SpliCS - Split Latency Cache System

Abstract

Memory access latencies are much larger than processor cycle times, and this gap has been increasing over time. Cache performance is critical to bridging this gap. Since caches cannot be both large and fast, cache design involves hierarchies with trade-offs between access times and miss ratios. This paper introduces a novel primary cache design, called the Split Latency Cache System (SpliCS). SpliCS employs two data stores: a small, fast cache (A) and a larger, but slower cache (B), inclusion is maintained and both caches are accessed in parallel. SpliCS improves the effectiveness of the cache hierarchy by allowing very fast access to some lines while still allowing a large primary cache to be used. Our results using an out-of-order processor model show that relative to a similarly configured traditional cache, SpliCS achieves a 12 to 13% improvement in CPI on commercial applications and 14 to 18% improvement in CPI on some benchmarks from the SPEC suite.

1 Introduction

As the gap between the main memory access time and the processor clock cycle widens, minimizing the data access time is crucial for achieving high performance. Primary caches are designed to provide the most frequently used data items with a minimum delay to the processor. However, caches cannot be both large and fast. Large primary caches have a low miss ratio but large access time; small primary caches decrease the access time, at the cost of increased miss ratio. Our experiments running commercial applications on an out-of-order processor model show that, on an average, doubling the cache size decreases the miss ratio by 25%, but doubling the cache access latency of a cache increases the cycles per instruction (CPI) by 9%. Hence, there is an inherent trade-off between miss ratio and average access time.

Traditional approaches to decrease the average access time focus on:

- Increasing the memory bandwidth: interleaving main memory, wider busses between levels of the memory hierarchy.
- Hiding the memory latency: more levels of memory hierarchy, using prefetching techniques.

Our experiments using one more level of memory hierarchy, an L0 cache, show an average *increase* of 8.3% in CPI over that of a traditional L1 cache. This counterintuitive result confirms Emma's observation that L0 caches cannot be conceptualized in the same manner as the L1 caches [5]. While our L0 cache is nonblocking and services hits to other lines when

misses are still pending, it stalls on a second miss to any line which decreases the overall performance. These issues are further explained in section 4.1.

New designs ([8], [15], [11], [7], [6]) have been proposed that incorporate an additional buffer and do more active primary cache management. These designs focus on modifying placement and replacement policies in the primary cache to reduce miss ratio and have been evaluated on noncommercial applications. Our evaluation of these designs (presented in section 6) reveals that on commercial applications their performance is generally worse than similarly configured traditional caches of higher associativity (>= 2).

In this work we modify the design of the primary caches, to achieve a reduction in access time without increasing the miss ratio and analyze it using both commercial applications and some benchmarks from the SPEC95 suite. Our design, called the Split Latency Cache System (SpliCS), uses 2 caches: a small, fast cache (A) and a larger, slower cache (B). The contents of cache A are strictly included in cache B.

The key difference between SpliCS and a traditional hierarchy is that caches A and B are accessed in parallel; thus cache B is part of the primary cache and a miss in cache A that hits in cache B is served in the cache B's latency time rather than A plus B latency time. More importantly, SpliCS has an advantage over an L1 cache because the very fast cache A (1 cycle access latency in this paper) enables cache B to be larger and slower than a conventional primary cache whose configuration is heavily influenced by cycle time constraints. We evaluate several methods of managing data placement and replacement within SpliCS and compare the performance of SpliCS to a traditional cache hierarchy. Our results on an out-of-order processor model indicate that, relative to a similarly configured traditional cache, SpliCS achieves a 12 to 13% reduction in CPI for commercial applications and 14 to 18% reduction in CPI for the benchmarks from the SPEC suite.

The rest of this paper is organized as follows. In section 2 we present our design, the Split Latency Cache System (SpliCS). In section 3, we describe the experimental framework and the benchmarks used in this study. In section 4 we discuss the results of the simulations for a traditional L1 cache system and a traditional (L0 + L1) cache system. In section 5 we present the results using SpliCS. Relevant prior work is discussed in section 6. Conclusions and promising directions for future work are discussed in section 7.

