Split Temporal/Spatial Cache: A Survey and Reevaluation of Performance

Miloš Prvulović*, Darko Marinov**, Zoran Dimitrijević, Veljko Milutinović
Department of Computer Engineering
School of Electrical Engineering
University of Belgrade
P.O BOX 35-54
11120 Belgrade, Serbia, Yugoslavia

*From August 98, at the University of Illinois at Urbana-Champaign
**From August 98, at the Massachusetts Institute of Technology

prvul@computer.org, darko@mit.edu, zorand@galeb.etf.bg.ac.yu, vm@etf.bg.ac.yu

Abstract

The purpose of this paper is to reevaluate the performance of the Split Temporal/Spatial (STS) cache. First we briefly survey the split cache designs found in the open literature. Then we propose quantitative definitions for both temporal and spatial locality. These definitions can be used to represent each split cache design (or any other method for optimized locality exploitation) as a line in a temporal-spatial locality plane. Then we explain the particular process used to evaluate the STS cache design, and finally we present the results of that evaluation. We conclude with possible improvements pointed to by our evaluation results.

Introduction

In recent years, the speed gap between dynamic memories and microprocessors has been steadily increasing. For this reason, a lot of effort is invested into finding ways to reduce or hide memory latency. One of the oldest and most powerful ways of reducing the memory latency is through use of cache memories.

Caches exploit the locality of data access. A small (but fast) memory is able to satisfy most memory access requests issued by the processor, so in most cases there is no need to wait for slow (but large) main memory to respond. Conventional cache designs non-selectively cache all data. If the memory request is not satisfied from the cache, the main memory response has to be waited for. However, at the same time memory contents are brought into the cache in hope that future processor accesses will reuse this data.

The property that the same data items tend to be accessed again in the near future is called temporal locality. Temporal locality is exploited by using smaller block sizes and a cache hierarchy.

Neighboring data items tend to be accessed in the near future, so spatial locality also exists. Spatial locality is exploited by bringing in an entire cache block (with several data words in it).

It is known that not all data exhibit both types of locality, and some data exhibit no locality at all. For example, clearing a large vector involves spatial locality only, because each data item is accessed only once. Frequent accesses to a single global counter are in most cases temporal only, because neighboring data items are often not used so heavily. Most accesses to large hash tables exhibit neither type of locality. If a data item exhibits no temporal locality, bringing it into the cache is useless. If no spatial locality is exhibited by a data item, bringing an entire cache block is even more wasteful.

Several ways to detect differing localities and exploit each in a manner suited to that particular locality type are found in the open literature. Here we will concentrate on an increasingly popular way of exploiting different data localities – splitting the cache into several subcaches, with each subcache designed with a particular type of locality in mind. First, we will survey the existing split cache designs. Then we will provide quantitative definitions for both locality types. Finally, we will provide a way to represent each solution as a line in the temporal-spatial locality plane.

A survey

Exploitation of temporal locality

Much work has been done in exploiting the differing temporal locality of the data accesses, using cache bypassing [Tyson95], [Johnson97a]. The main idea is not to cache non-temporal data. In this way, the entire cache capacity is dedicated to caching the data will be reused.

However, non-temporal data can still exhibit a high degree of spatial locality, so it is useful to cache the corresponding block for some time. Besides, if some data is wrongly presumed as being non-temporal, caching it in a small buffer reduces the penalty of such mistake. For both of these reasons, a small buffer is often added that caches such non-temporal data. The result is a cache split according to the temporal locality, consisting of a larger temporal subcache (main cache) and a smaller non-temporal subcache (buffer).

The assist cache [Chan96] is incorporated into the HP PA7200 CPU. The first level cache consists of a large, external, direct-mapped main cache and a smaller, on-chip, fully associative assist cache. Besides reducing conflicts in the main cache, the assist cache also serves as a "non-temporal" subcache, while the main cache serves as a "temporal cache." At compile time, some memory access instructions are marked as "spatial locality only." Data accessed by these instructions are not cached in the main cache, but only in the assist cache. Other data items are considered as having both temporal and spatial locality and can be cached by the main cache.

The non-temporal streaming (NTS) cache [Rivers96] is similar to the assist cache. The cache is split into a larger, direct-mapped, temporal subcache, and a smaller, fully associative, non-temporal subcache. The decision which subcache should be used is done at runtime. Each entry in temporal subcache is associated with a bit-vector containing one bit for each word used. If no reuse is detected while a particular cache block was in the temporal subcache, it is marked as non-temporal (on eviction from temporal subcache). Future accesses to this memory block would cause a fetch into a non-temporal subcache. If reuse is detected in a non-temporal subcache, the appropriate block is marked back to temporal.

