the spatial locality bit and reuse bit following the principles outlined above. To estimate the temporal locality, when a second hit in a line occurs due to temporal reuse of the same memory reference, the circuit checks whether or not the spatial locality bit is set. If it is set, then the access-pattern into that cache line's address space exhibits both types of locality behavior. In this case, maintaining the data residence in the spatial cache is better. Alternatively, if the consecutive access is due to the same memory word reference and the spatial bit is not set, there is a high probability that reference is temporal. In this case, the estimation circuit initiates a move operation of that particular data word into the temporal cache.

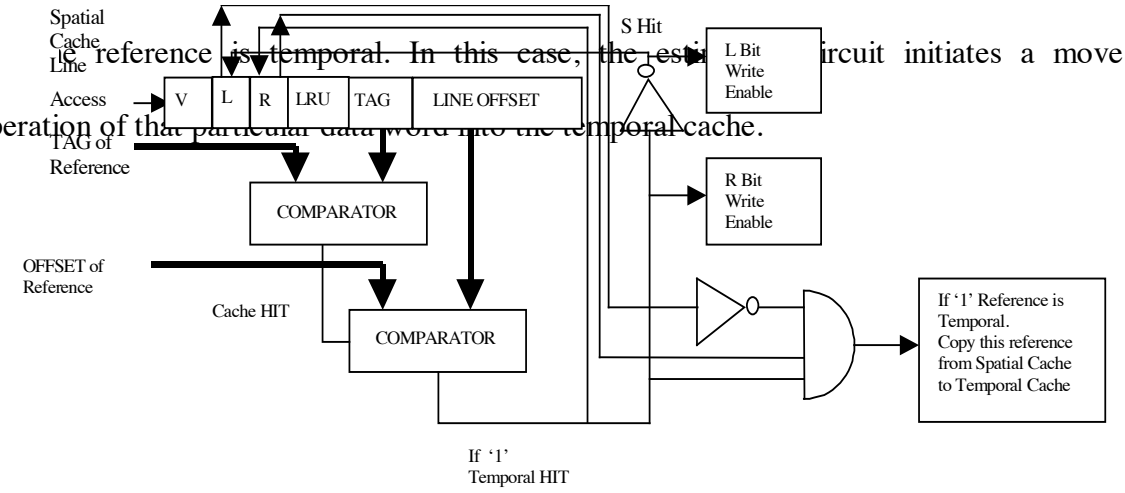


Figure 3.3: Basic Hardware Organization of the Locality Estimation circuitry

This simple runtime heuristic for estimating the locality behavior of memory accesses requires less hardware and avoids complexity in design as compared to other schemes proposed in [36,57], which only detect spatial locality. Simplicity in the hardware of the locality estimation circuit is a crucial design constraint. Simple hardware ensures that overall program access times that are enhanced by the split cache organization are not offset due to excessive latency in the estimation circuit itself. The prediction hardware instructs the cache controller to move data from the spatial cache to temporal cache when a hit is considered due to the temporal locality. Therefore, the cache read-write operation is transparent from the affect of this data movement. The data movement from the spatial to temporal cache occurs simultaneously at the cache speed while the 'hit-data' is transferred to the CPU register. The identification of the temporal hit requires only a comparator and an additional gate through the critical path. The split cache is considered L1 cache, which is fabricated on the same CPU die that offers very fast logic usage. Therefore, minimal latency for the comparator and the gate comprises the prediction latency, and doesn't affect the cache access cycle.

3.4 Experimental Results

A split data cache model that uses the locality prediction circuit defined above was simulated using *C* language constructs in *Unix* Platform. The performance of the locality prediction circuit is compared with the statistical metrics as described in chapter 2 for the SPEC92 benchmark suite.

Table 3.2 shows the comparison of estimated spatial locality by using the split cache as compared to the "true" locality characteristics as predicted by the corresponding statistical analysis [64]. These data are also shown in Figure 3.4 as a plot of the two estimates.

Table 3.2: Comparison of Circuit Estimated to Statistically Analyzed Locality

Benchmark Program	Spatial Locality Percentage estimated by Split Cache (%)	Spatial Locality Percentage estimated by Statistical Analysis (%)	Deviation (%)
nasa7	0.99	0.99	0.00
tomcatv	0.81	0.99	-18.18
espresso	0.76	0.54	40.74
ora	0.75	0.9	-16.67
alvinn	0.87	0.79	10.13
ear	0.89	0.81	9.88
swm256	0.99	0.96	3.13
su2cor	0.71	0.87	-18.39
eqntott	0.73	0.67	8.96
compress	0.95	0.63	50.79
wave5	0.86	0.63	36.51
mdljdp2	0.84	0.64	31.25
sc	0.74	0.55	34.55
li	0.61	0.54	12.96
mdljsp2	0.72	0.49	46.94
doduc	0.60	0.45	33.33
gcc	0.70	0.44	59.09
spice2g6	0.63	0.38	65.79

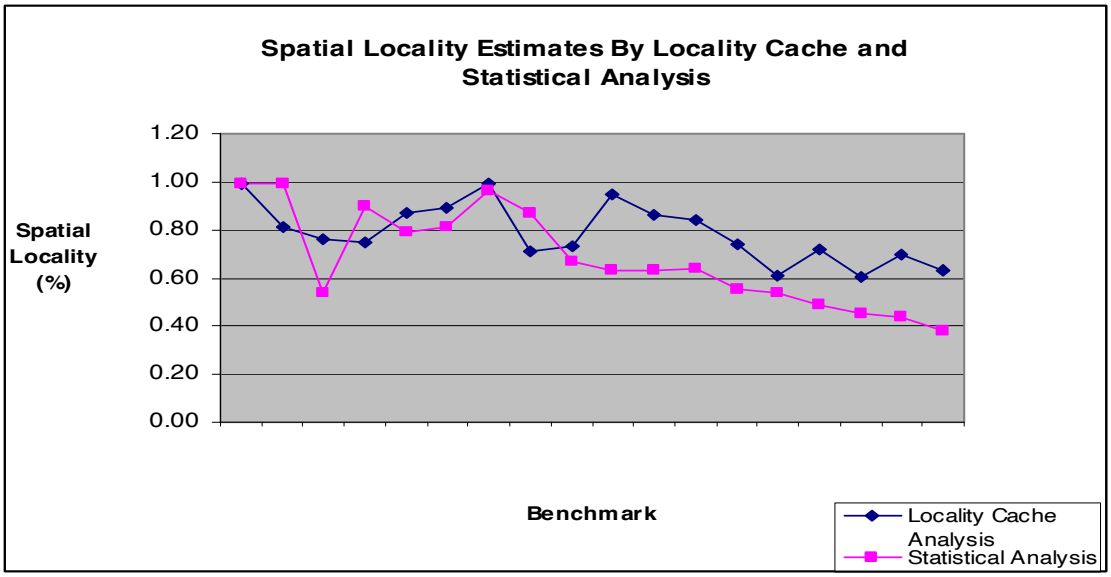
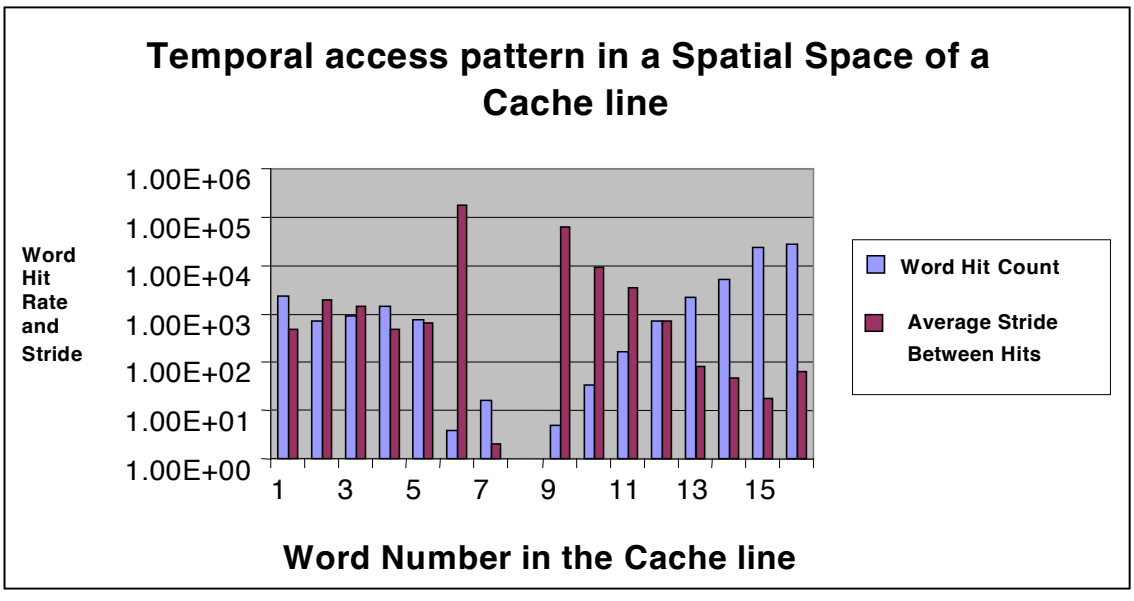


Figure 3.4 Spatial Locality estimates by Dynamic Estimation and Statistical analysis

As is evident from Figure 3.4, the locality estimation circuit usually provides a higher estimation of spatial locality as compared to the statistical analysis. As mentioned previously, all references initially result in spatial cache line fills. Since they are resident in the spatial cache initially, if there is an overlapped spatial and temporal access characteristic, the over estimation of the spatial locality is due to simple nature of the estimation hardware used and the fact that all data are placed into the spatial cache on cold-start initialization. The overlapped spatial and temporal access zone was also



apparent from the statistical analysis as depicted in Figure 3.5.

In some cases, the estimation circuit also under-estimates spatial locality characteristics. This occurs since, in these experiments, the spatial and temporal caches are divided into two equally sized caches. In comparing these results to the unified cache, we only utilize one-half of the capacity for the spatial cache as compared to the unified data cache that contains lines of size greater than one word throughout the entire cache. However, we are still striving to provide better performance even in the case where spatial locality is highly dominant. Since we effectively have a spatial cache with one-half the size of a corresponding unified cache, problems can occur due to "thrashing" where data simultaneously exhibits behavior that is consistent with both temporal and spatial locality. This can easily occur in a case where subsequent loops are present in program that sequentially accesses an array. Within a single loop, the array elements have spatial locality, but among the set of subsequent loops, a single array element may be accessed several times indicating temporal locality characteristics.

To alleviate this problem, the temporal cache was used to store "victim" blocks of data when they are being evicted from the spatial cache due to the replacement policy. Any hit of spatial data that resides in the temporal cache increases the temporal hit count and indicates the presence of more temporal locality in some cases. The justification of this explanation is obvious if we look at the overestimated temporal locality cases for the

benchmarks **tomcatv**, **ora** and **su2cor**. For these cases, the statistical analysis always suggests that the presence of spatial locality is greater than 80%.

3.5 Conclusion

A simple locality prediction circuit is designed and evaluated based on the run-time data access model presented in Chapter 2. The run-time cache resident data analysis indicated the design strategy for this hardware unit. The prediction circuit helps to determine into which cache a specific data block should reside during program run-time. In addition, the prediction circuit incorporates a small amount of additional overhead in terms of hardware complexity and access latency. Due to the simplicity of the hardware, the estimates did not fully agree with the statistical analysis of the locality characteristics as discussed in Chapter 2. The deviation from the statistical analysis is attributed to the cold start strategy, the spatial victim block placement policy and the overlapped temporal and spatial address spaces.

CHAPTER 4

Split-Cache Subsystem Design

The implementation of the locality estimation circuit in a cache organization requires defining the typical data path and the control hardware of the memory management unit (MMU) in the processor architecture. The targeted architecture is an uniprocessor to test the performance of the split data locality cache. In this Chapter, the hardware organization of the split cache subsystem is presented. After describing the hardware and

the associated test bench model, the performance evaluation of this subsystem is presented.

4.0 Split Data Cache Organization

The approach used to design the split data cache was to, a) define the data path, b) define the size and address mapping schemes for the spatial and temporal sub-caches, and c) define the replacement algorithm. Before going through each of these steps, structural placement of this cache in the processor architecture is discussed. Figure 5.1 shows the basic placement of the split data cache in a uniprocessor organization. Here the cache is considered as a level one (L1) cache. The size of the sub-caches is dependent on the optimum cache performance design. To store data into spatial or temporal cache, this organization requires the locality estimation circuit. Data localities are dynamically determined by the estimation circuitry after cold-start of the process.

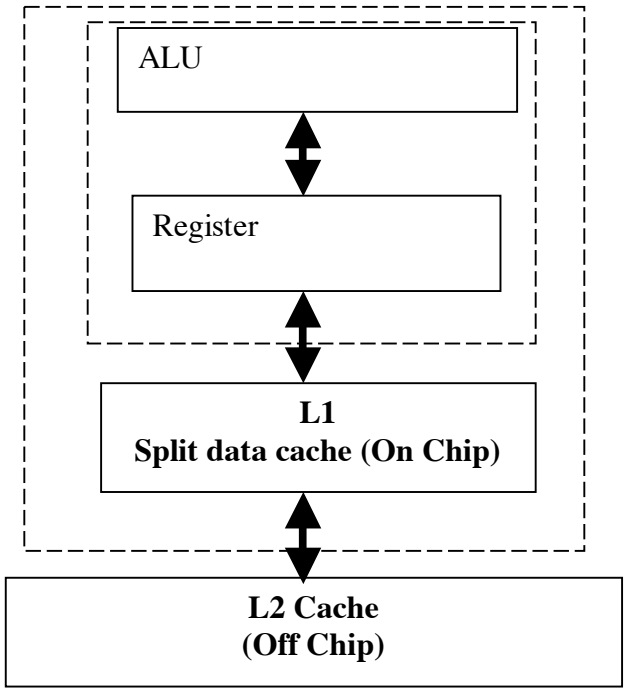


Figure 4.1 The Split Data Cache in a uniprocessor organization

The question now arises, into which cache data should be brought during the cold starts of the process. Incorrect placement of data during cold starts will introduce additional miss penalty. To find a solution, analysis of the cache performance by setting up a split cache simulation scheme for bringing data during cold start was done. The simulation scheme considers all new entries in the cache as having spatial locality initially in one scheme and as having temporal locality in another. During a cache hit, a comparison is performed that determines if hit is due to the same memory reference for which the line was brought into the cache or not. If the line is resident in the "wrong" cache (according to the locality estimate), then that line is copied to the other cache and the current entry is invalidated. The flow diagrams shown in Figure 4.2 and 4.3 depict these two schemes.