2 SpliCS - Split Latency Cache System

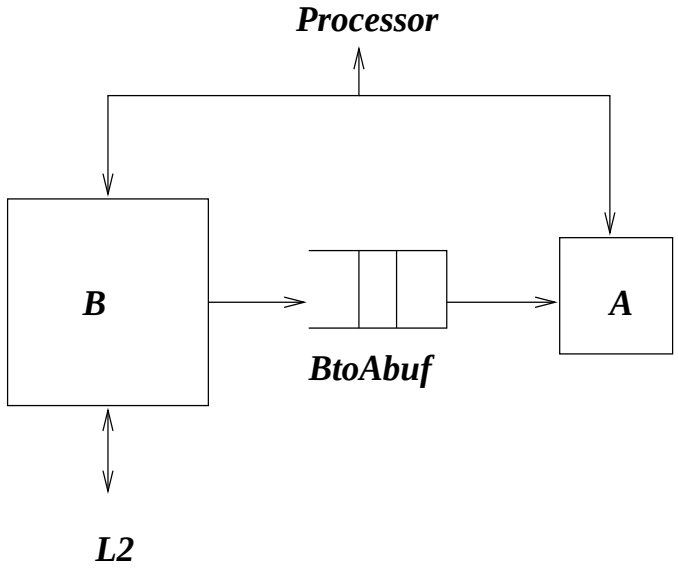

Figure 1: SpliCS Block Diagram

Figure 1 presents a high-level block diagram of SpliCS with 2 caches — labeled A and B. Cache A is smaller and faster than cache B and may have different associativity. When a datum is present in cache A, the cache system can respond more quickly than when the datum is in cache B. Thus, SpliCS has variable latency cache hits, similar to systems that use congruence class prediction, set prediction or other multiple probing techniques [16, 4].

The contents of cache A are strictly included in cache B. This simplifies maintenance of the data and enables the use of a single directory structure for both caches, if desired. and Cache-writes, caused by stores from the processor are written to both caches when the data are present. However, cache B is a write-back cache when viewed from the next level of the memory hierarchy. With the exception of a 1 line buffer for sending data from cache B to cache A, there are no queues associated with cache A. Cache B has an organization similar to a conventional L1 cache. Using the same line size for both the caches simplifies maintenance of inclusion.

In modern microprocessor designs, the latency of the primary cache access path has traditionally been a critical component determining the cycle time. It is desirable to have the largest primary cache possible without extending the cycle time. However, achieving acceptable cache miss rates in multiprocessors necessitates the use of larger primary caches than the desired

cycle time allows. SpliCS is specifically targeted at this issue. SpliCS enables the design of a high capacity primary cache with rapid access to a small portion of the data without negatively impacting cycle time. Our results show that 4 out of 5 hits are serviced by cache A. Thus cache B is used less frequently and therefore can be larger and thus slower than a conventional primary cache. Cache B's larger capacity will result in fewer misses. Furthermore, SpliCS also enables a power tradeoff relative to a conventional cache so that we can design cache B to operate at a lower power and more slowly than would otherwise be required.

The SpliCS directory, however, must be able to respond within the cycle time limit allowed by the small cache A and yet cover the data of the larger cache B. Conversations with our circuit designers indicate that this is quite feasible. By using a single directory, the placement of data in cache A may be restricted by the organization of cache B. If an implementation requires more flexible mappings of data in cache A to boost utilization, a separate directory for cache A can be used. Since cache A is so small — at most 32 lines in this study — the area expense for a separate directory is tolerable. Our results use two directories that are probed in parallel.

2.1 Cache Access Algorithm in SpliCS

We now present the details of the cache access in SpliCS. When we have a reference to a line y the following possibilities exist:

- Case 1: *Line y is in cache A (and in cache B):*

  The requested line is made the most recently used line in its set in cache A and the critical word is forwarded to the processor in 1 cycle (cache A's access latency). As the contents of cache A are strictly included in cache B, the line is also present in cache B. For every *store* in cache A, the line in cache B is also updated. For every hit in cache A, the line in cache B is also made the most recently used line in its set.

- Case 2: *Line y is in cache B, not in cache A:*

  The critical word is forwarded to the processor in 3 or 5 cycles (cache B's latency) and the requested line is made the most recently used line in cache B. In addition, y is promoted to cache A. A first-in-first-out buffer (BtoAbuf) is situated between cache A and cache B, as illustrated earlier in Figure 1. When a line is promoted from cache B to cache A,

it is copied to this buffer and is subsequently copied to cache A from the buffer when A has a free cycle. Since the BtoAbuf has finite size, not all promotions from cache B make it to cache A. For example, when a reply (for a pending miss) from the next level is processed, it might require the line being promoted to be cast out. In that case, the line being promoted is squashed in the BtoAbuf and never makes it to cache A, thereby maintaining strict inclusion of cache A with respect to cache B.

- Case 3: *Line y is not present in cache B (also not in cache A):*
  The requested line is retrieved from the next level of memory hierarchy and placed in cache B and the critical word is bypassed to the processor. If in the process, we replace a line from cache B to make room for line y, then to maintain strict inclusion we must also invalidate the replaced line from cache A (if present). Line y is made the most-recently-used line in cache B and is also copied to cache A.

- Case 4: *Line y is present in the BtoAbuf buffer:*
  It implies line y is in the process of being promoted to cache A as a result of prior reference. However, cache B must also have a copy of this line, the actions are similar to Case 2, except that no new promotion is generated.

- Case 5: *Line y is a pending miss*:
  If a prior request to line y is still being serviced, the cache blocks (i.e., it does not process any new requests) until the pending access to line y is resolved using one of the three cases (1, 2 or 3). Faster ways of handling this case could be devised, but we chose this simple way of handling it for these initial evaluation of SpliCS.