Exploitation of spatial locality

Spatial locality of data accesses has traditionally been exploited by "sequential" prefetching techniques and sectored caches. "Sequential" prefetching techniques, such as "always," "tagged," "miss," and "bi-directional" [Tse98], all prefetch blocks that are neighbors to the block being currently accessed. In this way, prefetching utilizes spatial locality that across cache block boundaries. Unfortunately, as was found in [Tse98], these prefetching techniques increase the bus traffic, make cache tag and data ports busy and introduce pollution into the cache.

Sectored caches exploit the fact that in many cases there is not enough spatial locality to justify bringing the entire block into the cache. Each cache block is therefore divided into several (usually two or four) subblocks that share the common tag, but each has its own validity bit. When a non-spatial access occurs, only one of the subblocks is brought into the cache, therefore reducing bus traffic. However, this approach still wastes cache space in non-spatial blocks, since the unused subblocks can not be used to cache other data.

An interesting approach that avoids this waste of cache space is presented in [Johnson97b]. In this design, the entire cache contains single-word blocks, but for data that are detected as being spatial, several blocks (corresponding to one larger block in split designs) are fetched. In effect, several consecutive blocks containing spatial data behave as if they together form one larger cache block. The main drawback of this approach is that it involves increased tag overhead (more tags for a cache of equal capacity) over the conventional cache, although most of the cache blocks are used by the spatial data.

A multi-word cache block is a waste of cache space if the cached data does not exhibit spatial locality, while a single-word cache block introduces unnecessary tag overhead for data that does exhibit spatial locality. Caches that are split according to the spatial locality exploit these facts. The cache is split into two subcaches. Spatial data is cached in a subcache with a larger block size, while non-spatial data is cached in a subcache that contains smaller blocks (typically one word).

The Split temporal/spatial cache was introduced in [Milutinovic95]. In this design, the cache is split into a temporal subcache (with block size of one word) and a spatial subcache (with a block size of four words). Three possibilities are explored for the ratio between capacities of spatial and temporal primary subcaches: 1:1, 1:2, and 1:4. A secondary cache is included only for the temporal subcache. The decision which subcache to use is done at runtime, profile time or both, using the same counter-based heuristic. Two counters per cache block are present in the spatial subcache. One of those counters, the Y counter, is incremented at each access to a particular cache block. At the same time, the other counter (X counter) is incremented when the upper half of cache block is accessed and decremented at access to the lower half.

When Y counter saturates, and the absolute value of X counter is larger than a specified threshold, the cache block is marked as temporal and removed from the spatial subcache. The "always" prefetch [Tse98] strategy is also employed in this design, but only if both the block causing the prefetch and the prefetched block are marked as spatial. In this prefetching scheme, each spatial cache access causes one consecutive cache block to be prefetched.

A major drawback of this design is the assumption that most data are either temporal or spatial. In fact, most data exhibit both types of locality. Two issues result from this. The first is that a secondary cache would be useful in the spatial part, since it caches the data that exhibits both types of locality as well as spatial only data. The other issue is the counter-based heuristic. If data is accessed so that one word is used several times before the next word is accessed, the heuristic may mark that block as temporal if Y counter saturates while only one half of the block is used. Thereafter, this block is always cached in the temporal part and causes four misses (one per word used) instead of only one it would cause in the spatial subcache.

An interesting piece of research is presented in [Sahuquillo99]. It is an extension of the Split temporal/spatial cache for the shared memory multiprocessor systems.

The Dual data cache was introduced in [Gonzalez95]. In this design, the primary data cache is split into a smaller (33% of total cache capacity) temporal subcache and a larger (66% of the total cache capacity) spatial subcache. Temporal subcache has a block size of eight bytes, while the spatial subcache has a larger block size (16 or 32 bytes). The decision which subcache to use is made at runtime, using a variant of stride directed prefetch predictor. The predictor, in essence, tries to detect if a particular load/store instruction accesses memory at addresses that differ by a constant stride. Using this information, at each cache miss the cache controller decides if data should be cached in temporal subcache, spatial subcache, or not cached at all. Prefetching the next block on a cache miss is also done in this design, but only in the spatial subcache. The most important drawback of this design is that its locality detection is based on instructions than on data. This means that if the CPU accesses a data block in a spatial manner, but words of that block are accessed by different instructions, the detection logic could still decide that a block is non-spatial. Moreover, one instruction could see the block as non-spatial while the other may see it as spatial. This means that a particular word may be cached in both the spatial and non-spatial subcache. While the authors have shown that this does not lead to inconsistencies if properly implemented, the performance may still suffer.

In [Sanchez97], a substantially modified design of the dual data cache is proposed. The temporal subcache in this new design is only a small fully associative buffer (up to 16 single word 8-byte entries) while the spatial subcache remains large and with the block size of 32 bytes. The decision which subcache to use for each data access is made at compile time, by performing data locality analysis.