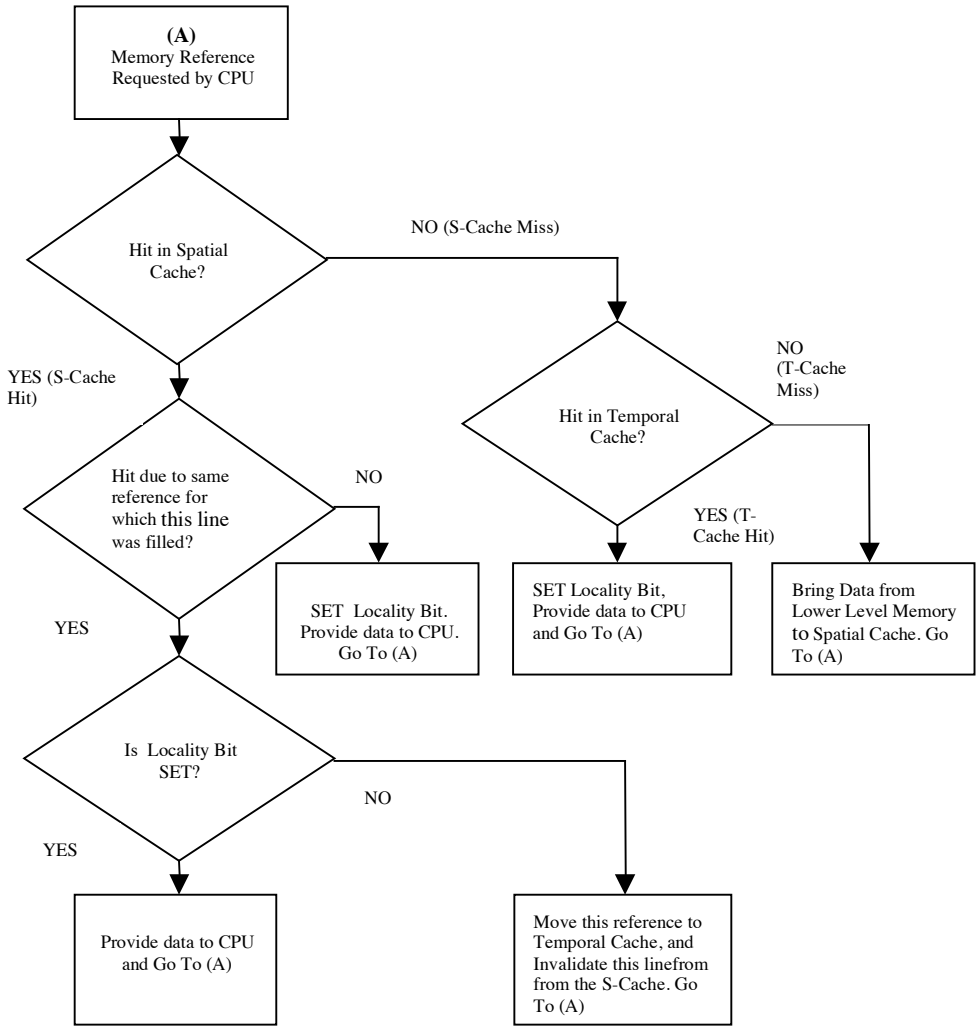


Figure 4.2: Flow diagram of caching scheme where initial references are considered spatial

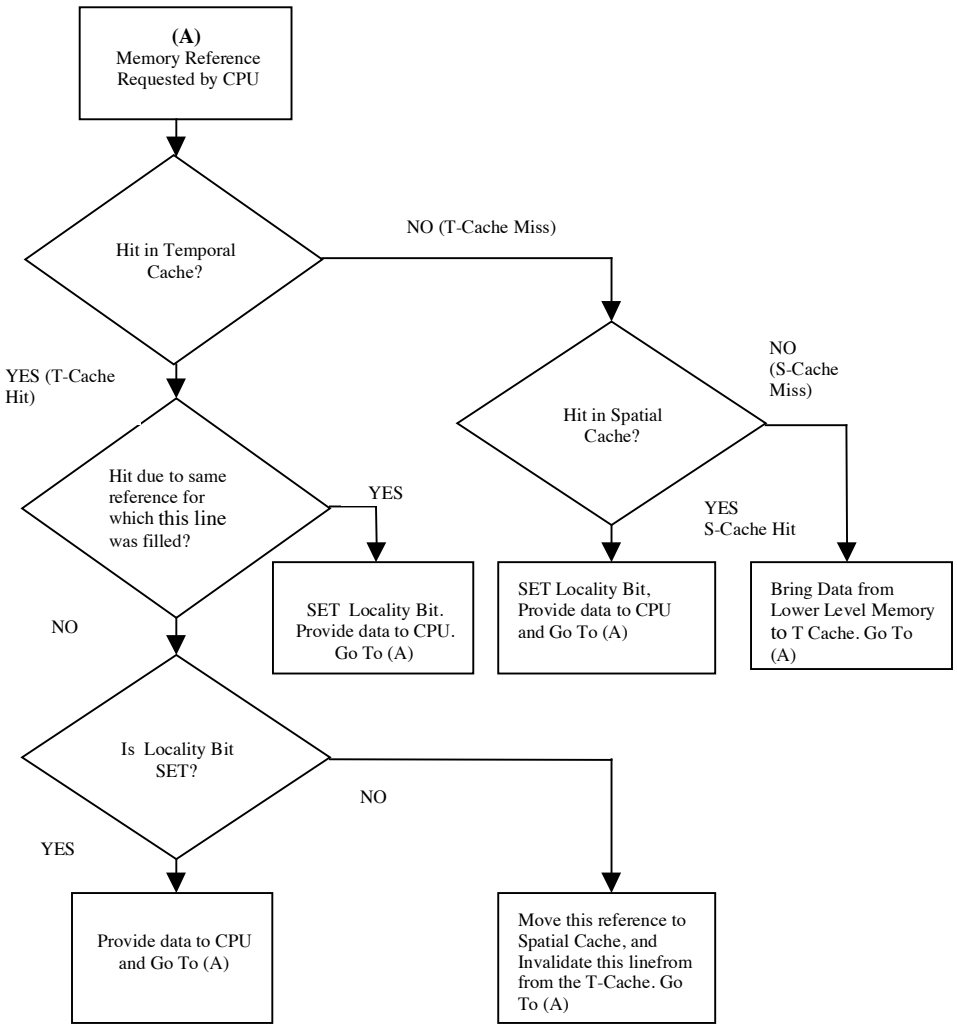


Figure 4.3: Flow diagram of caching scheme where initial references are considered temporal

Two different data locality based cache simulators were created using the ***C*** programming language in ***UNIX***. The simulators predicted the performance of these cache organizations for varying cache line sizes for both temporal and spatial caches. The simulation results reveal how cache performance is affected by using the locality estimation based caching scheme and by varying cache line sizes. In both schemes, a 16 KB 4-way set associative organization initially was used. The 16 KB address space was

further divided into two 8 KB sub-caches (one for temporal and the other for spatial locality).

Using the two 8-KB organizations, a series of simulations for six benchmark programs by varying the cache line sizes in words were conducted. The performance of the caches by keeping the spatial cache line size fixed at a particular word size and varying the line sizes of the temporal cache line sizes by words such as 1, 2, 4, 8, 16 and 32 words were simulated. The simulated performance of the cache for six benchmark programs based on the resulting miss rates was recorded. Next, the cache was simulated by keeping the temporal cache line size fixed and by varying the spatial cache line size.

The simulation results are shown in Table 4-1. As expected, the deciding factor in the success of a locality-based cache depends on the ability to predict the data locality bias of a particular program. The simulation results indicate that varying the size of the spatial cache lines (when all data is initially placed in the temporal cache) does not affect overall hit rates significantly. A more important factor is that the size of the temporal cache line appears to affect the miss rate to a larger extent. This is the case regardless of whether "cold start" data is assumed to exhibit spatial, or temporal, locality. This attribute to the fact that the temporal access behavior of data initially present in a spatial cache helps to reduce the overall miss rate just as it would if it were initially present in the temporal cache.

Table 4-1: Summary of Miss rates of the Locality Estimation based Cache

All data stored in spatial Cache first:							All data stored in temporal Cache first:						
	Spatial Line Sizes:						Spatial Line Sizes:						
Benchmark	4 Byte	8 Byte	16 Byte	32 Byte	64 Byte	128 Byte	4 Byte	8 Byte	16 Byte	32 Byte	64 Byte	128 Byte	Temporal Line Size
LINPACK	21.847	21.931	19.203	17.304	16.510	17.181	21.948	21.948	21.948	21.948	21.948	21.948	4 Byte
	21.863	21.904	19.090	17.181	16.391	16.943	22.052	21.980	21.963	21.963	21.963	21.952	8 Byte
	10.939	10.929	14.533	12.635	11.750	11.371	21.182	21.175	12.180	11.217	10.798	10.558	16 Byte
	7.305	7.142	8.412	7.603	6.749	6.309	16.357	16.119	7.370	6.366	5.892	5.598	32 Byte
	5.728	5.413	5.388	4.573	4.246	3.637	15.287	14.335	5.296	4.368	3.617	3.596	64 Byte
	5.104	4.778	3.962	3.152	2.907	2.358	17.823	13.794	5.415	4.290	3.668	2.782	128 Byte
MATMULT64	3.077	5.367	5.769	6.995	7.096	7.291	4.036	4.036	4.036	4.036	4.036	4.036	4 Byte
	3.107	5.402	5.792	6.965	7.127	7.350	9.099	5.529	5.977	6.409	6.634	6.742	8 Byte
	3.309	4.660	5.723	6.386	10.032	10.618	10.299	7.234	4.958	5.409	5.291	5.341	16 Byte
	3.704	4.949	5.807	9.451	9.973	10.707	10.346	7.629	6.857	6.954	7.239	8.407	32 Byte
	3.912	5.233	5.819	9.967	9.438	10.491	10.971	8.330	7.879	6.796	5.124	7.859	64 Byte
	4.120	5.293	5.723	10.284	10.566	11.119	11.711	9.385	8.873	8.959	8.498	6.894	128 Byte
QSORT	3.599	2.824	2.625	2.084	1.528	1.173	3.847	3.846	3.847	3.848	3.850	3.851	4 Byte
	2.908	2.318	2.297	1.908	1.522	1.152	2.903	2.259	2.291	2.013	1.571	1.152	8 Byte
	2.401	1.912	2.268	2.000	1.682	1.212	2.349	1.919	2.275	2.065	1.716	1.201	16 Byte
	1.900	1.571	1.862	2.099	1.752	1.248	1.888	1.625	1.920	2.143	1.794	1.244	32 Byte
	1.393	1.253	1.503	1.779	1.533	1.099	1.495	1.337	1.599	1.858	1.719	1.149	64 Byte
	0.983	0.949	1.151	1.454	1.270	0.782	1.218	1.095	1.305	1.553	1.516	0.965	128 Byte
WORDFREQ	0.757	1.453	1.639	2.376	3.173	3.639	0.852	0.794	0.789	0.756	0.709	0.682	4 Byte
	0.886	1.278	1.687	2.100	2.869	3.313	6.197	0.947	0.912	0.889	0.903	0.947	8 Byte
	1.561	1.285	1.207	1.836	2.508	3.130	20.956	5.754	1.482	1.285	1.167	0.981	16 Byte
	1.108	1.034	1.049	1.507	2.256	3.003	23.173	7.002	6.355	1.541	1.351	1.169	32 Byte
	1.206	1.121	1.151	1.255	1.899	2.743	18.730	13.109	8.685	6.218	1.320	1.264	64 Byte
	1.639	2.922	2.113	1.038	1.681	1.989	18.136	17.784	15.084	12.977	11.070	2.966	128 Byte
QUEENS	0.008	0.007	0.005	0.003	0.003	0.054	0.008	0.008	0.008	0.008	0.008	0.008	4 Byte
	0.007	0.005	0.004	0.003	0.003	0.054	0.601	0.005	0.005	0.004	0.004	0.003	8 Byte
	0.006	0.005	0.004	0.003	0.003	0.054	6.618	0.912	0.004	0.004	0.003	0.002	16 Byte
	0.005	0.004	0.003	0.003	0.002	0.003	5.228	3.184	0.029	0.003	0.002	0.002	32 Byte
	0.003	0.003	0.003	0.002	0.002	0.002	5.411	0.500	2.846	1.036	0.002	0.002	64 Byte
	0.003	0.002	0.002	0.002	0.002	0.002	0.451	0.539	2.888	1.078	0.044	0.001	128 Byte
CELLAUTO	0.151	0.671	0.361	0.256	2.513	3.038	0.308	0.299	0.298	0.295	0.294	0.294	4 Byte
	0.841	0.658	0.359	0.255	2.512	2.789	7.010	0.634	0.338	0.184	0.109	0.108	8 Byte
	1.729	0.649	0.353	0.254	2.511	2.722	10.467	2.912	0.338	0.186	0.111	0.063	16 Byte
	1.717	0.648	0.353	0.194	2.387	2.463	16.272	12.027	1.519	0.186	0.110	0.102	32 Byte
	1.791	0.648	0.353	0.194	0.111	0.104	10.942	8.630	4.909	0.226	0.109	0.099	64 Byte
	1.905	0.618	0.353	0.185	0.104	0.064	13.693	8.678	4.901	0.243	0.107	0.080	128 Byte

Moving data from larger lines present in the spatial cache to smaller ones in the temporal cache avoids an external memory access, however the converse of this is not true. Due to the variance in locality bias exhibited by the benchmark programs, leads us to believe that the default-starting cache should not be fixed. Rather, these results indicate that the default should be allowed to dynamically change during program execution for all new line fills. Of course, at the beginning of a programs' execution, there must be some initial default cache. Based on the reasoning in the previous paragraph, the initial default cache should be the spatial cache. This will result in wasting memory space due to having a data word with temporal locality consuming a (relatively larger) spatial cache line, but it will avoid the miss penalty due to having a word exhibiting spatial locality present in a temporal cache initially.