SpliCS caches can be used for instruction or data caches. In this paper, only the data cache uses a SpliCS organization. While we have similar very positive results using SpliCS for an instruction cache, presentation of those data would add little to this paper. When both instruction and data caches use SpliCS organizations, observed reduction in CPI exceed those presented in this paper; they are, in effect, complementary.

3 Experimental Framework

We use trace-driven simulations to evaluate the performance of the primary caches. For this paper we use a cycle-level processor timer developed and used for many years by the microprocessor performance group at the IBM T. J. Watson Research Center for the study of microarchitectures. The timer used in this study was based on a non-product configuration called LOOP which stands for Limited Out-of-Order Processor. In LOOP, instructions can issue out of program order only in a limited instruction window and only if they are completely independent of both the other instructions executing and the instructions before them in the issue queue.

The goal of LOOP is to explore simple pipeline implementation options and as such LOOP does not have reservation stations or register renaming hardware. While it might at first glance seem impractical to omit reservation stations and renaming hardware, these omissions greatly simplify the implementation and allow for a shorter and higher-frequency pipeline. The lack of reservation stations and renaming *clearly* limits the exploitable instruction level parallelism. However, when instructions are scheduled such that independent operations are near each other in the instruction stream, LOOP performs dynamic instruction scheduling to reduce the impact of cache misses or other long latency operations. LOOP does have an extra pipe stage at the end of the pipeline for retiring completed instructions in order. Thus LOOP is aimed at catching the low-hanging fruit of out-of-order execution while enabling a fast, simple implementation. As in the LOOP processor model, the memory system model is detailed and simulates bus transfers, bus contention, and the resulting queuing.

The LOOP processor model is configured to have a 6 stage pipeline with two integer execution units, 1 floating-point execution unit, 1 load-store unit, and a branch unit. It has 4 instruction buffers, each of which can hold 4 instructions. The instruction decode window is capable of holding 8 instructions. LOOP can dispatch at most the first 4 instructions in the decode window. At most 4 operations can be executing in a given cycle. Also, 4 operations can write back their results in a cycle. Only one load or store can be initiated per cycle because there is only one load-store unit and one port to the primary cache. Branch prediction is performed using a 10 entry link-stack, an 8K entry branch target buffer and an 8K entry gshare-like conditional branch predictor. PowerPC's load-multiple and store-multiple

instructions are broken down according to the data bus widths for accessing the instruction and data caches.

The LOOP memory system model has two TLBs (data and instruction) with a 40 cycle penalty for a TLB miss. It has two levels of memory hierarchy (L1 and L2) and a main memory with a 30 cycle access latency. The L2 cache is 4MB with 128 byte lines, 8-way associativity and a 10 cycle access latency for the leading edge. The system has separate L1 instruction and data caches which maintain strict inclusion with the L2 cache. The L1 instruction cache has a 2 cycle access latency and is 64KB with 128 byte lines and 2-way associativity. LRU replacement policy is used by all the caches. The L1-L2 bus is 32 bytes wide; transferring a cache line requires 4 bus cycles. As the focus of this study is the design of the L1 data cache we study its performance for different sizes, associativities and access latencies. The L1 instruction cache and the L2 cache configuration remains unchanged for the rest of the paper.

We describe the latency of the L1 caches in terms of their load-use penalty. The load-use penalty is the number of cycles after a load instruction performs its address generation and before the data can be used by a subsequent instruction. A 1-cycle cache requires one address generation cycle and 1 cache access cycle. Similarly, a 3-cycle cache requires one address generation cycle and 3 cache access cycles.

3.1 Benchmarks Studied

We ran traces of database workloads, namely TPCC and TPCD, and a set of 12 benchmarks from the SPEC suite on RS6000, collected by by the microprocessor performance group at the IBM T. J. Watson Research Center. Table 1 presents the total number of instructions and the memory references of the applications. All the benchmarks in the SPEC suite were run using the reference inputs and we chose the execution profile suggested in [3]. For 9 (*go, ijpeg, li, perl, mgrid, applu, fpppp, tomcatv*) SPEC applications, we skipped the first billion instructions in the program and traced the next 500 million instructions. For *gcc*, we skipped only 100 million instructions (since it ran for about 500 million instructions). For *m88ksim* and *su2cor*, we took 10 uniformly spaced contiguous samples of 50 million instructions. Charney and Puzak [3] validate the segment traces by examining the top ten blocks which account for most of the instructions in the execution and show that the execution profile for the chosen 500 million instructions closely matches the execution profile for the full workloads.

Benchmark	Instructions (millions)	Memory Refs (millions)
TPCC	100	36
TPCD	58	21
126.gcc	453	151
099.go	500	188
132.ijpeg	500	162
130.li	500	218
124.m88ksim	400	145
134.perl	500	215
110.applu	500	225
145.fpppp	500	265
104.hydro2d	500	210
107.mgrid	500	205
103.su2cor	500	230
101.tomcatv	500	208

Table 1: Benchmark characteristics

4 Results with Traditional Caches

To understand the effect of change in size and access latency we evaluated the performance of traditional primary caches with different access latencies and sizes. As the trends in the results for different associativities and line sizes remained the same, we only present the results for the L1 data cache with an associativity of 2 and line size of 128 bytes. The cache size was varied from 32KB to 256KB and the access latency was varied from 1 to 5 cycles. Figure 2 shows the CPI for the database workloads.