Moreover, the compiler can mark a particular load/store instruction as non-cached (i.e. bypass), if no locality for that particular data access is expected. The main drawback of this approach is that the changes to the instruction set are required to implement different flavors of load/store instructions. Also, locality information is still based on instructions rather than on data, and locality of many data accesses is difficult to determine at compilation time.

The Array cache [Tomasko97a] design splits the primary data cache into a smaller (25% of the total cache capacity) scalar subcache and a larger (75% of the total cache capacity) array cache. Block size in scalar subcache is 32 bytes while in the array cache it is larger (experimentally varied from 64 to 512 bytes). The decision which subcache to use for a particular data access is done at compile time, by marking the scalar variable accesses to use the scalar subcache, while array accesses go to the array subcache. The main drawback of this approach is that the changes to the instruction set are required to implement different flavors of load/store instructions. In addition, the scalar/array heuristic may be widely inaccurate. Many arrays are accessed in a “random” manner that exhibits almost no spatial locality, while scalars that are used in a particular part of the program are usually stored in neighboring memory words.

Definitions of Locality

Temporal locality

To exploit the differing amounts of locality in different ways, a way to determine the amount of each type of locality is needed. One way is to rely on the intuitive “sense” of what is temporally or spatially local and what should not. For example, it is clear that a variable that is accessed only once during program execution exhibits no temporal locality, while a variable which is accessed heavily throughout the program exhibits a high degree of temporal locality. Similar extreme examples can also be produced for spatial locality. Our intent is to treat different amounts of locality in different ways. It is clear that we want to treat the extreme cases differently, but where should one type of treatment end and the other begin?

The intuition tells us this border should probably be somewhere between the extremes, where intuitive “sense” for locality is not very decisive.

A quantitative definition of each type of locality would be useful. For temporal locality, a good quantitative definition that closely corresponds with the intuition is:

Definition 1: Temporal locality of a data word that is accessed at time T is $\frac{1}{T_{next} - T}$, where T_{next} is the time of the next consecutive access to that particular word.

From this definition, it can be concluded that a temporal locality of a data word that is never accessed again is zero. If time is measured in CPU cycles, the data word that is accessed again in the following cycle has the temporal locality of one. The definition is based on two consecutive accesses to the same word, so temporal locality changes at each access to that particular word. This agrees with the intuitive notion that temporal localities of data change during the execution of a program. A replacement policy "replace the least temporal word", based on our definition, closely corresponds to the ideal replacement policy of "replace the word that will not be accessed for the longest time in the future". Exact spatial locality of a particular word can be determined only with exact knowledge of the future.

For simplicity, let us ignore spatial locality for a while. If the capacity of the cache is N words, at each point of time we want N most temporal words to reside in the cache. A clear criterion for cache bypassing results: a missed word should bypass the cache if it would cause eviction of a word that is more temporally local. As the ideal replacement policy, this criterion also can only be approximated because the exact future behavior is unknown.

A cache controller's bypassing logic faces a question: to bypass or not to bypass. The penalty of bypassing a reference that has high temporal locality is higher than the penalty of not bypassing a reference that has low temporal locality. For this reason, we want to be conservative about bypassing and bypass only the references with very low temporality. The most conservative approach (other than "no bypassing at all" approach) is to bypass only zero-temporality references.

However, a cache of finite capacity can not exploit any amount of temporal locality, even with the ideal replacement policy. If temporal locality of a data word is too low, it will be replaced from the cache before it is reused, so a non-zero temporal locality may still be useless from the perspective of a particular cache.

Definition 2: In a given cache architecture, a particular data word exhibits useful temporal locality if it will be accessed again before it is replaced from the cache.

Given a particular cache design, useless temporal locality is as good as none. We actually should bypass references that do not exhibit useful temporal locality.

It should be noted that usefulness is closely tied to the cache architecture. All other things being equal, a cache of larger capacity can use temporality that is useless for a smaller cache. The most important aspect of this large/small cache bypassing issue is that of a secondary/primary caches. For a cache of infinite capacity, caching even a word with no temporal locality is not harmful, since it does not replace any other word. Most bypassing techniques bypass only the primary cache. The secondary cache is considered large enough to be close to infinite for the purposes of bypassing. Some other bypassing techniques determine whether both, only primary or none of the caches should be bypassed.

As with the replacement policies, we should try to predict future behavior from the known past behavior. A simple way to predict whether temporality of a data word will be useful is to determine if it was useful in the past. In other words, on a cache miss, we check if the missing word was used at least twice during its previous residence in the cache. Two bits are needed for this – one per word to flag the first use and the other to flag a reuse. In caches that have multiple words per block, the entire block is either bypassed or not. Therefore, one bit per word is needed to mark first use of a particular word, and one additional bit per block is needed to mark reuse of any word in that block. This is exactly how the NTS cache detects if a block is temporal (having useful temporal locality) or non-temporal (having no useful temporal locality).