In the subsequent sections, the detail design of the split data cache is presented. These include defining the critical data path, defining the size and address-mapping scheme, defining the cache replacement policy. Following this, the implementation strategy of the locality cache and performance modification features such as modified line replacement policy and spatial victim placement policy maintained in the design process are discussed.

4.0.1 Defining the Critical Data Path

The critical data path of the split data cache architecture is shown in Figure 4.4. The critical data path involves the path through which the data needs to travel for a read or write operation at a minimum. The critical path includes the chip data path, the cache

controller, and the tag RAM. For a read or write operation during a search in the cache, the cache controller sends a read/write signal to the cache and the spatial and temporal tags of the memory address is compared simultaneously by the comparators with the stored tags in the tag ram. As this is the usual process of searching in a conventional cache, there is no additional delay for storing or retrieving data for using this organization. The locality estimation circuitry operates independent from data read or writes operations, and therefore does not add to the critical path. As such additional access latency into the cache are not introduced.

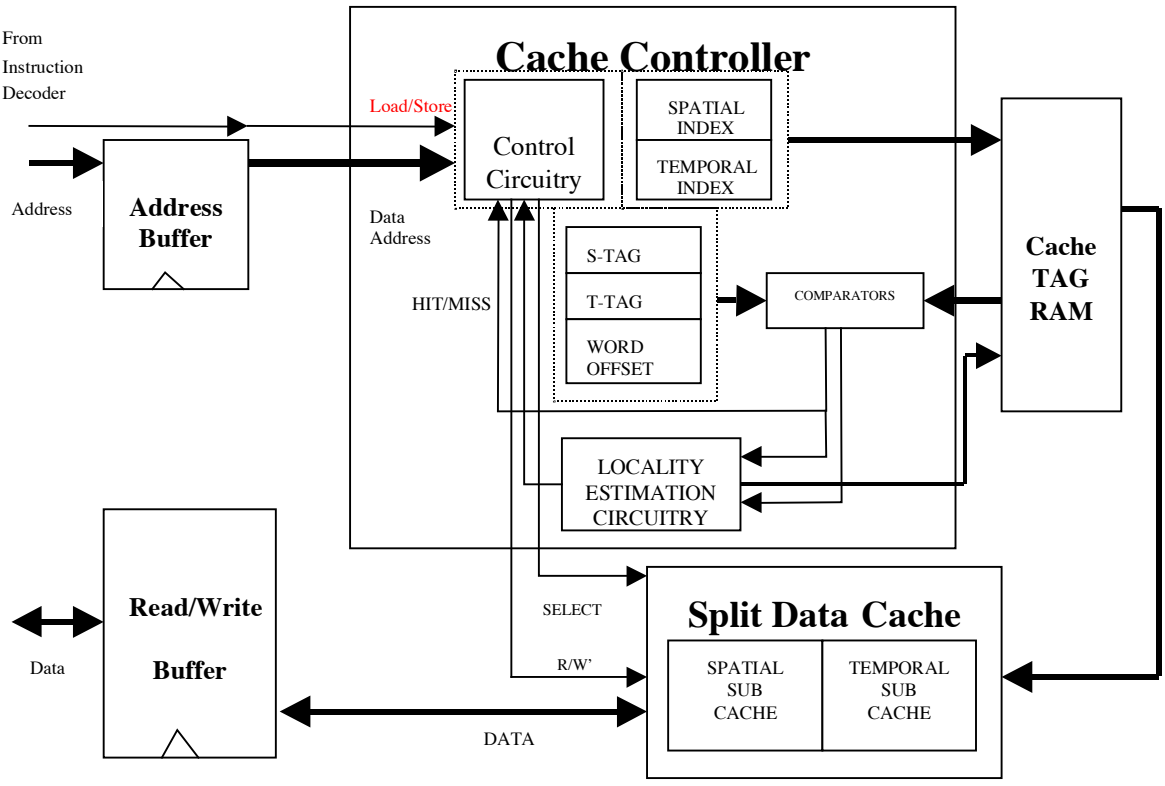


Figure 4.4 Critical Data Path of the Split Data Cache

4.0.2 Defining the size and address mapping schemes

Past research has shown [59, 60, 61] that a data cache which stores a relatively small number of recently accessed or written memory locations can potentially service more than 60% fraction of loads and stores. The goal here is to design a L1 data cache for which a 16KB size is chosen based on the observation that for the benchmark program traces used, 16 KB is enough memory to keep the total miss rate less than 5% including cold start misses in almost all cases and without using any L2 or other assist cache. The SPEC92 benchmark suite can create a substantial amount of bus traffic on the data memory system. Thus, if a small cache can provide good performance for this test suite, then it will perform equally well for many other application programs. Using a small cache size has another advantage; the cache access time for all blocks within the cache remains nearly constant. Using a bigger cache may reduce the miss rate but it will also incorporate unequal access times for different blocks within the cache with the increase of size simply due to the increased distance of data blocks within the cache. Typical observations also show that in most cases, the cache size requirement is very small compared to the cache capacity contained by superscalar processors.

The choice of the address-mapping scheme depends on factors such as, cache lookup speed, hardware complexity and cache performance. The available choices here are direct mapped, set associative and, fully associative. The direct mapped address scheme provides faster lookup time and requires less hardware, however, the cache performance suffers when multiple main memory blocks must map into the same cache blocks. Frequent cache misses and cache updates become a bottleneck in this case. The fully

associative mapping technique suffers minimally from misses and updates in this respect, but it increases the hardware overhead due to usage of comparators equal to the number of lines in the cache. A compromise between these two mapping schemes generally is based on associative mapping. In this case, the number of comparators depends on the degree of associativity in the cache. For example, an 8-way set associative cache requires a total of 8 tag comparators. Figure 4.5 shows the variance of the cache performance [61] on the degree of associativity for typical data cache for SPEC92 benchmarks.

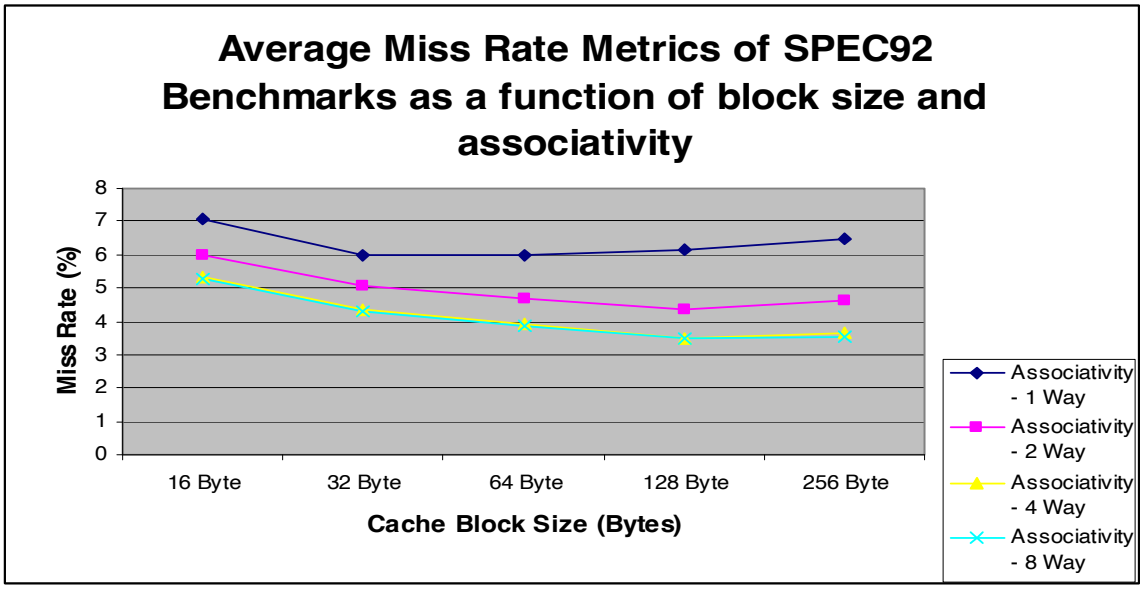


Figure 4.5 The data cache performance metrics as a function of block size and associativity

Figure 4.5 suggests that the impact of increasing associativity on cache performance is minimal after degree 4. The 4-way set associative scheme is also very popular in industry allowing the results of the split data cache to be compared to a large number of systems. For these reasons, 4-way set associative mapping scheme was selected in the target cache organization. From the locality analysis experiments, it is evident that a cache line size

of 128 bytes provides a good window size to determine the locality trend of data accesses over the SPEC92 benchmarks. From Figure 4.5, we observe that a cache line size of 128 bytes also provides good cache performance in terms of minimum miss rate. Considering these two experimental outcomes, the spatial cache line size is chosen to be 128 bytes or 32 words (1 word = 4 bytes). The temporal cache line size is chosen to be 4 bytes or 1 word, as truly temporal data does not require a larger line size to accommodate spatial locality.

4.0.3 Defining the cache replacement policy

When a cache miss occurs after a cold-start, the critical decision becomes which block in the cache should be replaced with the new block from the main memory. Due to the small size of the cache, it is not possible to keep all the working sets of the executing program in the cache. In a direct mapped cache, there is no choice since only one block can be replaced by the new block. However, in set associative or fully associative cache, there are multiple blocks available, which can be replaced with the new block. The placement policy largely depends on the locality property of the reference in the programs. Generally, fixed space replacement algorithms are used for this constrained mapping mechanism. For example, in the set associative cache, the block to be replaced is within a set, thus, the replacement algorithm is invoked for block frames within that set.

Least Recently Used (LRU), First In First Out (FIFO), and Random (RAND) are examples of some common fixed space replacement algorithms. In LRU policy, the block, which was used in least recent time, is the candidate for replacement. In FIFO, the

longest resident is replaced based on first come first out strategy. In RAND, a random block is selected for replacement. The LRU replacement algorithm performs best among these three policies due to the obvious demerits of the other two.

The LRU replacement policy was used in the designed cache organization finding as the best candidate. The LRU policy could be implemented efficiently in the hardware for a small set size and can operate at the cache speed. There are several implementation strategies available to implement the LRU replacement policy in the hardware.

One simple implementation of the LRU policy in hardware uses an aging counter. For a 4 way set associative cache, a counter only requires 2 bits for each line of the set. Therefore, it can count from 0 to 3. Though there are multiple sets in a cache, only one set of LRU bits needs to be updated on a hit. Thus, the maximum number of 2 bit counters required for a 4 way set associative cache is 4. Each time a reference results in a hit and the block frame with count M is referenced, its counter is reset to 0 and all the counters within that set having a value less than j is incremented. The other counters are unmodified. If the reference results in a miss and the set is full, the block with counter value of j = 3 is overwritten with the new block and its counter is reset to 0. The counters of the other three blocks are incremented by 1. The block with the counter value of 3 can be obtained by an associative search of the counters. LRU bits update process on line hit is shown in Figure 4.6.

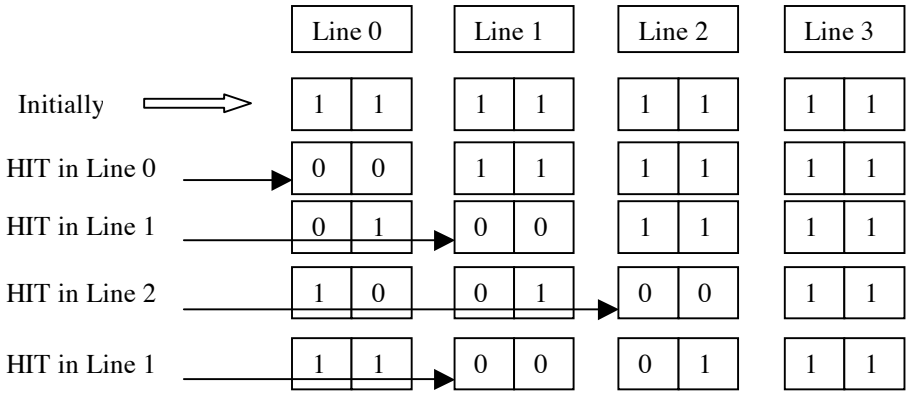


Figure 4.6 LRU Bits update process on line Hit

4.1 Implementation of the locality cache

Determining the line size of the spatial sub-cache has already been discussed. For the temporal sub-cache, a line size of 1 word (4 bytes) was chosen. For the data locality estimation, we need to store the run-time data locality history. For this purpose 3 additional bits in the spatial cache line are used to estimate the data locality type when a cache hit is encountered during the execution of a program. Figure 4.7 shows the organization of the spatial cache line. This organization was chosen to avoid keeping a separate locality prediction table in the hardware and to maintain a simple strategy for estimating the data locality type. The temporal cache line does not require any additional bits like S, T or Line Offset and uses an L bit per line to indicate that an entry in a particular line is temporal.