A traditional data cache with 1-cycle access latency provides a lower bound on achievable CPI. As the latency increases from 1 to 3 cycles the average CPI increases by 8%. As we further increase the latency from 3 to 5 cycles the CPI increases by 12.5%. Although the miss ratio of these caches decreases with increase in size, the penalties in CPI of the 3 and 5 cycle caches relative to the ambitious 1 cycle cache are considerable.

Figures 3 and 4 show the CPI for the SPEC benchmarks. Our results show that the average miss ratio of SPEC benchmarks is less than 3% for a 64KB L1 data cache. Since the performance was not significantly affected by increasing the cache size further, we present results only for 32KB and 64KB caches. Figures 3 and 4 shows that on an average, the CPI increases by 5.3% as the latency increases from 1 to 3 cycles.

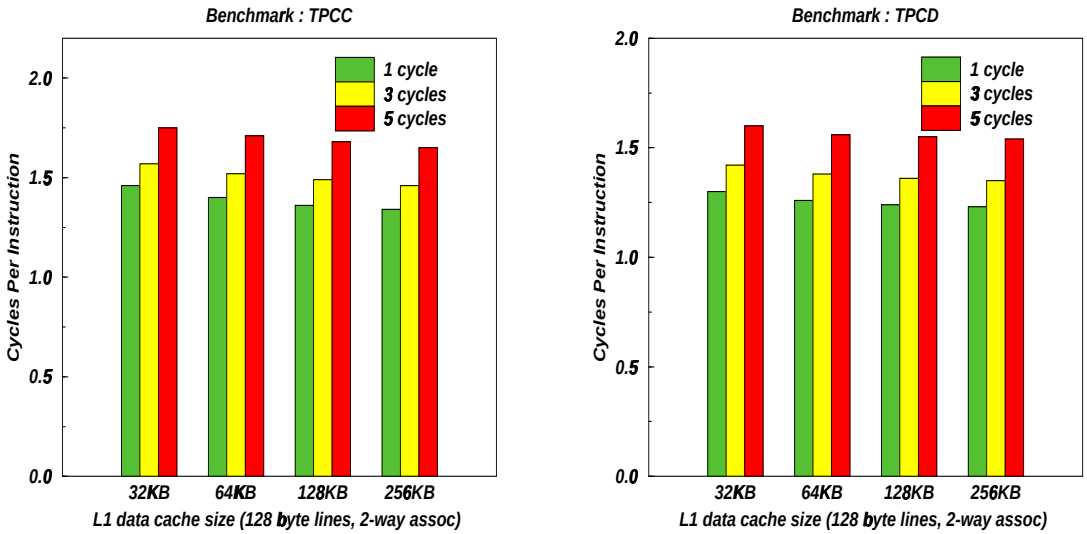

Figure 2: CPI for TPCC and TPCD using traditional L1 data cache

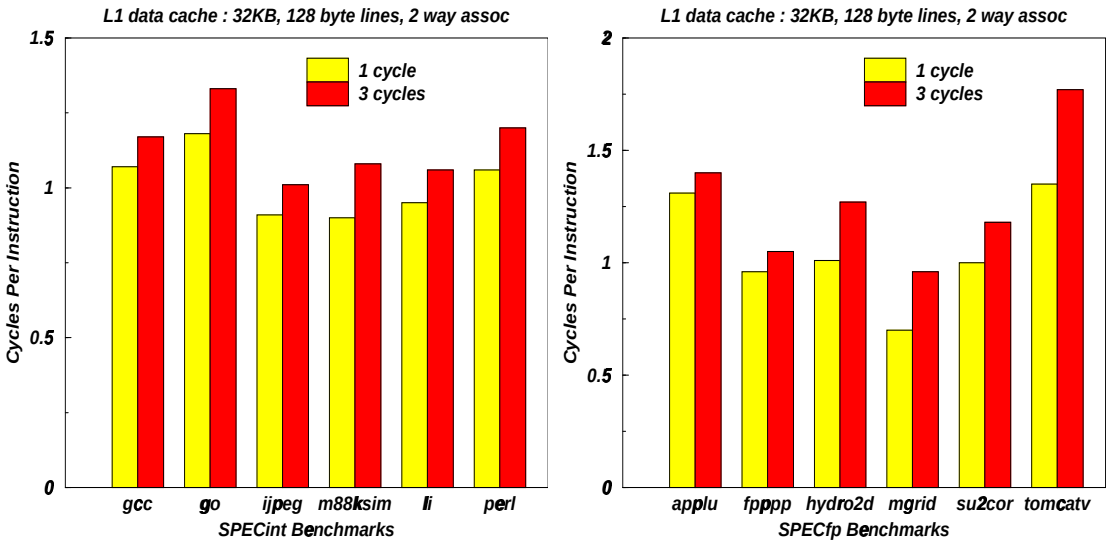

Figure 3: CPI for SPECint and SPECfp using 32KB traditional L1 data cache

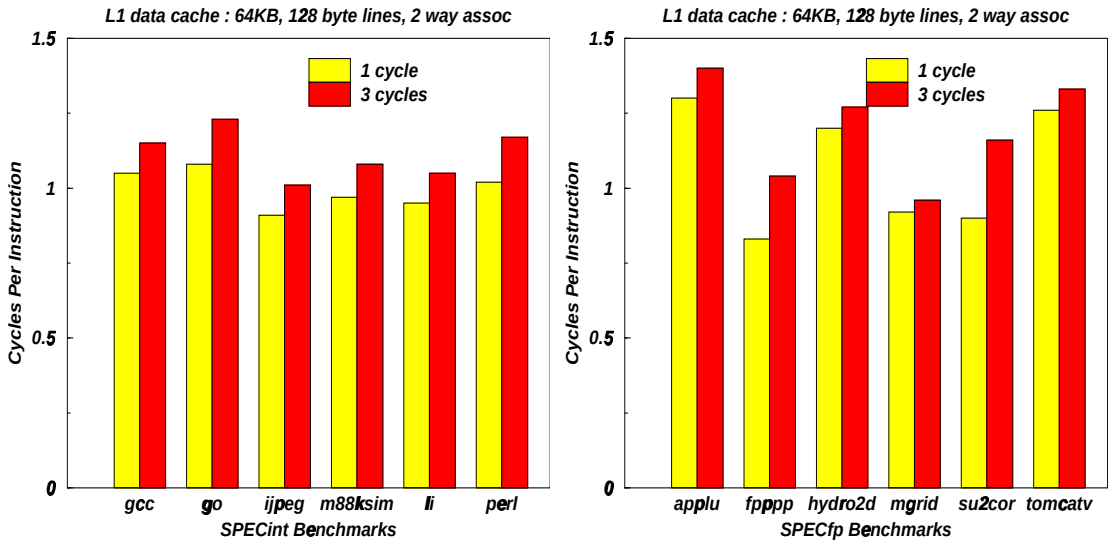

Figure 4: CPI for SPECint and SPECfp using 64KB traditional L1 data cache

From the above results we see that there is a clear trade-off between size and speed of the L1 caches. We also observe that for the database workloads, the 256KB cache with 1 cycle latency performs the best. However, the available technology may make it prohibitively expensive to build such a cache.