Spatial locality

Spatial locality is harder to quantify than temporal locality. All reasonable definitions of temporal locality state that the temporal locality is the property of a data word to be accessed *again* in the *future*. We see that the temporal locality is expressed by time, even in a qualitative definition. On the other hand, spatial locality is always defined by terms that describe both space (*nearby* addresses to the one accessed now) and time (will probably be accessed in the *near future*). So, spatial locality must be expressed by both time and space.

"Time" component of spatial locality expresses the intuitive notion that, the sooner accesses to nearby words happen, the more spatial locality the word being accessed exhibits. In caches, when a word is accessed, spatial locality enables us to improve performance by fetching words that are near the one being accessed. If those nearby words are not accessed soon enough, they will be evicted from the cache. This means that spatial locality may be useless in a given cache architecture. We want to exploit only useful spatial locality.

Let us now consider the "space" component of spatial locality for a given data word W. It is intuitively clear that the more words from the W's neighborhood are accessed in the near future, the more spatial locality W exhibits. In this paper, we seek to exploit spatial locality through sequential prefetching techniques and varying cache block sizes. Therefore, if some word P is prefetched when W is accessed, we assume then all the words between W and P are either already present in the cache, or prefetched at the same time when P is prefetched. Under these conditions, we can define useful spatial locality:

Definition 3: Let data word W at address A be accessed. Let S be a contiguous sequence of data words that contains the word W, having the property that each word in S would be accessed before it is evicted from the cache, were it fetched when W was accessed. Let L be the number of words in the longest of all such sequences. Useful spatial locality of W is then equal to L-1.

From this definition, we see that if no nearby word is accessed within reasonable time after word W at address A is accessed, the spatial locality of W is zero. If only word at A+1 is accessed within reasonable time, the spatial locality of W is one and so on. Spatial locality also changes with each access to data word W, which also corresponds with our intuitive notion of spatial locality.

In a data cache, two obvious methods of spatial locality exploitation exist: caching multiple words per block and prefetching. Let us consider a cache that consists of blocks that are K words wide. If all data exhibits spatial locality of about K or more, the miss rate is reduced K times. However, if spatial locality L exhibited by the data is lower than K, then the hit rate is reduced only by L, while the remaining word in the cache line are useless. The words that are brought into the cache and never used afterwards only waste cache space and bus bandwidth.

Split caches seek to reduce this waste by fetching and caching large blocks only when spatial locality justifies it. As with bypassing, the penalty for fetching only a single spatial word is higher than the penalty of fetching multiple words on a low spatiality miss. Again, we would want to be conservative about this issue, and the most conservative approach is to fetch a single word only when it is not-spatial (its spatial locality is zero). In all other cases, an entire block of multiple words is fetched.

Exact spatial locality of a data word can not be known when it is accessed, since it requires knowledge of the future. Again, the best we can do is to predict future from the past behavior. Keeping track of spatiality/non-spatiality on a per word basis is impractical, so an entire cache block may be considered non-spatial if during its previous residence in the cache only one of its words was accessed. To detect this, only one bit per word is needed. This bit should be set at first access to that particular word. When the block is evicted from the cache, these bits are checked. If more than one bit is set, then the block is spatial. Otherwise, it is non-spatial. An interesting observation that should be made here is that both the spatial locality detector described here and the temporal locality detector described in the previous section (and used in NTS cache) use these per-word bits for the same purpose.

Therefore, if detection of temporal locality described in the previous section is already present, spatial locality detection can be done using the same per-word bits, at almost no additional hardware cost.

Prefetching is another way of exploiting spatial locality. If spatial locality is more than the size of a cache block, then prefetching can further exploit the existing spatial locality. Again, if there is no spatial locality to justify prefetching, an entirely useless cache line could be prefetched.

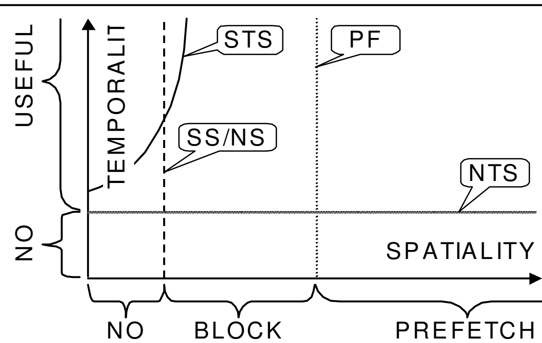

Figure 1: Temporal-spatial locality plane. Definitions of temporal and spatial locality can be used to represent each split cache design, or another method for optimized locality exploitation, as a line in the temporal-spatial locality plane. Several posible split cache designs are presented: STS (Split Temporal/Spatial), SS/NS (Split Spatial/Non Spatial), PF (Prefetch/No Prefetch), and NTS (Non Temporal Streaming).