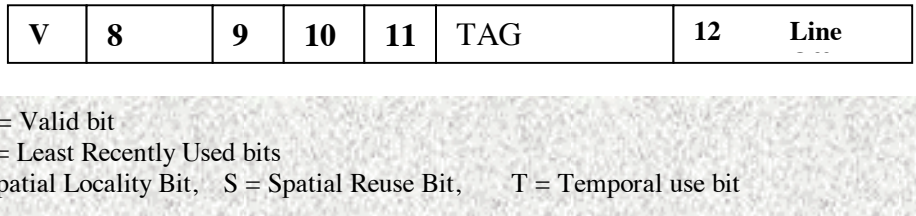


Figure 4.7. The spatial cache line organization

Both the spatial and temporal Caches use a 4-way set associative address-mapping scheme. The line size of spatial cache is 32 words and the temporal cache line size is 1 word. The line size of spatial cache was chosen to be 32 Words to provide the best window size for the data locality analysis based on results obtained from the heuristics for different line sizes that were analyzed in [63,64].

4.1.1 Locality estimation protocol

A simple locality estimation protocol was designed using 3 bits L, S, and T. During a line fill, the spatial Cache line stores the line offset to indicate the particular word for which that line was brought to cache. On successive hits into a line, the line offset is always compared with the initial line offset. The estimation circuit sets the L (locality bit), S (spatial reuse bit), and T (temporal use bit) bits according to the state transition diagram shown in Figure 4.8.

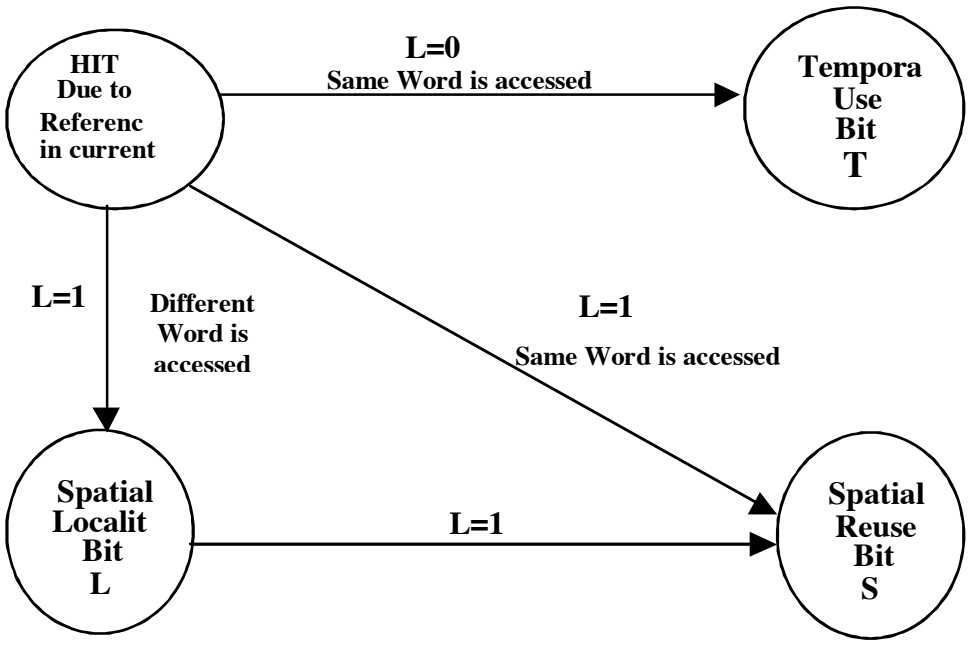


Figure 4.8 State transition diagram for the cache status

The decision diagram to set the L, S and T bits is shown in Figure 4.9. The temporal use bit sets to '1' if the hit is due to same word reference for which the line was brought into cache initially and at the same time the spatial locality bit is not Set. The spatial Locality bit sets to '1' if the hit is not due to same reference. The spatial Reuse bit sets to '1' if the L bit is set and Hit is due to same reference. On a second temporal hit, when the L bit is not Set then the probability of that reference being temporal is quite high, so, the cache organization copies that particular reference to the temporal Cache. In addition, this time the cache circuitry invalidates that entry from the spatial Cache to avoid cache-coherence problem.

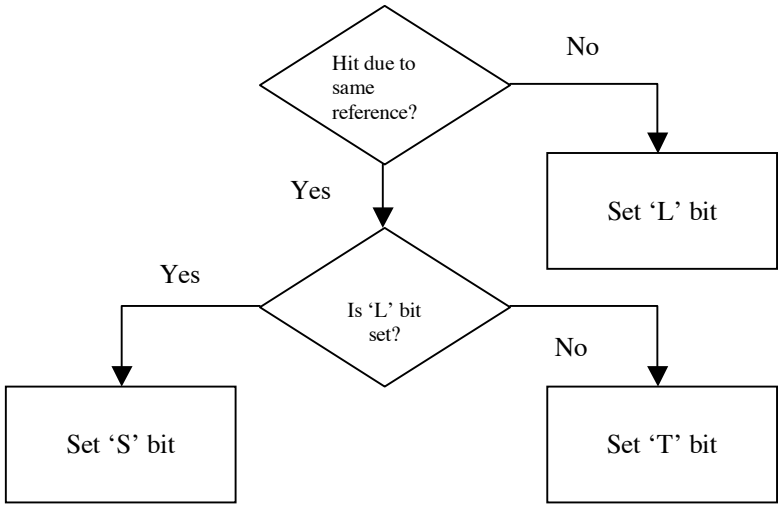


Figure 4.9 The decision diagram for setting L, S and T bits

The basic circuitry for the locality estimation protocol is shown in Figure 4.10. For simplified implementation of the Split-Data Cache the L bit from the temporal cache, and the S bit from the spatial cache could be avoided with the sacrifice of minor cache performance. The additional storage space required for this cache for the spatial sub-cache is 64 Bytes for 8KB-cache capacity, which is quite low in comparison to 8.75 KB required to maintain a spatial Footprint Table suggested in [57].

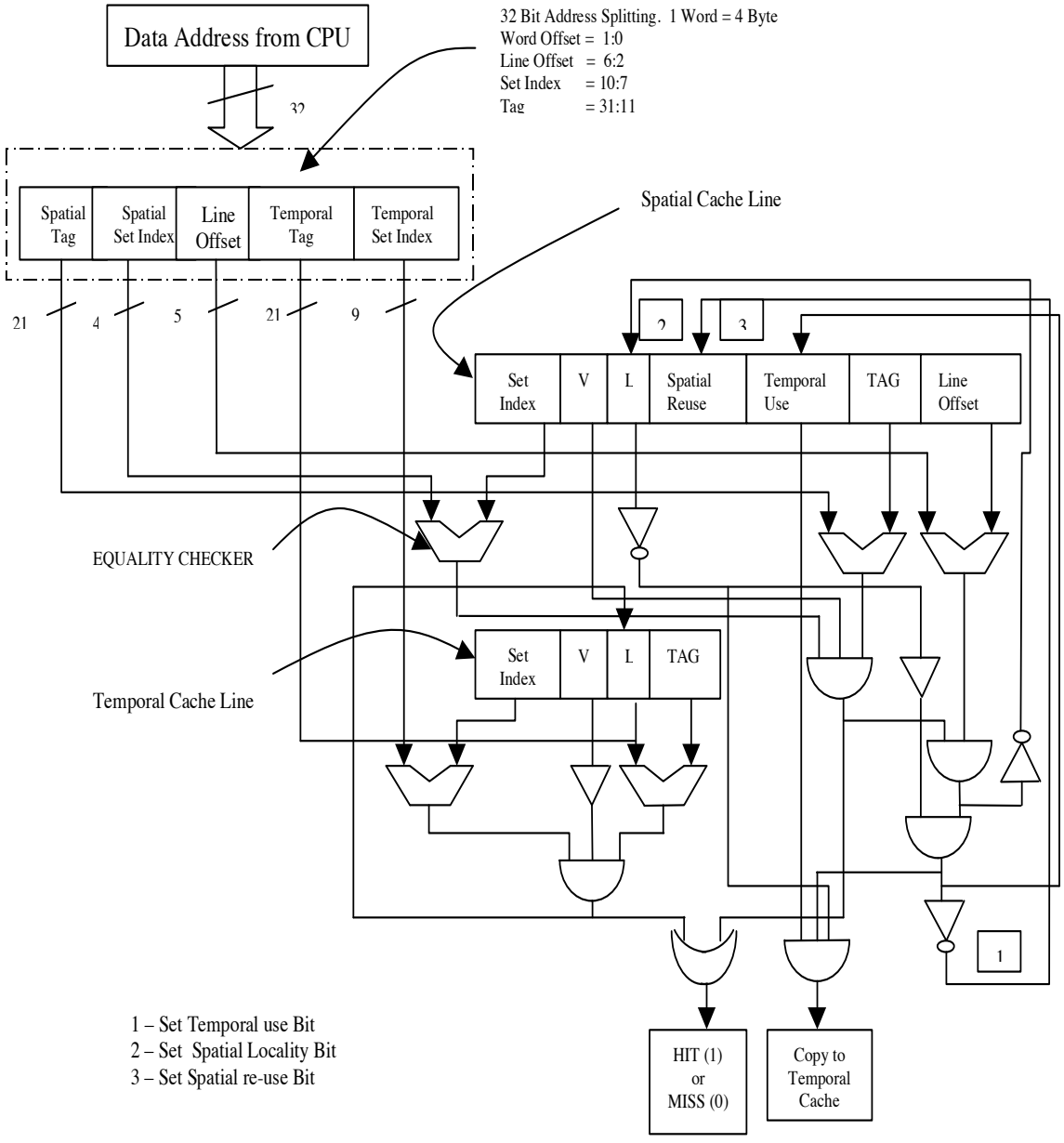


Figure 4.10 Locality estimation circuitry of the Split Data Cache

4.2 Modified line replacement policy of the split data cache

The locality based split data cache is further benefited from the use of the spatial reuse bit 'S'. The reuse bit 'S' will be set when the spatial line is used more than once. The probability of setting the 'S' bits for the highly reused spatial lines are quite high. In this case, if we can increase the residency period of these lines in the cache, then the cache performance will increase further. With this goal, a modified replacement algorithm as depicted in Figure 4.11 has been suggested. This algorithm will replace the lines with the 'S' bit set with a new line only when it is unavoidable. The performance enhancement of using this scheme can be readily compared with the usual LRU replacement policy. If the split data cache uses a normal LRU replacement policy, the use of the 'S' bit is redundant and can be avoided to save storage space.

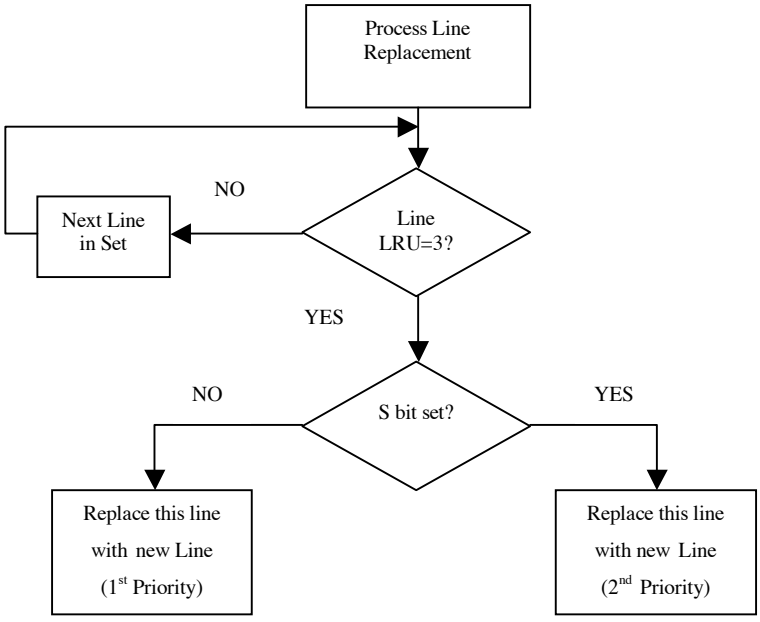


Figure 4.11 Modified line replacement policy

In the temporal cache, this modified line replacement policy can be used for keeping the temporal locality data longer in the cache using the temporal locality bit. In a normal LRU replacement policy, this temporal locality bit is also redundant in the temporal cache and thus can be avoided.

4.3 Handling spatial Victim Blocks

One of the major performance bottlenecks arises due to the reduction of cache capacity due to the division of the cache space into two separate sub-caches. The spatial sub-cache suffers more from this problem. A 16 KB 4-way set associative cache can place a total of 128 (4*32) lines memory blocks of size 128 bytes. For the split cache for equal spatial capacity to the temporal, the cache capacity for spatial sub-cache is 8KB with 4-way associativity. This implies that it can place a total of 64 (4*16) lines memory blocks of size 128 bytes. So, when mapping into the split cache more frequently the memory blocks from the cache will be evicted with compared to the 16KB unified cache. This increased number of spatial victim blocks must be properly handled to maintain the cache performance. With this in mind, a simple victim placement policy was used to reduce the conflict misses that occur more frequently in the split data cache. Usually the initial placement of a memory block is always in the spatial cache since we do not initially know what the access locality would be for a newly accessed memory block. It is essentially increasing the traffic into the spatial cache and results in creating more

victims. Thus, whenever a block is evicted from the spatial cache, instead of removing the block, the block is placed into the temporal cache. This process utilizes the temporal cache bandwidth and space properly and helps to eliminate the problem, which arises due to the reduction of the spatial cache capacity. This transfer of the victim blocks can be done quite easily within the hardware making a parallel transfer of the spatial cache block into temporal size multiple blocks. Since the process is internal to the cache, it operates at the speed of the cache and will not harm the normal read/write operation of the cache. For faster data transfer, the temporal cache is capable of writing 32 Words of the spatial line at the same time into itself if required. Figure 4.12 illustrates spatial to temporal transfer process.