The above results motivate the need to explore different alternatives to achieve the performance of a large and fast cache using a combination of a small, fast cache and a larger but slower cache. A straightforward way to achieve the effect of a faster cache is to use one more level of memory hierarchy in front of L1. To evaluate this, we added an L0 data cache to our design. The L0 cache is added between the processor and the L1 data cache as shown in figure 5 and is queried by the processor. The L1 data is queried only on a miss in the L0 cache.

In our simulations, the L0 cache is a write-through cache All the updates to the items in the L0 cache are propagated to the L1 via the L0-L1 bus. The L0 cache model is similar to the detailed L1 cache model.

The L0 cache is a non-blocking cache and processes hits under miss. However, similar to the L1 caches, the L0 cache also "blocks" on a second miss to the same line, i.e., while a miss to a line y is being serviced, the L0 cache processes hits or misses to other lines, but blocks on another reference to line y.

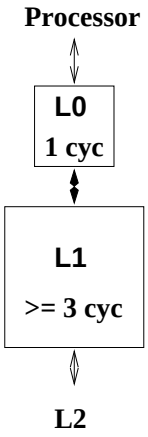

Figure 5: L0 - L1 Cache System

4.1 Results with a L0 + L1 Cache

We simulated an L0 data cache with 128 byte lines and 1 cycle access latency varying its size from 1KB to 4KB. We varied the associativity of this cache from 1 to 4. As the trends of the results remained the same, we present results only for 4KB, 2-way associative L0 cache with 128 byte lines and 1 cycle access latency.