Although a flag-based decision making similar to those described above can be devised, it is far more common to detect a sequence of accesses and the prefetch the cache block which should be the next in this sequence. When a large amount of spatial locality is present in a program, most probably it comes from a very particular type of data access - array walk-through. Therefore, various methods that try to detect array walk-through should be enough to support our prefetching decisions.

Experimental Set-Up

The performance evaluation of Split Temporal/Spatial (STS) cache in [Milutinovic96a] was done using the ATUM traces [Agarwal86] collected on a DEC VAX by altering its microcode. These traces were fed into software simulators of the STS cache and the conventional cache (for comparison). Performance results are reported in [Milutinovic96a] by giving the percentile performance improvement of STS over the classical cache design, using the average memory latency as the performance indicator. The results are presented in Figure 2. The ATUM traces have the advantage of including not only the memory accesses of a single application. These traces contain all the data accesses of a processor, including OS activities.

However, these traces are quite old and contain fewer data accesses when compared to modern benchmarks. Therefore, we felt that evaluation of STS using the newer IBS traces (which also include OS references) and SPEC benchmarks would increase confidence in the STS design and perhaps direct us toward improved split cache designs.

To re-evaluate the performance of the split temporal/spatial cache, we developed a functional cache simulator. Then we fed the simulator with the data memory access streams extracted from IBS traces, as well as the data memory access streams obtained by running applications from the SPEC 95 benchmark suite.

Development of the cache simulator proved to be the easiest part. The simulator was developed in C++. The major components are a generic trace reader and functional simulator of the classical cache design.

Inheritance mechanisms were then used to derive several variants of STS cache from the classical cache design and to derive the IBS and Pixie trace readers from a generic one.The IBS traces were readily available. However, to get the data access streams of SPEC95 applications we had to employ the Pixie instrumentation tool on an SGI MIPS-based workstation.

SPEC95 benchmark suite is used to evaluate high-performance computers. Each application in SPEC95 has three different input data sets: test, train, and reference. Test data sets should be used to test the correctness of program execution. Training data sets should be used for profiling in optimizing compilers. Reference data sets are intended for use in actual evaluation of computer systems.

A single, non-instrumented SPEC95 application using its reference input data set takes on the order of hours to execute on an actual SGI MIPS R10000-based workstation. It is obvious that it is neither possible to simulate the entire reference stream of a reference run, nor it is possible to store such trace for the purpose of simulation.

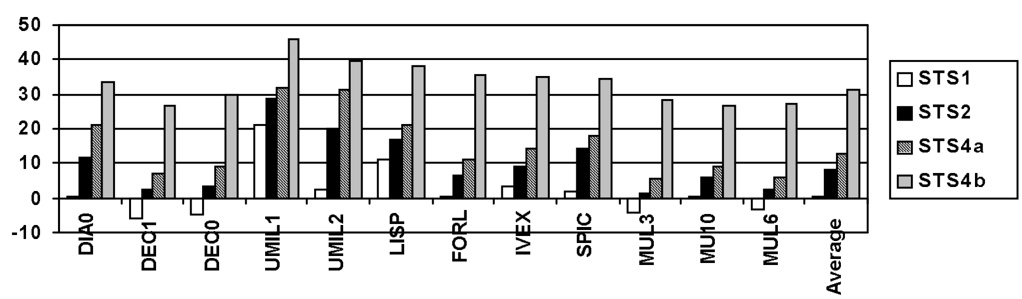

Figure 2: Relative percentile performance improvement of the STS variants compared to the conventional cache (ATUM). Conventional cache is a two level cache with C1 primary and C2 secondary cache. STS1 is STS cache with C1/2 primary spatial part with prefetch mechanism, C1/2 primary temporal part and C2/2 secondary temporal part. STS2 is the same as STS1 with different cache sizes: C1, C1/2, and C2/2, respectively. Cache sizes for both STS4a and STS4b are 2C1, C1/2, and C2/2. The access times for the conventional hierarchy and hierarhies of the temporal parts in the STS1, STS2, and STS4a variants are two cycles for the first level and four cycles for the second level, and one cycle for the first level and two cycles for the second level of STS4b.