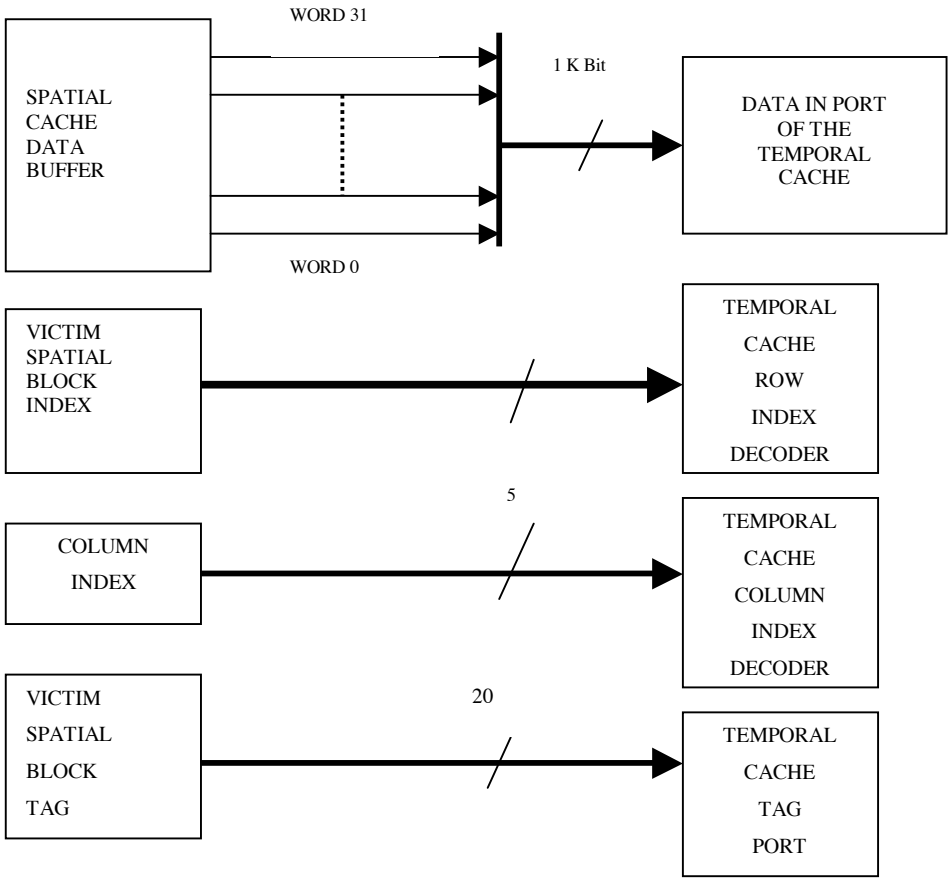


Figure 4.12 Spatial to temporal Cache Data Transfer

4.4 Summary

The basic implementation strategy of the split data cache emphasizing performance modification techniques was presented. The cache performance metrics for different spatial and temporal cache line sizes and placing the data as predicted by the locality prediction circuit for several benchmark programs were analyzed and characterized by simulating the locality cache organization. This performance metric led us to determine which cache the data should initially be stored in during the "cold-start" phase of program execution. Based on the simulation results, it was determined that the organization should initially store data in the spatial cache to avoid additional memory accesses that would occur if spatial data were initially incorrectly placed in the temporal cache. The split cache subsystem design process also incorporates some further performance modification issues by using modified line replacement policy in the spatial cache and the placement of the spatial victim blocks in the temporal cache.

Performance evaluation of the split data cache is presented in the next chapter. The performance metrics presents a detailed step by step evaluation of the different factors that contributes in the increase of the cache performance.

CHAPTER 5

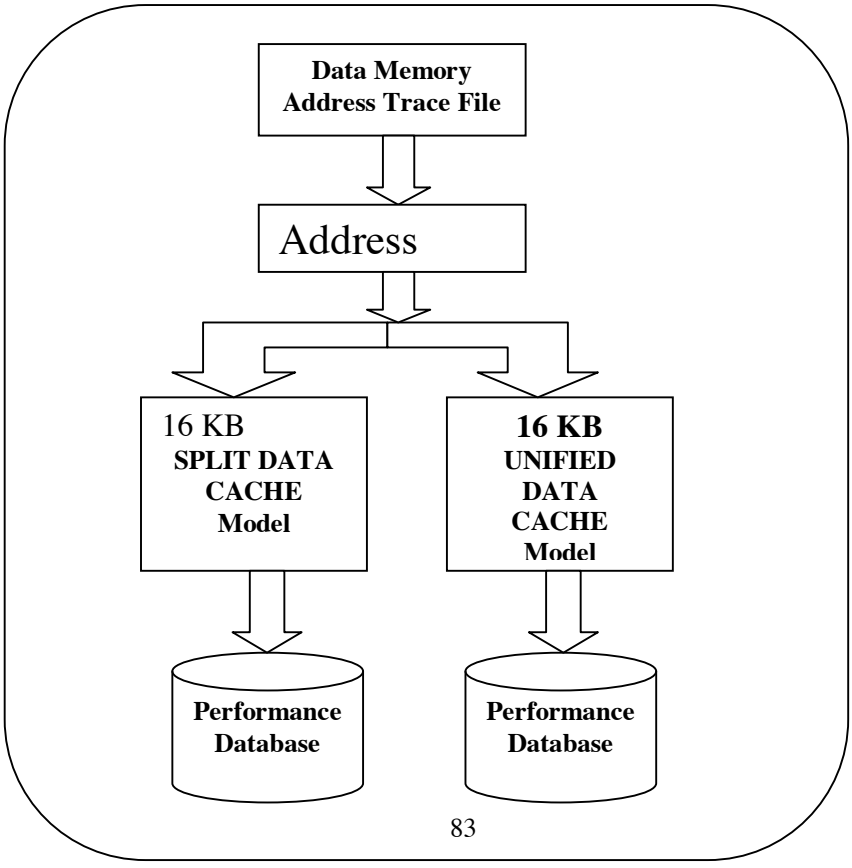
Performance metrics of the split data cache

The performance metrics of the split data cache along with the impact of the implementation strategy is presented in this chapter. The performance metrics show a step-by-step evaluation of the impact of using the locality estimation circuit, the impact of modified replacement policy over conventional LRU policy, and the performance impact of placing the spatial victim blocks into the temporal cache. Finally, this chapter concludes with a generalized evaluation of the split data cache.

5.0 Experimental setup

The split data cache organization was modeled using *C* language and compiled under ***UNIX*** using ***cc*** compiler in a **Sun** Workstation. The program simulates the cache and

accumulates runtime profiles of memory accesses into the temporal and spatial sub-caches. The spatial sub-cache is organized to store 128 bytes (32 Word) of data per line using 4-way set associative address scheme and a capacity of 8KB. The spatial line size is kept at 128 bytes, identical to that used in [7] for locality analysis. The temporal cache line is 4 bytes (1 Word). This was chosen after evaluating several other line sizes. The temporal sub-cache data storage capacity is 8 KB and the total data storage capacity of the spatial sub-cache is 8 KB. The split data cache organization follows the same caching strategy described in chapter 4 to place data into appropriate sub-caches. Data memory address (load and store) traces for different SPEC benchmarks were used to evaluate this cache performance. The performance of the split data cache was compared with the performance of the unified data cache with a storage capacity of 16 KB. Figure 5.1 shows the experimental setup.



Runtime profiles of spatial and temporal hit rates were mainly maintained with data traffic from the memory. After final execution of each benchmark program, measures of total memory read, write, miss, hit, total bus traffic and percentage of spatial reuse were recorded. A demand fetching policy was used that fetches data only during a cache miss. For writing, write allocation on miss, and, write updates on write-buffer policies were used. For a write, no write allocation on miss and write update on write buffer may be followed to increase the cache performance further. This has an advantage over using the write allocation on write miss. The advantage comes, due to that fact that, on write misses, if the cache is not updated immediately, the cache performance will not be affected. In most of the write cases, data is written once in the memory. Therefore, if the same location is not accessed again, which is true in many cases, then the cache will suffer less from write misses. If the written data is required to be read again, then it will be cached by a read-miss. The experimental setup discussed here uses the write allocation policy to observe the cache performance even when the cache is suffering extra misses for the absence of memory write locations.

The performance of a 16-KB 4-way set associative unified data cache was also investigated using a line size of 128 bytes. The performance metrics of the Split and Unified data caches were stored for various benchmarks in separate databases for further analysis and comparison. Table 5.1 shows the description of the SPEC92 benchmarks

used in this experiment. The SPEC92 benchmark suite consists of public domain, non-trivial programs that are widely used to measure the performance of computer systems in a Unix like operating system environment [61]. These benchmarks were expressly chosen to represent real-world applications and were intended to be large enough to stress the computational and memory system resources of current generation computers.

Table 5.1 Description of the SPEC benchmark programs used in the cache test bench

Benchmark Program	Language	Description
Alvinn	C	Robotics neural network training
Compress	C	Reduces file size by Adaptive Lempel-Ziv compression
Doduc	Fortran	Thermohydrolic simulation of a neural network
Ear	C	Human ear simulation
Eqntott	C	Builds truth table from a Boolean expression
Espresso	C	Boolean function minimization
Gcc	C	GNU C compiler
Mdljdp2	Fortran	Molecular dynamics (double precision)
Mdljsp2	Fortran	Molecular dynamics (single precision)
Nasa7	Fortran	Seven floating-point synthetic kernels
Ora	Fortran	Ray tracing
Sc	C	Spreadsheet calculator
Su2cor	Fortran	Quantum physics mass computation
Swm256	Fortran	Shallow water equation solver
Wave5	Fortran	Maxwell's equation solver
Li	C	LISP interpreter solving the nine queens problem
Tomcatv	Fortran	Mesh generation program

In the subsequent sections of this Chapter, the performance evaluation of the split data cache is presented. The effect of the modified replacement policy used for the cache, the effect of using the temporal sub-cache as a victim cache for storing the spatial victim blocks, and finally the hardware requirement and relative size comparison of the unified and split data cache models used are all discussed.

5.1 Split Data Cache Performance

Table 5.2 and Figures 5.2 and 5.3 show the tabulation and graph of the split and unified cache miss rates and bus traffic for different SPEC92 benchmark programs.

Table 5.2 Miss rate and bus traffic of the SPEC benchmarks using split and unified data caches

Benchmark	Miss Rate (%) Unified Data Cache	Miss Rate (%) Split Data Cache	Miss Rate Reduced (%)	Bus Traffic (Bytes) Unified Data Cache	Bus Traffic (Bytes) Split Data Cache	Bus Traffic Reduced (%)
espresso	0.30	0.21	30.00	72,960	51,440	29.50
spice2g6	0.94	0.48	48.94	259,840	129,860	50.02
doduc	0.93	0.55	40.86	291,968	170,008	41.77
li	0.65	0.54	16.92	212,992	169,920	20.22
eqntott	0.59	0.52	11.86	173,184	148,824	14.07
compress	4.39	4.36	0.68	1,564,928	1,552,468	0.80
mdljdp2	1.04	0.68	34.62	307,584	195,100	36.57
wave5	0.16	0.14	12.50	34,944	31,872	8.79
tomcatv	1.35	1.26	6.67	666,368	618,316	7.21
ora	0.07	0.07	0.00	18,816	17,920	4.76
alvinn	0.14	0.13	7.14	32,512	31,888	1.92
ear	0.45	0.43	4.44	130,816	124,748	4.64
sc	0.38	0.24	36.84	95,232	57,468	39.65
mdljsp2	0.54	0.22	59.26	128,640	51,452	60.00
swm256	0.14	0.14	0.00	36,992	36,992	0.00
gcc	1.68	0.85	49.40	475,264	232,416	51.10
nasa7	0.72	0.71	1.39	182,784	179,932	1.56
fpppp	1.02	0.62	39.22	391,552	230,112	41.23