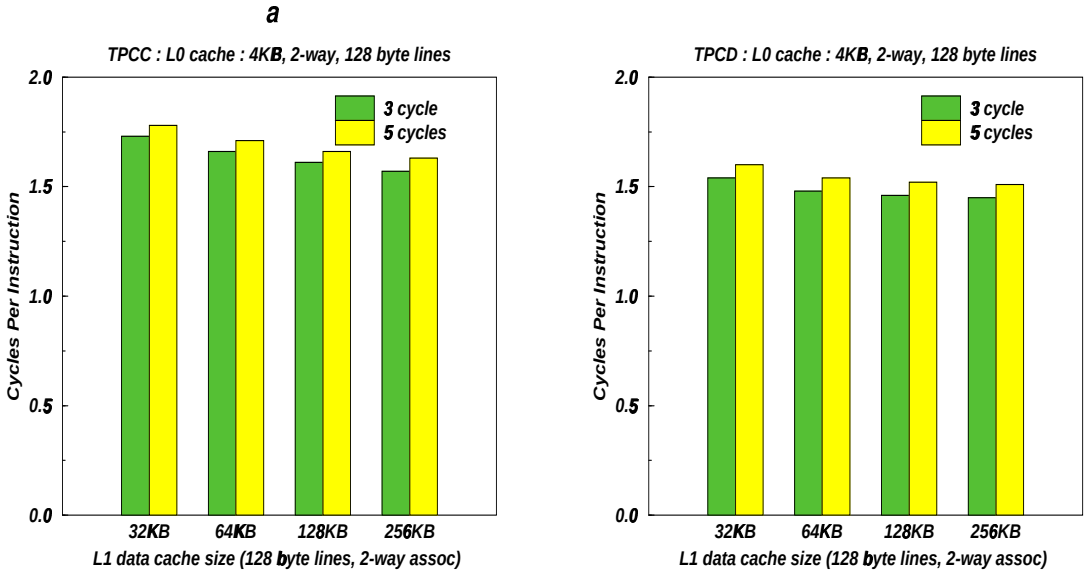

Figure 6: CPI for TPCC and TPCD using (L0 + L1) data cache

Figure 6 shows the CPI for the database workloads. As in section 4, we present results for SPEC benchmarks using only 32KB and 64KB L1 caches with 3 cycle access latency. Figure 7

shows the CPI for the SPECint and SPECfp benchmarks.

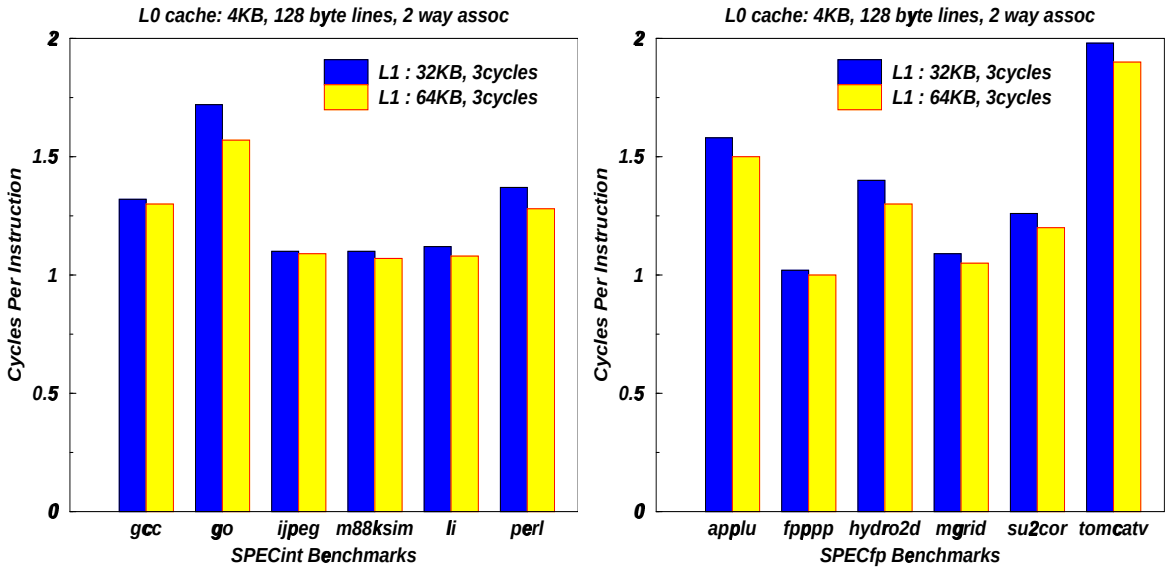

Figure 7: CPI for SPECint and SPECfp benchmarks using (L0 + L1) data cache

Contrary to intuition, we observe that the (L0 + L1) system performs worse than the traditional L1 cache system shown in Figures 2, 3, and 4. It is important to note that although the accesses that hit in L0 take 1 cycle, those that miss in L0 take 4 or 6 cycles if they hit in the L1 cache.

The two main factors contributing to this decrease in the performance of the L0 cache are: (i) the high miss rate of the L0 cache (12 to 13% for the database workloads and 5 to 7% for the SPEC benchmarks), and (ii) the stalls due to L0 "blocking". As explained above the L0 cache stalls when there is a reference to an already pending miss. In Tables 2 and 3 we present the percentage of the total cycles spent in L0 stalls for an (L0 + L1) system and compare it to the traditional system without L0 cache. We see that on an average the (L0 + L1) system has about 3 to 4 times as many stall cycles as the system without the L0. Note that the design with L0 cache has negligible stall cycles due to L1 "blocking". This is to be expected because of the reduction in the number of L1 accesses. However, the L0 stalls are in the critical path and decrease the overall performance.