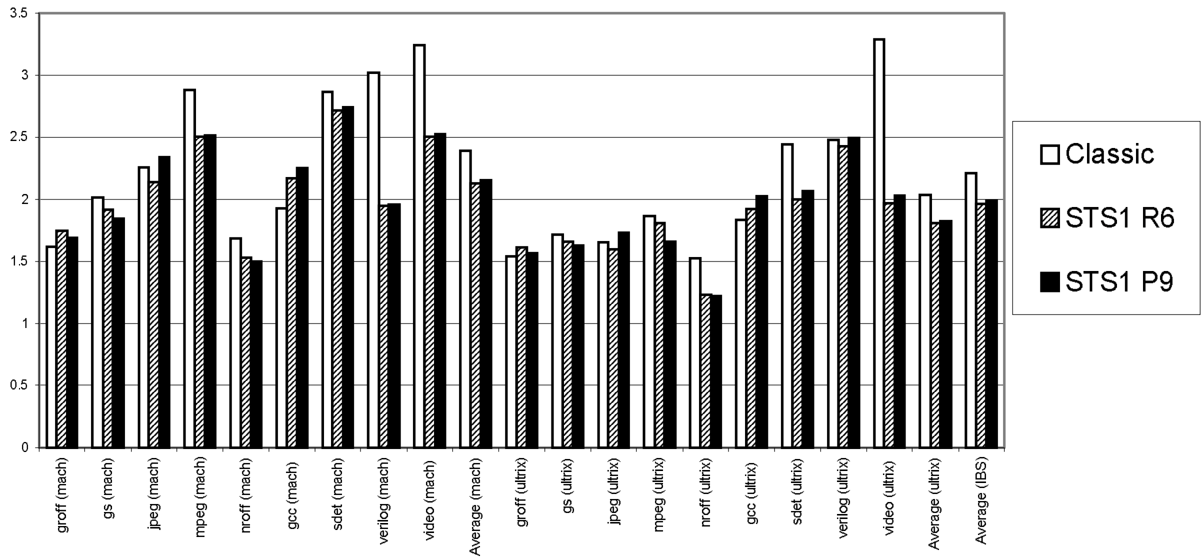

Figure 3: Average memory latency (IBS traces). STS1 R6 is an STS cache that has equal spatial and temporal primary subcaches; locality decision is made at runtime and threshold for X and Y counters are both set at six. STS1 P9 makes locality decisions by profiling using nine as the threshold for both counters. Classic is a conventional cache with a primary cache equal to the sum of capacities of STS subcaches and a secondary cache twice the secondary temporal cache of STS [Uhlig95].

Not storing traces is not a problem. The trace stream generated by the instrumented application can be immediately fed to the cache simulator through a pipe. However, reduction in the number of data accesses to be simulated in each application should maintain the overall data access pattern of that particular application. A sampling technique, in which evenly spaced parts of the trace are simulated while the remaining parts are skipped, is a good candidate.

However, this approach leads to many cache misses at the beginning of each simulated portion, where the working set of one simulated portion is discontinuously exchanged with the working set of another.

The approach used in this paper is to execute a statistics-gathering run using the reference data set of an application. Statistics such as the percentage of writes and reads and several data locality indicators such as conventional cache hit ratios are collected during this run.

Then we run the same application using its training, test, and reference input data set and search for a portion of trace that has statistics close enough to those of the entire reference run.

When evaluating a particular cache design, we simulate the entire data access stream from the beginning of the trace, but collect performance statistics only during execution of the "significant" portion we found. It is unnecessary to simulate past the end of the "significant" portion. It is obvious that it is desirable to find a "significant" portion as close to the beginning of the trace as possible, in order to reduce the number of data accesses simulated before the start of the "significant" portion.

Let P be the number of data memory accesses before "significant" portion begins, and let S be the number of accesses in the "significant" portion. We need to simulate a total of P+S accesses, although statistics are gathered only for S accesses of a "significant" portion. For our evaluation, for each application we seek a significant portion that meets the following conditions:

1) "Significant" portion contains at least 10 million accesses ($S \geq 10{,}000{,}000$)
2) If the percentage of writes in the entire reference run is W_p, the percentage of writes in the "significant" portion must be between $0.99W_p$ and $1.01W_p$.

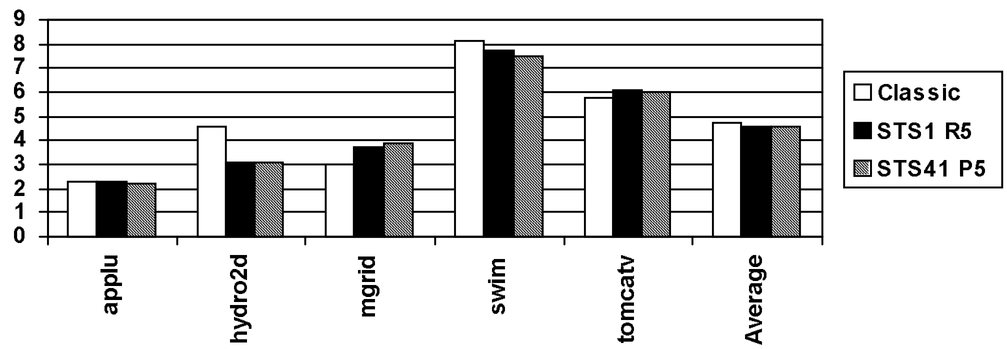

Figure 4: Average memory latency (SPEC). STS1 R5 is an STS cache that has equal spatial and temporal primary subcaches; locality decision is made at runtime and threshold for X and Y counters are both set to five. STS1 P5 makes locality decisions by profiling using five as the threshold for both counters. Classic is a conventional cache with a primary cache equal to the sum of capacities of STS subcaches and a secondary cache twice the secondary temporal cache of STS.