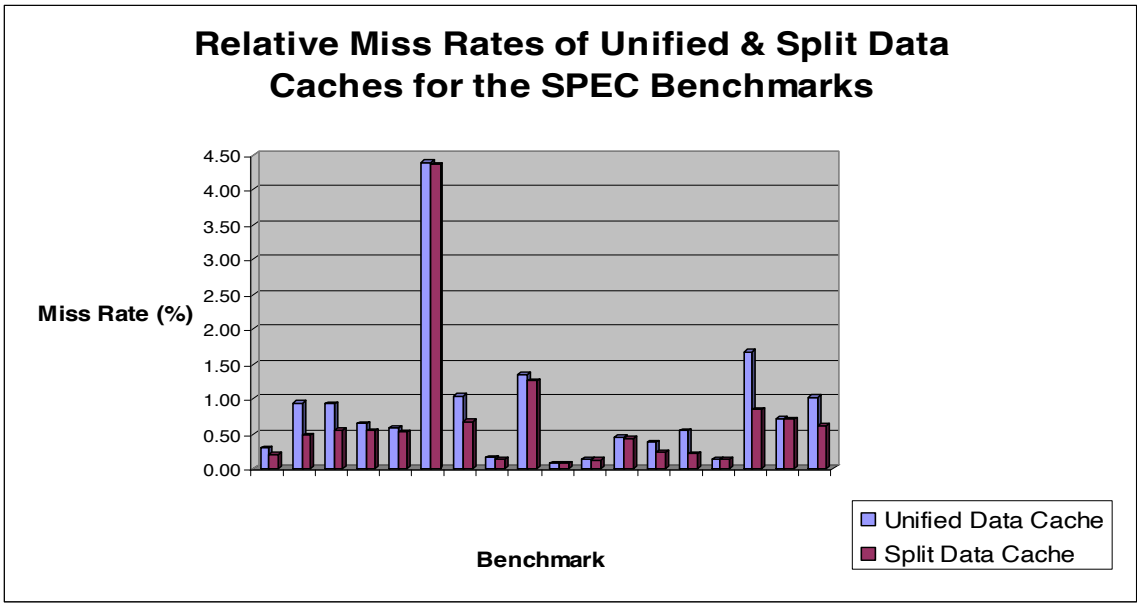


Figure 5.2 Relative cache miss rates of split and unified data caches

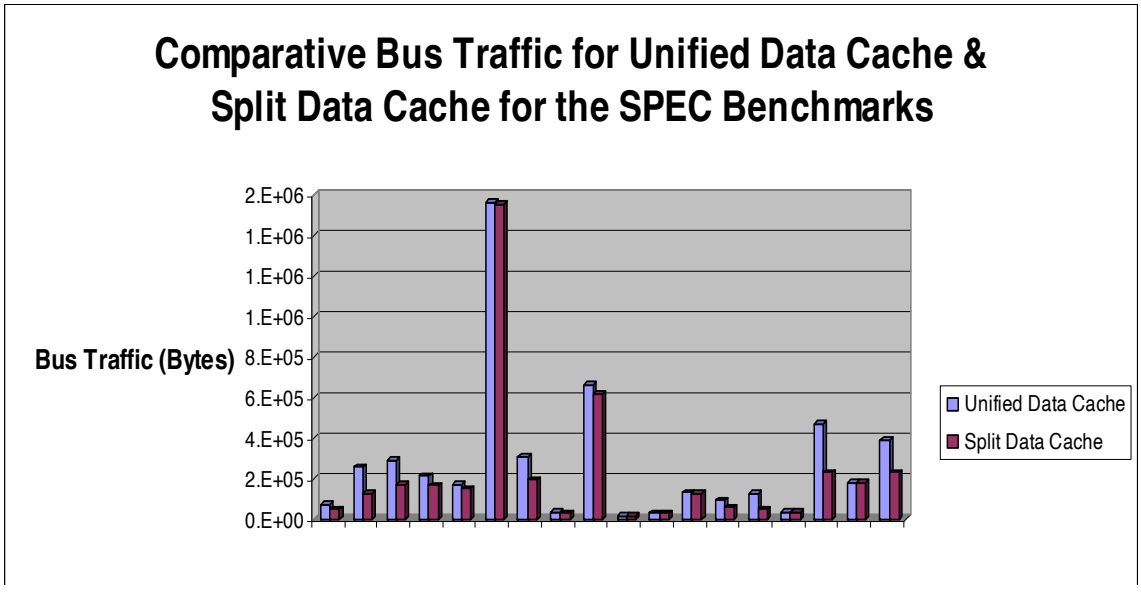


Figure 5.3 Comparative bus traffic for using split and unified data caches

The performance metrics presented in Table 5.2 indicates the miss rate is reduced up to 59% and bus traffic is be reduced up to 60% in the best cases. In most of the cases, the performance increase of the split data cache is significant when compared to a traditional unified data cache. It is also noted that in some cases, such as for benchmark **ora,** the bus traffic is reduced by 5% although the cache miss rate remains the same as that for a unified data cache.

It is interesting to note by observing Table 5.2 that the performance increase for the benchmark programs whose temporal locality (by statistical analysis) is higher than that for the spatial locality cases (i.e. **spice2g, doduc, mdljdp2, mdljsp2, gcc**). This implies that the locality estimation circuit tends to predict temporal locality more accurately than spatial locality.

The split data cache has to perform better even in the case where spatial locality is highly dominant. Due to the nature of splitting the capacity smaller than the original size in the unified cache, the split cache will suffer from a thrashing problem in the presence of a majority spatial locality data more than the unified data cache. To solve this problem, the temporal cache was used to store the victim blocks of data when they are being evicted

from the spatial cache due to the thrashing problem. In addition, a modified replacement policy based on the spatial reuse frequency of a line in the spatial cache was used. The justification of using this scheme is to give more residency time to the spatial blocks in the spatial cache whose reuse frequency is higher. The effect of this modified replacement policy and spatial victim replacement policy is presented in the next section.

5.2 Affect of the Modified Line Replacement Policy in the spatial sub-cache

The split data cache uses a spatial reuse bit to mark lines in the spatial cache if that particular line is used more than one time spatially. Typically in normal address patterns, the probability of future reuse of cache lines is quite high. In this situation if a line is detected for repeated spatial use, keeping that line longer in the cache increases the hit rate of the cache. To facilitate this hit increase process, the modified replacement policy replaces spatial lines whose spatial reuse bit is not set first. If no lines are available to continue this operation, only then will the cache replace a spatially reused line. The general LRU replacement policy is also followed to select lines with least recently used signatures. The impact of this modified replacement policy is shown in Figure 5.4, which uses the data of Table 5.3.

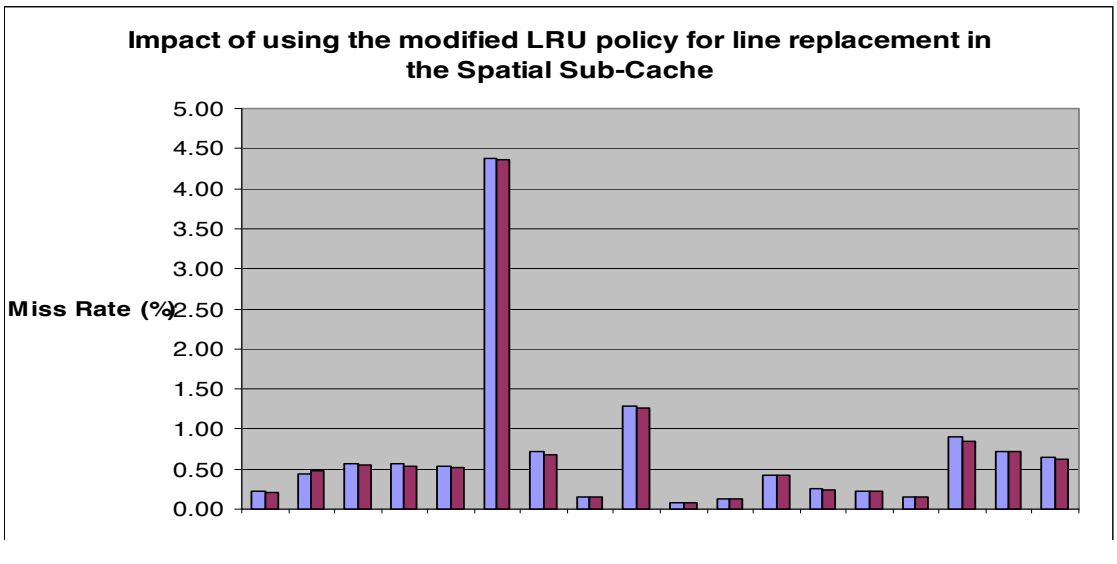


Figure 5.4 The impact on the miss rate due to the modified line replacement policy used in the spatial sub-cache

Table 5.3 The contribution of the modified LRU policy on reducing miss rate

Benchmark	Miss Rate (%) using General LRU Policy	Miss Rate (%) using Modified LRU policy	Contribution of the modified LRU policy (%)
espresso	0.22	0.21	4.55
spice2g6	0.44	0.48	-9.09
doduc	0.56	0.55	1.79
li	0.57	0.54	5.26
eqntott	0.53	0.52	1.89
compress	4.38	4.36	0.46
mdljdp2	0.71	0.68	4.23
wave5	0.15	0.14	6.67
tomcatv	1.28	1.26	1.56
ora	0.07	0.07	0.00
alvinn	0.13	0.13	0.00
ear	0.43	0.43	0.00
sc	0.25	0.24	4.00
mdljsp2	0.22	0.22	0.00
swm256	0.14	0.14	0.00
gcc	0.89	0.85	4.49
nasa7	0.72	0.71	1.39
fpppp	0.65	0.62	4.62

The observed contribution of the modified replacement policy is not highly significant and is only about 7% of the reduced miss rate at maximum. Therefore, the modified line

replacement policy could be eliminated in the actual implementation of the Split Data cache without sacrificing any significant performance factor. This will also reduce hardware requirement costs.

5.3 Affect of the spatial Victim Placement Policy

In the split cache design, separate victim cache was not included to place the spatial victim blocks. The number of victim blocks in the split data cache is expected to be higher than the unified data cache due to the reduced data storage space of the spatial cache than the unified data cache. Main goal is to keep the split data cache performance similar to the unified data cache even in the case when the data locality bias shows strong spatial dominance. In this case, some sort of victim cache arrangement to avoid the increased victim traffic of the spatial sub-cache should be included. To solve this problem, the designed split cache uses the temporal cache to place the victim spatial blocks only when their spatial reuse bit is set. The performance variation of using this victim placement policy is tabulated in Table 5.4 and the corresponding comparison bar graph is shown in Figure 5.5.

Table 5.4 The contribution of the victim placement policy on reducing miss rate

Benchmark	Miss Rate (%) without using Victim placement Policy	Miss Rate (%) using Victim placement policy	Contribution of the Victim placement policy (%)
espresso	0.21	0.21	0.00
spice2g6	1.35	0.48	64.44
doduc	1.45	0.55	62.07
li	0.75	0.54	28.00
eqntott	0.79	0.52	34.18

compress	4.68	4.36	6.84
mdljdp2	1.05	0.68	35.24
wave5	0.17	0.14	17.65
tomcatv	1.61	1.26	21.74
ora	0.09	0.07	22.22
alvinn	0.14	0.13	7.14
ear	0.46	0.43	6.52
sc	0.39	0.24	38.46
mdljsp2	0.53	0.22	58.49
swm256	0.30	0.14	53.33
gcc	1.38	0.85	38.41
nasa7	0.73	0.71	2.74
fpppp	0.79	0.62	21.52

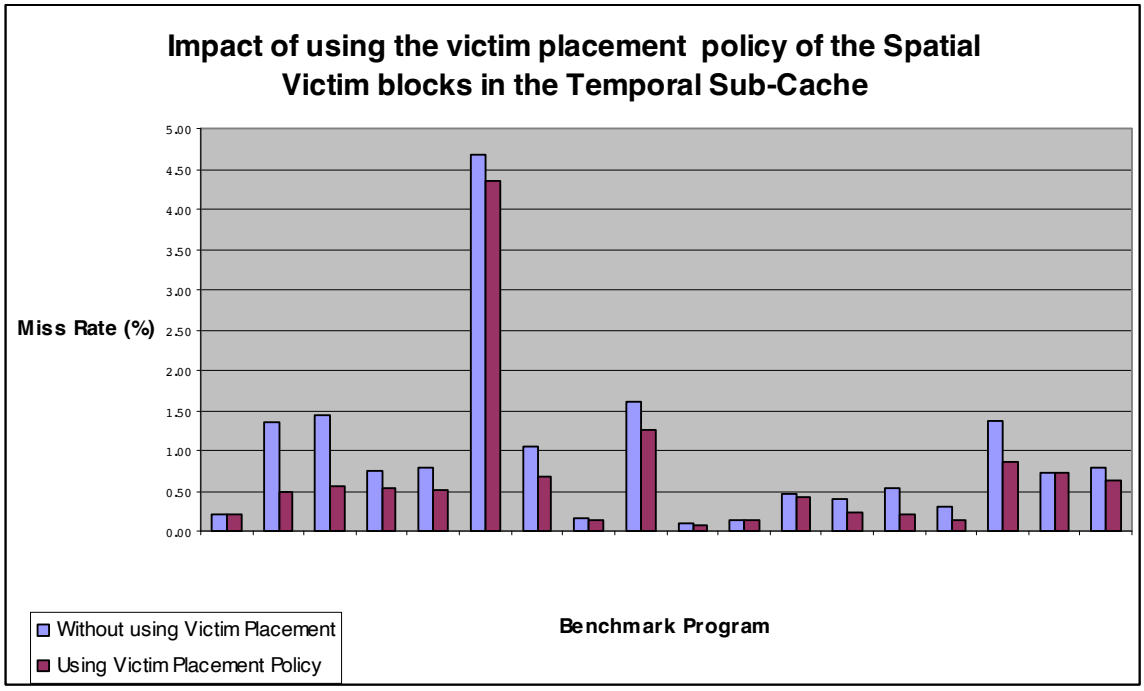
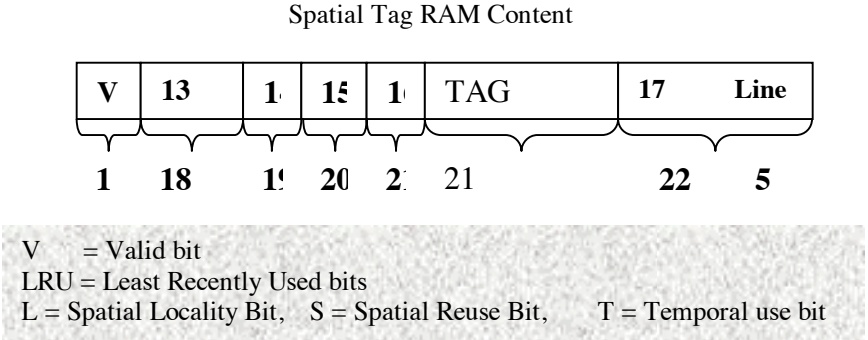


Figure 5.5 The impact on the miss rate due to the victim placement policy used for the spatial victim blocks

The performance metrics presented in Table 5.4 clearly indicates the importance of the spatial victim placement policy. If spatial locality is highly dominant, then without using the victim placement policy the cache performance can degrade more than 60%. This issue clearly challenges the advantage of using the split data cache without using a victim cache. Usual caches in modern computers use a victim-cache to gain a performance boost. In the current split cache design, no overhead of using a separate victim cache is preferred due to the fact that, the goal should be to use the existing cache space more fruitfully. The split cache organization does not posses the need for using any additional victim cache as a contrast to unified caches.