These results indicate that adding one more level of memory hierarchy is not the solution to approaching the performance of a faster and bigger, but infeasible L1 cache. The rate at

Size of L1 (KB)	% of cycles in stalls for	
	L1	(L0 + L1)
32	4.67	10.07
64	3.63	9.78
128	2.89	8.64
256	2.33	8.18

Size of L1 (KB)	% of cycles in stalls for	
	L1	(L0 + L1)
32	2.21	6.26
64	1.49	5.56
128	1.28	5.37
256	1.12	5.25

Table 2: % of Cycles in Stalls for L1 and (L0 + L1) system using a 3 cycle L1 cache (TPCC and TPCD workloads)

Size of L1 (KB)	% of cycles in stalls for	
	L1	(L0 + L1)
32	2.67	8.01
64	1.01	4.99

Size of L1 (KB)	% of cycles in stalls for	
	L1	(L0 + L1)
32	12.2	19.32
64	11.3	18.52

Table 3: % of Cycles in Stalls for L1 and (L0 + L1) system using a 3 cycle L1 cache (SPECint and SPECfp workloads)

which the requests/replies occur between L0 and L1 is higher than the available bandwidth. Multi-porting the L0 cache may alleviate some of the queuing delays. For example, by allowing the L0 cache to process new requests from the processor using one port and process replies from the L1 using another port some of the queuing delays can be reduced. However, true multi-porting is a very expensive and impractical solution. These results motivated us to design the SpliCS where the L0 "blocking" is eliminated.

5 Results with SpliCS Caches

In this paper we present results for a system with only the L1 data cache designed as SpliCS. The rest of the system is the same as described in section 3. As with the L0 cache we simulated cache A with 128 byte lines and 1 cycle access latency. We varied the size of cache A from 1KB to 4KB and its associativity from 1 to 4. As the trends of the results remained the same, we present results for one of the cache A configurations, namely, 4KB, 2-way associative cache with 128 byte lines and 1 cycle access latency. Cache B's configuration is the same as the L1 data cache configuration in section 4.

To more easily compare the performance of all the systems modeled, we present the CPI

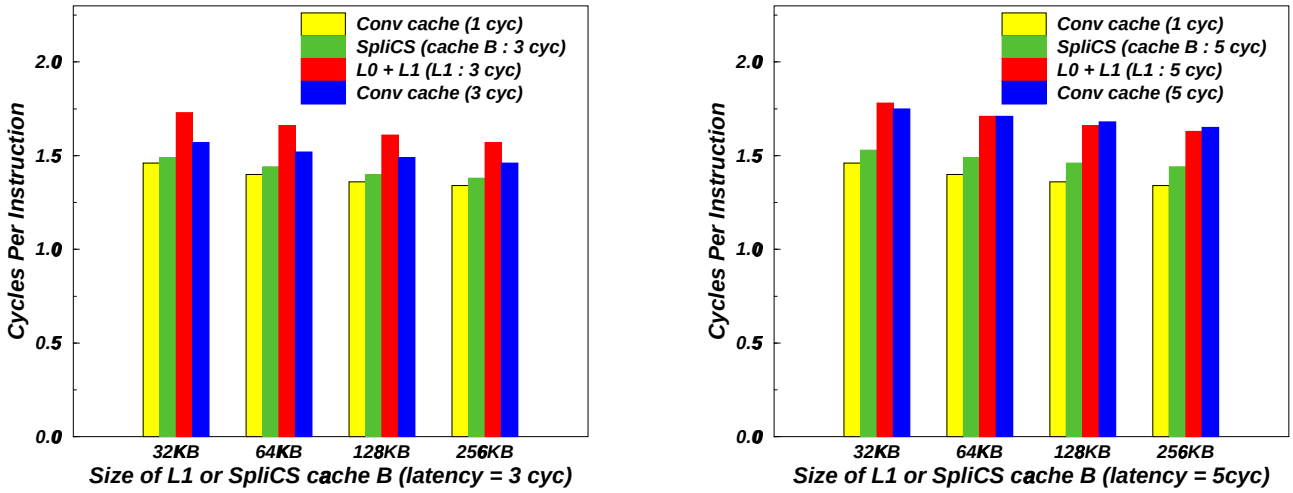

Figure 8: CPI for TPCC

for the database workloads for all the caches in Figures 8 and 9. Each group of bars represents a given size of L1 for the conventional caches or the size of cache B for the SpliCS cache. Within each group of bars, we present from left to right the lower bound CPI using a 1 cycle L1 cache, CPI for SpliCS with 3 or 5 cycle cache B and 1 cycle cache A, CPI for (L0 + L1) cache, with 3 or 5 cycle L1 cache and 1 cycle L0 cache, and CPI for a traditional L1 cache with 3 or 5 cycle latency.