3) Let HR_1 and HR_2 be the hit rates of 2kB primary and 16kB secondary caches in the conventional cache hierarchy for an entire reference run, and let HP_1 and HP_2 be the corresponding hit rates for the "significant" portion. Then the following must hold: $0.99HR_1 \le HP_1 \le 1.01HR_1$ and $0.99HR_2 \le HP_2 \le 1.01HR_2$.

4) We examine other statistics for the "significant" portion and an entire reference run and reject portions that satisfy 1), 2), and 3) but are dissimilar to the entire reference run according to some other statistic.

Of all the portions satisfying the above conditions, we take the one that has the smallest P+S.

Experimental Results

The IBS traces are readily available and contain few enough accesses to be simulated in entirety. Therefore, our initial re-evaluation of STS cache design was done using the IBS traces. We assumed the same STS cache design as in [Milutinovic96a]. All caches are 4-way set associative. The capacity of primary cache in STS is equally divided between its temporal and spatial subcaches. Secondary cache in the conventional cache system is eight times larger than the primary cache. Secondary temporal cache is eight times the size of the primary temporal subcache. Spatial subcache has no secondary cache. Access times differ from those in [Milutinovic96a] because the technology has progressed since then. We assume that each primary cache hit takes one cycle. A secondary cache hit takes four cycles for the first four-byte word and a cycle for each additional word. A cache miss takes 16 cycles for the first word and a cycle for each additional word (i.e., the data busses are all 32 bits wide). Conventional cache and spatial subcache of STS cache have line sizes of four words, while the temporal subcache of STS has the line size of one word.

It is obvious from Figure 3 that, on the average, both runtime and profiling-based STS designs outperform a conventional cache of similar complexity (see [Milutinovic96b] for STS cache complexity evaluation). Such results on IBS benchmarks justify the effort needed to evaluate the STS cache design using the SPEC95 benchmark suite.

When SPEC benchmarks are run on a R10000 SGI workstation, the machine word is eight bytes. For this reason, line sizes are different than in [Milutinovic96a]: temporal cache line size is eight bytes, spatial and conventional caches have 32-byte line size, and the busses are 8-byte (64bits) wide. All the other parameters are as previously described.

Results of evaluation using five SPECfp95 applications are shown in Figure 4. The STS cache outperforms the classical hierarchy on the average, but the difference is not so obvious as in the case of IBS traces. Careful analysis of our simulation results revealed that SPEC benchmarks access fewer non-spatial data words than the IBS traces do. At the same time, the working sets are much larger in the SPEC applications than in the IBS traces. Therefore, most primary cache misses happen in the spatial part of STS, which does not contain a secondary cache.

Analysis of the Results

The STS cache shows improved overall performance over the conventional cache. However, it can be seen that for some applications STS has lower overall performance while for some other applications it shows huge improvements. As the first step in improving the STS design, we want to explain this variation in performance.

Our experiments with varying X and Y counter thresholds (XMax and YMax) indicate optimum values for these thresholds vary greatly between the applications. For example, in runtime-based STS most IBS traces show best results when the XMax and YMax are both six or seven, but JPEG under Ultrix shows best performance when the thresholds are 17. The purpose of counters is to determine if only one half of a spatial cache line is used most of the time. If a single half of particular cache line is accessed Xmax times in a row, then the line is marked temporal. Consider a program that accesses a particular word several times and then moves on to the next word. An STS cache having small Xmax and Ymax may declare the line temporal before the program accesses gets to access the other half, thus moving an essentially spatial line into the temporal subcache. On the other hand, if the thresholds are too high, an actually temporal line gets accessed a few times, but Y counter never saturates and the line is never declared temporal.

To overcome this problem, a different heuristic for detecting temporal locality is needed. We are currently evaluating a scheme in which a simple flag is kept for each part of the spatial cache line. This flag is initially reset to zero and is set to one when that part of the cache line is first accessed. On eviction, if only one of the flags is set, the line is temporal (because only one part of it was accessed). For example, two flags per line are enough to detect if only a single half of cache line was accessed.

Another problem in STS design is that, once the data is declared temporal, there is no way for it to be declared spatial again. If a line that is actually spatial is marked as temporal, it will always be cached in the temporal part. Caching lines that do exhibit spatial locality in the temporal part is costly - instead of incurring one miss per four words in the spatial part, one miss for each of the four words is incurred in the temporal part.

To overcome this problem, a way to detect spatial locality in the temporal subcache is needed. We are currently investigating a scheme in which, at each temporal cache miss, a neighboring line is sought in the temporal cache. If a neighboring line is present, it is evicted, and a larger line is fetched into the spatial part (and marked spatial, of course).