5.4 Hardware cost and area analysis

One of the main objectives in designing the split data cache was to keep the hardware design and cost simple and minimum. In this respect, the question arises of how much space should be allocated for the spatial and the temporal sub-caches. Obviously, we want to obtain a cache organization which will not suffer from the majority data access bias whether it is spatial or temporal. In this respect, splitting the spatial and temporal sub-cache size as two equal parts is advantageous in one sense; it provides an equal space for both types of locality, and the tag sizes in two sub caches remains the same. In the experimental setup, 8KB spatial and 8KB temporal sub-caches have been used. Figure 5.6 shows the structure of the tag-rams for this organization.



The number of sets for the 8KB spatial and 8KB temporal caches comes from the following calculation.

Spatial sub-cache:

Cache size = 8 KB = 2^{13} bytes

Associativity = 4 = 2^2

Block size = 128 bytes = 2^7 bytes

Therefore, the number of lines per set

$$= \frac{2^{13}}{2^2 \text{ X } 2^7} = 2^4 = 16$$

Total lines = 16 * 4 = 64

For 32 bit data address the splitting of the address is shown in Figure 5.7.

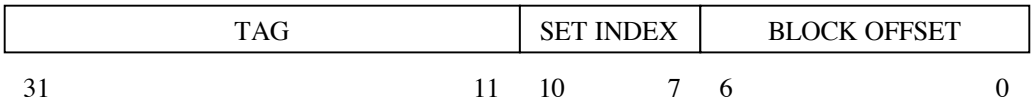


Figure 5.7 32 bit address splitting of the spatial sub-cache

Temporal sub-cache:

Cache size = 8 KB = 2^{13} bytes

Associativity = 4 = 2^2

Block size = 4 bytes = 2^2 bytes

Therefore, the number of lines per set

$$= \frac{2^{13}}{2^2 \text{ X } 2^2} = 2^9 = 512$$

Total lines = 512 * 4 = 2048

For 32-bit data address, the splitting of the address for the temporal cache is shown in Figure 5.8.

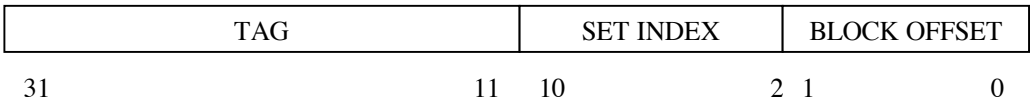


Figure 5.8 32 bit address splitting of the temporal sub-cache

The comparison of this organization with the conventional 16 KB 4 way set associative cache is shown in Figure 5.9.

Cache Type	Size	Total Lines	V+LRU+L+S+T+OFFSET	TAG	Data	Total Bits
Unified 4-way data cache	16KB	4*32	1+2+0+0+0+0	20	128*8	134016
Split 4-way data cache	8KB Spatial	4*16	1+2+1+1+1+5	21	128*8	184320
	8KB temporal	4*512	1+2+1+0+0+0	21	4*8	

Figure 5.9 The relative storage cost for the 4-way unified and split data caches

The potential disadvantage of splitting the data storage capacity in equal size sub-caches is the increased overhead of storing tags in the tag-ram for the temporal sub-cache. For spatial sub-cache to store and track 128 bytes of data, only requirement is one tag to store in the tag-RAM. However, to store 128 bytes into the temporal sub-cache requires 32 separate tags to be stored to track the data. If the tag size is 21 bits, then the overall cost for storing the separate tags for the temporal cache increases dramatically.

The statistical analysis suggests that using a higher spatial cache size is advantageous since, in most cases, the data access behavior either shows more spatial dominance or overlapping spatial and temporal access zones.

If 75% of the space for the spatial sub-cache and 25% for the temporal sub-cache is allocated, then the total size is 12 KB for spatial and 4 KB for temporal caches. In this case, the total tag storage requirement is significantly reduced. The problem still exists due to the fact that, the number of lines in the temporal cache sets is now 256, that reduces the tag storage cost by 50% but the total number of the storage bit requirements is still higher than the unified data cache. Another problem that arises for a 12KB spatial cache is the address mapping issue if 4-way mapping-scheme is still used. In this case, one can make it 3 way for uniform mapping of addresses, but, the cache performance will suffer with comparison to that of a 4 way-mapping scheme of a unified data cache.

Another approach to reduce the storage cost is to reduce the data storage space and use a different space allocation. In this case, if we make the spatial cache size 8-KB and the temporal cache size 4 KB, then the total cost (in bits) of this organization would be

Spatial	64 * (32+1024) =	67584 bits
Temporal	4*256*(26+32) =	59392 bits
Total:		126976 bits = 15872 bytes

Therefore, the total space savings would be in this case with comparison to the unified cache is

((134016-126976)/134016) * 100 = 5.25%

with the sacrifice of the data storage space of 25%. The simulation of the split cache architecture using this space savings plan and the performance of this organization is presented in the next section.

5.5 Performance of the alternate organization

The spatial cache capacity has been kept higher than the temporal cache for several reasons. Statistical analysis reveals that the average data access behavior shows higher spatial locality compared to the temporal locality. In addition, in many cases, the data access shows highly overlapped spatial and temporal access zones. Therefore, it is always advantageous to keep the spatial sub-cache higher than the temporal sub-cache, which also reduces problems such as ***'thrashing'***. The performance metrics and graph of the reduced storage space Split Data caches are presented in Table 5.5 and Figure 5.10 respectively.

Table 5.5 The performance metrics of the reduced data storage space

Locality Cache and the Conventional Data Cache

Benchmark	Miss Rate (%) Unified Data Cache	Miss Rate (%) of the reduced space Split Data Cache	Miss Rate Reduced (%)
espresso	0.30	0.24	20.00
spice2g6	0.94	0.85	9.57
doduc	0.93	0.66	29.03
li	0.65	0.75	-15.38
eqntott	0.59	0.59	0.00
compress	4.39	4.66	-6.15
mdljdp2	1.04	0.86	17.31
wave5	0.16	0.15	6.25
tomcatv	1.35	1.35	0.00
ora	0.07	0.07	0.00
alvinn	0.14	0.14	0.00
ear	0.45	0.44	2.22
sc	0.38	0.30	21.05
mdljsp2	0.54	0.36	33.33
swm256	0.14	0.14	0.00
gcc	1.68	1.36	19.05
nasa7	0.72	0.71	1.39
fpppp	1.02	0.83	18.63

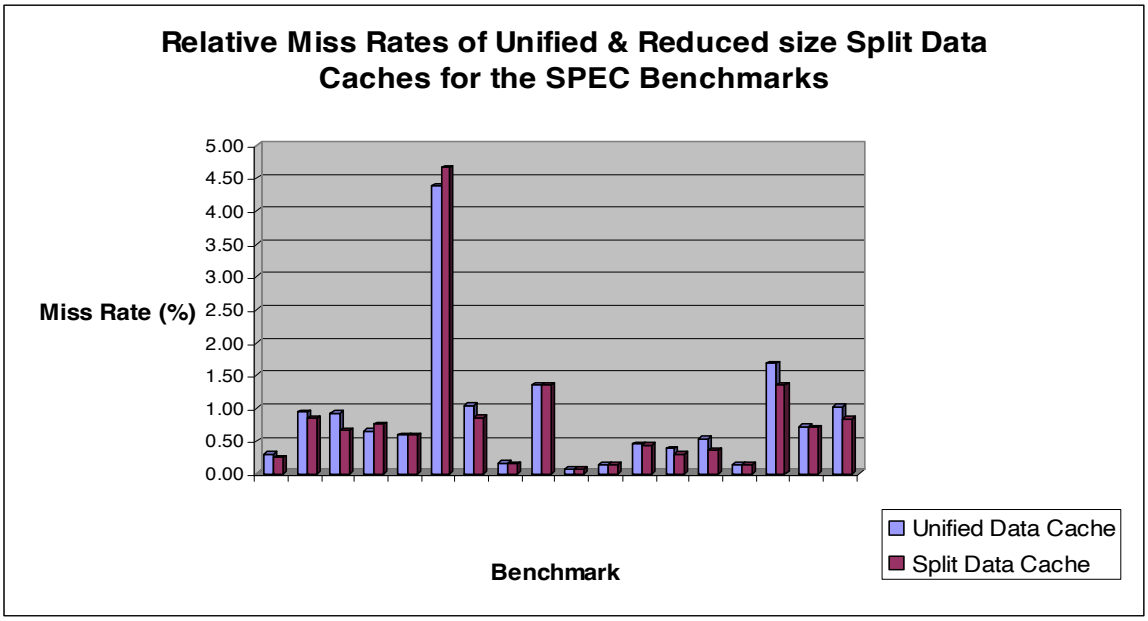


Figure 5.10 The performance comparison between the reduced data storage space Locality Cache and the Conventional Data Cache

The data storage reduction to keep the split data cache size comparable with the conventional cache poses an additional challenge in order to provide better performance than the conventional cache with a reduced data storage resource in the cache. The performance metrics of the implementation strategy as shown in Table 5.5 still signifies that the average performance is better for the split data cache than the unified data cache. The miss rate can be reduced up to 33% than the conventional large data storage capable cache. Only in two cases (***li, compress***), the observed performance degraded due to the inherent requirements of using more cache space. To make the split data cache more comparable with a unified data cache and to increase the performance even more we can utilize the unused 5% space to provide some additional space for the spatial blocks. The associativity of the temporal cache is considered to be reduced to attack the storage space problem. However, the observation of the performance effects reveals that the cache performance degrades more for reduced associativity of the temporal cache than the size. Therefore, the better strategy is to keep the associativity as 4-way and to find some alternate size-tailoring scheme.

5.6 Summary

The performance evaluation of the split data cache is presented in this chapter. The split data cache can provide up to 250% performance boost over the conventional cache, reduce the bus traffic at a similar rate and does not pollute the available cache bandwidth. The contribution of the modified replacement policy and the victim placement policies are also evaluated in detail. The space cost problem due to increased tag overhead of the temporal cache is also presented and an approach to overcome this problem are discussed

with the simulation results of a reduced storage space split data cache organization.

Chapter 6

Conclusions and Future work

The higher main-memory cycle time creates a major obstacle in utilizing the full CPU

performance in modern computer organizations. CPU clock speeds are becoming faster more quickly than the main memory bandwidth is increasing. The off-chip main memory cannot utilize this increased bandwidth due to its' slower access cycle time and the latency introduced due to the chip interface path. Using a cache memory is a remedy to reduce this performance gap. The trend of using cache subsystems is not new. Since the introduction of the first use of cache memories in the early 1980s' research approaches have been investigated that attempt to develop new organizations that can keep pace with increasing CPU bandwidth. The use of small on-chip caches on the same CPU die is a must in modern computer architectures. These on-chip level one (L1) caches can utilize CPU bandwidths more effectively since access delays due to chip packaging constraints are avoided during cache hits.

Due to the finite size of a cache, an optimization scheme is required to utilize the storage assets. Since the cache works based on the locality behavior of the accessed address space of the main memory, many optimizations of the data access layout for locality are found in other research endeavors.. These proposed optimization schemes either perform a better locality distribution during compile time or try to identify the memory reuse zones during run-time in order to keep those data more in the conventional cache architecture. Pre-fetching additional memory blocks greater than the cache line size is also being utilized in almost all-modern computer organizations in order to reduce the overall memory access latency. Though, pre-fetching can hide the memory access latency effectively, it also incorporates the overhead of bringing additional non-usable memory blocks into the cache. Pre-fetching creates cache space pollution and bandwidth waste

and ultimately can reduce the performance benefit obtainable from using a cache.

Instead of blind pre-fetching, proper anticipation of the data locality of the programs can aid in the cache performance. Typical memory access locality behavior shows two major sub-classes: a) Spatial and b) temporal. Average data memory accesses are contributed from both types of the mentioned localities. Conventional caches are not highly optimized for taking advantage of either of these locality behaviors. Rather, they are designed to take operate such that temporal and spatial localities are equally exploited. The locality optimization schemes used by compiler-based approaches can perform better for specific sub-sets of application programs. The dynamic run-time data access pattern fails to take advantage of such compiler based locality optimizations. The advantage of doing compiler optimization is that the compiler can optimize the whole programs' data access patterns when the program contains all the necessary information for the compiler. The challenge in performing dynamic locality estimation during the run-time occurs due to the limited hardware resources that can reasonably be used to detect and optimize locality. This poses the importance of investigating simple non-complex hardware that can be utilized for performing run-time locality prediction at a better rate for the application programs. A run-time locality prediction scheme can further aid to cache data in the appropriate sub-cache that is not possible otherwise in conventional cache architecture. Run-time locality estimation does not suffer from the above mentioned compiler based limitations and it is architecture independent. Therefore, such an organization can be used in any computer system without posing any architectural or compiler constraints. Analyzing this potential, a locality prediction hardware was

investigated throughout this research to define a locality based split data cache which will perform dynamic caching of the data in the appropriate caches during the runtimes of the applications.

The findings of this investigation are the following:

a. The average locality behaviors of the accessed data are a combination of spatial and temporal locality in varying ratios. In some cases the data access behavior shows predominantly spatial or predominantly temporal access behavior and in other cases, it is the combination of both types of locality. Based on this fact, the application programs need a cache organization which can support both types of data access locality in order to achieve a better overall performance.

b. The line-fetching policy can waste about 60% of available cache space in extreme cases. Thus, proper placement of data into separate Spatial and temporal sub-caches can aid in reducing the cache space pollution.

c. Simple locality-estimation circuitry is sufficient to detect the run-time locality behavior of data.

d. The run-time data locality analysis and prediction hardware support can be used to define a split data cache organization which uses two sub-caches termed as spatial and temporal to improve the performance over the conventional cache organization.

e. The performance metrics indicate that the cache resource utilization can be increased up to 150% for many application programs and the average performance can always yield a better result over a comprehensive set of benchmark programs.

f. The only restriction that arises for defining such a split data cache is the increased storage space requirements for the tag ram overhead of the temporal sub-cache. The investigation further shows that the performance boost can also be obtained in most of the cases by using a smaller temporal cache and in that case, the organization may require smaller data storage space than the conventional data cache.

g. The split cache organization does not pollute the cache space by caching non-usable data in the cache. The bus traffic is reduced significantly for using this organization even in the cases where the cache miss rates are similar to the conventional cache. Reduction of the bus data traffic indicates the improved utilization of the cache bandwidth, which might be very useful in modern multi-processor computing organization design.

6.1 Future research direction

The current locality data cache organization is tested using a uniprocessor organization. The utilization of this organization in a multi-processor computing environment needs the cache protocol support that is appropriate for that environment. Fine-tuning of the

locality cache hardware scheme for such an environment may be of interest in future research. The increased storage space overhead due to tag ram of the temporal cache also needs to be further addressed. Several different cache organizations can be tried to improve the performance further and not increasing the storage overhead of the tag ram. Future research should therefore address all of these issues in order to increase the performance boosts available by using a split data cache with a dynamic locality estimation circuit.

Bibliography

1. Patterson D., Anderson T., Cardwell N., Fromm R., Keeton K., Kozyrakis C., Thomas R., and Yelick K., "A Case for Intelligent RAM: IRAM," IEEE Micro, vol.17, (no. 2), March-April 1997, pp.34-44.

2. Patt Y. N., Patel S. J., Evers M., Friendly D. H., and Stark J., "One Billion Transistors, One Uniprocessor, One, Chip," IEEE Computers, September 1997, pp. 51-57.

3. Burger D., Goodman J. R., and Kagi A., "Memory Bandwidth Limitations of Future Microprocessors", In Proceedings of ISCA '96, 5/96, USA.

4. Burger D., Goodman J. R., and Kagi A., "Limited bandwidth to affect processor design," IEEE Micro, November/ ember 1997, pp. 55-62.

5. Handy J., "The Cache Memory book", 2nd Edition, Academic Press, New York, 1998, pp. 188-198.

6. Smith A. J., "Cache Memories," Computing Surveys, vol. 14, 3, September, 1982.

7. Kabakibo A., Milutinovic V., Silbey A., and Furht B., "A Survey of Cache Memory in Modern Microcomputer and Minicomputer Systems," Tutorial: Computer Architecture, IEEE Computer Society Press, 1987.

8. Silbey A., "Improved Cache Scheme Boosts System Performance", Computer Design, Vol. 24, November 1985, pp. 83-86.

9. Tanenbaum A. S., “Structured Computer Organization,” Prentice Hall, New Jersey, 1999.

10. Culler D. E., Singh J. P., and Gupta A., “Parallel Computer Architecture – A Hardware/Software Approach,” Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999.

11. Przybylski S. A., “Cache and Memory Hierachy Design,” Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

12. Mckee S A, Wulf M. A., and Landon T. C., “Bounds on Memory Bandwidth in Streamed Computations,” Proceedings of Europar ’95, Stockholm, Sweden, August 1995.

13. McKee S A, Klenke Robert H, Kenneth L Wright, William A Wulf, Maximo H Salinaas, James H Aylor, and Alan P Baston, “Smarter Memory: Improving Bandwidth for Stream References,” Computers, IEEE Computer Society, July 1998.

14. Deijl E V, Kanbier G, Temam O, and Granston E D, “A Cache Visualization Tool,” Computers, IEEE Computer Society, July 1997, pp. 71-78.

15. Rajamony R., and Cox A. L., "Performance Debugging Shared Memory Parallel Programs Using Run-Time Dependence Analysis," In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Seattle, WA, June, 1997.

16. Lee R. L., Yew P. C., and Lawrie D. H., "Data prefetching in shared memory multiprocessors," In Proceedings of the Int. Conf. on Parallel Processing, 1987, pp. 28-31.

17. Wiel V. S. and Lilja D. J., "When Caches Aren't Enough: Data Prefetching Technique," IEEE Computer, July 1997, pp. 23-30.

18. Mowry T. C., "Tolerating latency through software-controlled data prefetching," Ph. D. Dissertation, Stanford University, March 1994.

19. Yamada Y., Gyllenhaal J., Haab G., Hwu W.W., "Data Relocation and Prefetching for Large Data Sets," In Proceedings of the 27th Annual ACM/IEEE Int. Symp. On Microarchitecture, San Jose, California, November 1994, pp. 217-227.

20. Chen T.F. and Baer J. L., " Reducing Memory Latency via Non-Blocking and prefetching Caches," Technical Report 92-06-03, University of Washington at Seattle, June 1992.

21. Fu J. W. and Patel J. H., "Stride directed prefetching in scalar processors," In Proceedings of the 25th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1992, pp. 245-259.

22. Chen T. F., "Data Prefetching for High-Performance Processors," Technical Report 93-07-01, University of Washington at Seattle, July 1993.

23. Coleman S. and McKinley K. S., "Tile size selection using cache organization and data layout," In Proceedings of PLDI'95, June 1995, pp. 279-289.

24. Jeremiassen T.E. and Eggers S. J., "Reducing false sharing on shared memory multiprocessors through compile time data transformations," In Proceeding of PPOPP'95, July 1995, pp. 179-188.

25. Kodukula I., Ahmed N., and Pingali K., "Data-centric multi-level blocking," Proceedings of PLDI'97, May 1997, pp. 346-357.

26. Lam M. S., Rothberg E.E., and Wolf M.E., "The cache performance and optimizations of blocked algorithms," In Proceedings of ASPLOS'91, April 19991, pp. 63-74.

27. McKinley K. S., Carr S., and Tseng C. W., "Improving data locality with loop transformations," ACM Transaction on Prog. Lang. Syst., 18(4), July 1996, pp. 424-453.

28. Philbin J. E., Anshus O. J., Douglas C. C., and Li K., "Thread scheduling for cache locality," Proceedings of ASPLOS'96, October 1996, pp. 60-71.

29. Mowry T.C., Lam M. S., and Gupta A., "Design and Evaluation of a Compiler Algorithm for Prefetching," In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 27, October 1992, pp. 62-73.

30. Censier L. M., and Feautirier P., "A new solution to the coherence problem in Multi cache systems," IEEE Trans. On Computers C-27:112, 1992, pp. 1112-1118.

31. McKinley K. S., Temam O., "A Quantitative Analysis of Loop Nest Locality", In proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, October, 1996.

32. Kandemir M., Ramanujam J., and Choudhary A., "A Compiler Algorithm for Optimizing Locality in Loop Nests", In Proceedings of the 11th ACM Int'l. Conference on Supercomputing, Vienna, Austria, July 1997, pp. 269-278.

33. McKinley K., Carr S., and Tseng C. W., "Improving Data Locality with Loop Transformations," ACM Transactions on Programming Languages and Systems, 1996.

34. Wolf M. and Lam M., "A Data Locality Optimizing Algorithm," In Proceedings of ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, June 1991, pp. 30-44.

35. Sanchez J, Gonzalez A., "Fast, Accurate and Flexible Data Locality Analysis", In Proceedings of PACT'98, October 13-17, Paris, 1998.

36. Johnson T. L., Merten M. C., Hwu W., "Run-time Spatial Locality Detection and Optimization", In Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 57-64, Research Triangle Park.

37. Moshovos A. I., "Memory Dependence Prediction", Ph.D. dissertation, Department of Computer Science, University of Wisconsin, Madison, 1998.

38. Hennesy J. L., Patterson D., "Computer Architecture A Quantitative approach", 2nd Edition, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996, pp. 390-426.

39. Flynn M. J., "Computer Architecture – Pipelined and Parallel Processor Design", Narosa Publishing House, London, 1996, pp. 396-417.

40. Abraham S. G., Sugumar R. A., Rau B. R., and Gupta R., "Predictability of Load/Store Instruction Latencies", Proceedings of the 26th International Symposium on Microarchitecture, December, 1993, pp. 139-152.

41. Chen T.F., "Reducing memory penalty by a programmable prefetch engine for on-chip caches," Microprocessors and Microsystems, Vol. 21, 1997, pp. 121-130.

42. Callahan D., Kennedy K., and Portefield A., "Software prefetching", Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems, April, 1991, pp. 40-52.

43. Chen T. F., and Baer J. L., "A Performance Study of Software and Hardware Data Prefetching Schemes", Proceedings of the 21st Annual International Symposium on Computer Architecture, April 1994, pp. 69-73.

44. Kaplow W. K., Szymanski B. K., Tannenbaum P, Decyk V. K. and CalTech Jet Propulsion Laboratory, "Run-Time Reference Clustering for Cache Performance Optimization", In Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architecture Synthesis, March 17-21, 1997, Aizu-Wakamatsu, Fukushima, Japan.

45. Avila A., “Reference prediction based on memory access patterns for scientific codes”, Ph.D. Dissertation, University of Arkansas, Fayetteville, December 1998, pp. 15-19.

46. Dwarkadas S., Lu H., Cox A. L., Rajamony R., and Zwaenepoel W., “Combining Compile-Time and Run-Time Support for Efficient Software Distributed Shared Memory”, Dept. of Computer Science, Univ. of Rochester and Dept. of Electrical & Computer Engineering, Rice University.

47. Sanchez F. J., Gonzales A., and Valero M., “Static Locality Analysis for Cache Management,” In Proceedings of IEEE , 1997.

48. Milutinovic V., Milutinovic D., Ciric V., Starcevic D., Radenkovic B., and Ivkovic M., “Some Solutions for Critical Problems in the Theory and Practice of Distributed Shared Memory: Ideas and Implications”, IEEE Proceedings, 1997.

49. Prvulovic M., Marinov D., Dimitrijevic Z., and Milutinovic V., “The Split Spatial/Non-Spatial Cache: A Performance and Complexity Evaluation”, IEEE TCCA Newsletters, 1999, pp.18-25.

50. Prvulovic M., Marinov D., Dimitrijevic Z., and Milutinovic V., "Split Temporal/Spatial Cache: A Survey and Reevaluation of Performance", IEEE TCCA Newsletters, 1999, pp. 8-17.

51. Chan K. K., Hay C. C., Keller J. R., Kurpanek G. P., Schumacher F. X., and Sheng J., "Design of the HP PA 7200 CPU", Hewlett-Packard Journal, February 1996.

52. Gonzalez A., Valero M., and Aliagas C., "A data cache with Multiple Caching Strategies Tuned to Different Types of Locality", Proceedings of ICS 95, pp. 338-347, 1995.

53. Agarwal A., Horowitz M., and Hennessy J., "An Analytical Cache Model", ACM Trans. Computer Systems, Vol. 7, No. 2, May 1989, pp. 184-215.

54. Sanchez J, Gonzalez A., "Data Locality Analysis of the SPECfp95", In Proceedings of the ISCA, 1998.

55. Horowitz M., Martonosi M., Mowry T. C., and Smith M.D., "Informing Memory Opeartions: Providing Memory Performance Feedback in Modern Processors", In Proceedings of the 23rd Annual International Symposium on Computer Architecture, May, 1996.

56. Yan Y., Zhang X., and Zhang Z., "A Memory-layout Oriented Run-Time Technique for Locality Optimization", In Proceeding of 1998 Int'l. Conference on Parallel Processing (ICPP '98), August 1998.

57. Kumar S., Wilkerson C., "Exploiting Spatial Locality in Data Caches using Spatial Footprints", IEEE, pp. 357-368, 1998.

58. Kandemir M., Choudhary A., Ramanujam J., Shenoy N., and Baerjee P., "Enhancing spatial locality via data layout optimizations", In Proceedings of Europar 98, Southampton, UK, September, 1998.

59. Hill M., and Smith A. J., "Evaluating Associativity in CPU Caches," IEEE Trans. On Computers, 38, 12, December, 1989, pp. 1612-1630.

60. Smith A. J., "Line (Block) Size Choices for CPU Cache Memories,", IEEE Trans. On Computers, vol. C-36, 9, September 1987, pp. 1063-1075.

61. Gee J. D., Hill M. D., and Smith A. J., "Cache Performance of the SPEC92 Benchmark Suite," In IEEE Micro, 1993.

62. "New Mexico State University Trace Database", Parallel Architecture Research Laboratory, (Online), Available: ftp://tracebase.nmsu.edu/pub/README., Accessed: January 15th, 2000.

63. Samdani Q. G., Thornton M. A., and Andrews D. L., “A Split Data Cache Organization Based on Dynamic Locality Estimation”, Technical Report, Department of Computer Science and Computer Engineering, University of Arkansas, 2000.

64. Samdani Q. G., Thornton M. A., “Cache Resident Data Locality Analysis”, Technical Report, Department of Electrical and Computer Engineering, Mississippi State University, 2000.

65. Samdani Q. G., Thornton M. A., “A Split Data Cache Organization based on Runtime Data Locality estimation”, Technical Report, Department of Electrical and Computer Engineering, Mississippi State University, 2000.