Different applications access different mixtures of spatial and temporal (non-spatial) data. The results in previous section indicate that a second level of cache is needed in the spatial part of STS. However, future experiments are needed to determine if:

1) only the spatial part should have secondary cache
2) each part should have its own secondary cache
3) there should be only one secondary cache shared by both the temporal and the spatial part

Finally, optimum ratio of spatial to temporal subcache capacity varies. This problem can only be attacked by dynamic (runtime) allocation of cache capacity between spatial and temporal subcache. We are currently not considering such designs.

Conclusion

Split caches offer performance improvements over conventional (non-split) caches because they are better suited to exploitation of different locality patterns commonly found in applications. We have evaluated the performance of Split Temporal/Spatial cache design and of conventional two level cache hierarchy, using the IBCS traces and several SPECfp95 applications as workload. We have shown that STS design does indeed offer improved performance, but our experiments indicated that there is still room for refinement of the STS design. This is why we, guided by these experimental results, concluded with a brief overview of possible changes that could improve the STS design. Our future work will concentrate on simulation and evaluation of these improvements.

Acknowledgments

The authors are thankful to their colleagues at the University of Belgrade: Jelica Protić, Aleksandar Milenković, and Igor Ikodinović for the help and numerous sugestions during the work on this paper.

References

[Agarwal86] Agarwal, A., Sites, R., Horowitz, M., "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th Annual Symposium on Computer Architecture,* Tokyo, Japan, June 1986, pp. 119-127.

[Chan96] Chan, K. K., Hay, C. C., Keller, J. R., Kurpanek, G. P. Schumacher, F. X. Zheng, J., "Design of the HP PA7200 CPU," *Hewlett-Packard Journal*, February 1996, pp. 1-12.

[Gonzalez95] Gonzalez, A., Aliagas, C. and Valero, M., "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," *Proceedings of the International Conference on Supercomputing (ICS '95)*, Barcelona, Spain, 1995, pp. 338-347.

[Johnson97a] T. L. Johnson and W. W. Hwu, "Run-time Adaptive Cache Hierarchy Management via Reference Analysis," *Proceedings of 24th International Symposium on Computer Architecture*, Denver, Colorado, June 1997.

[Johnson97b] T. L. Johnson, M. C. Merten, and W. W. Hwu, "Run-time Spatial Locality Detection and Optimization," *Proceedings of Micro-30*, Research Triangle Park, North Carolina, USA, December 1997.

[Milutinovic95] Milutinovic, V., "The STS Cache," *University of Belgrade Technical Report #35/95*, Belgrade, Serbia, Yugoslavia, January 1995.

[Milutinovic96a] Milutinovic, V., Markovic, B., Tomasevic, M., Tremblay, M., "The Split Temporal/Spatial Cache: Initial Performance Analysis," *Proceedings of the SCIzzL-5*, Santa Clara, California, USA, March 1996, pp. 63-69.

[Milutinovic96b] Milutinovic, V., Markovic, B., Tomasevic, M., Tremblay, M., "The Split Temporal/Spatial Cache: A Complexity Analysis," *Proceedings of the SCIzzL-6*, Santa Clara, California, USA, September 1996, pp. 89-96.

[Sanchez97] Sanchez, F. J., Gonzalez, A., Valero, M., "Software Management of Selective and Dual Data Caches," *IEEE TCCA NEWSLETTERS*, March 97, pp. 3-10.

[Sahuquillo99] Sahuquillo, J., Pont, A., "The Split Data Cache in Multiprocessors Systems: An Initial Hit Ratio Analysis," *Proceedings of the 7th Euromicro Workshop on Parallel and Distributed Prcessing*, Madeira, Portugal, February 1999.

[Rivers96] Rivers, J. A., Davidson, E. S., "Reducing Conflicts in Direct-mapped Caches with a Temporality Based Design," *Proceedings of the International Conference on Parallel Processing*, 1996.

[Tomasko97a] Tomasko, M., Hadjiyiannis, S. and Najjar, W. A., "Experimental Evaluation of Array Caches," *IEEE TCCA NEWSLETTERS*, March 97, pp. 11-16.

[Tomasko97b] Tomasko, M., Hadjiyiannis, S. and Najjar, W. A., "Evaluation of a Split Scalar/Array Cache," *Technical report CS-TR-97-105*, Colorado State University, Fort Collins, Colorado, USA, January 1997.

[Tse98] Tse, J., Smith, A. J., "CPU Cache Prefetching: Timing Evaluation of Hardware Implementations," *IEEE Transactions on Computers*, May 1998, pp. 509-526.

[Tyson95] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, ``A modified approach to data cache management," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 93--103.

[Uhlig95] Uhlig, R., Nagle, D., Mudge, T., Sechrest, S., Emer, J., "Instruction Fetching: Coping with Code Bloat," *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 345-356.