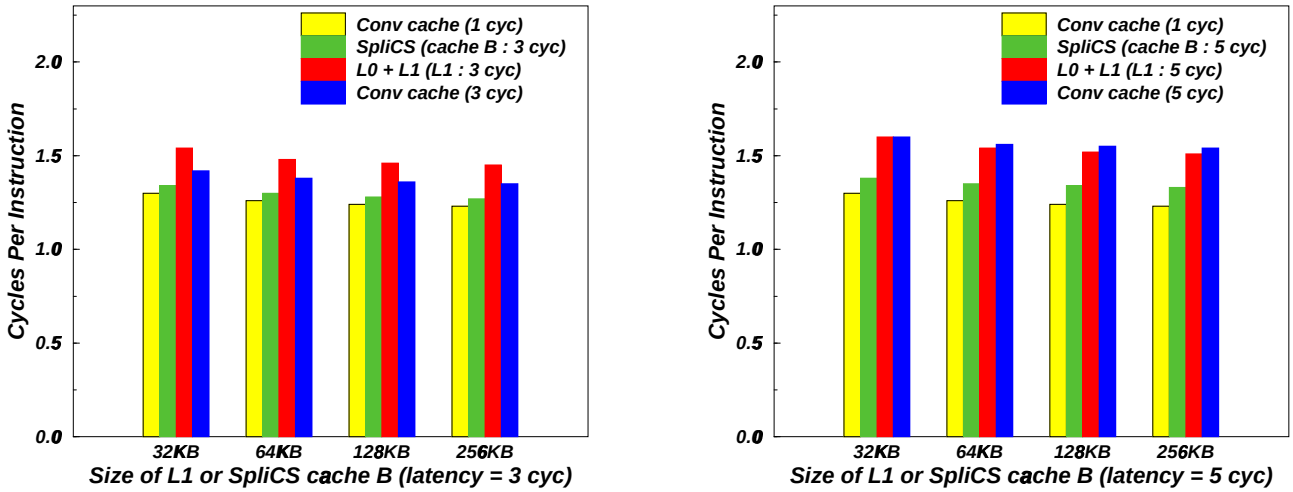

Figure 9: CPI for TPCD

As we increase the latency of cache B, the benefits of the small, fast cache A becomes more evident. For instance, with a 5-cycle cache B, SpliCS achieves higher performance than a traditional L1 cache of twice the size. Similarly SpliCS cache using a slower cache B outperforms a traditional L1 cache with a faster access. For example, the CPI using SpliCS with 5 cycle cache B is lower than the CPI using a 3 cycle traditional L1 cache of similar configuration. This suggests that SpliCS enables building larger caches with much higher latencies without being penalized. Compared to a similarly configured traditional cache SpliCS achieves a 12 to 13% reduction in CPI for commercial applications.

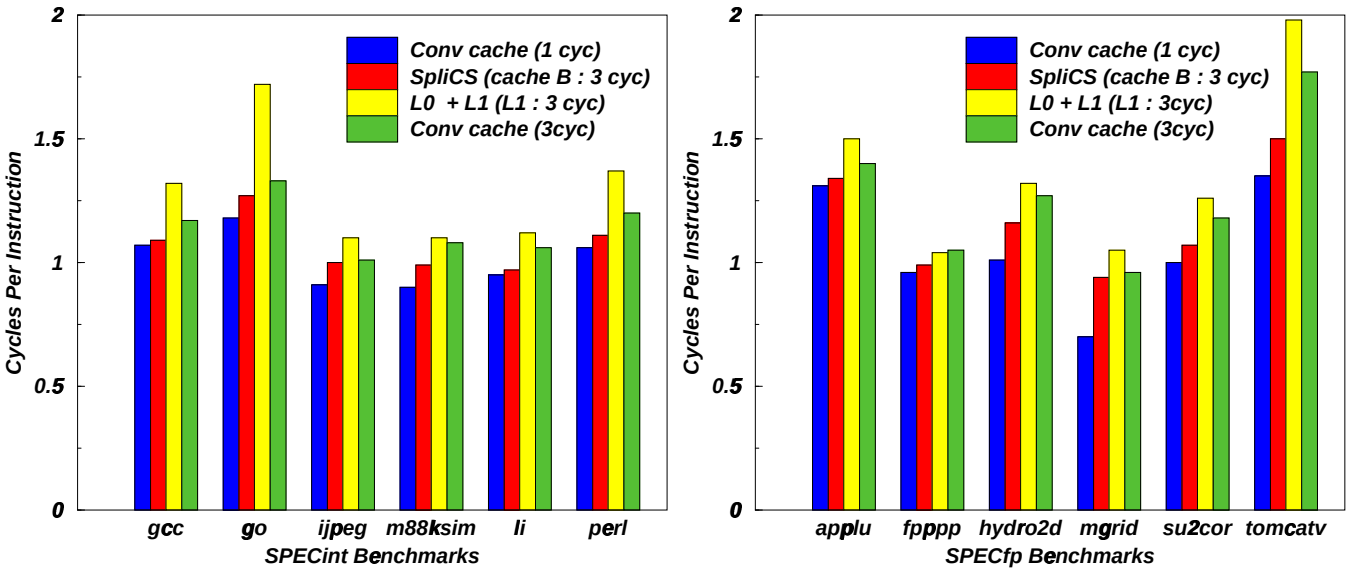

Figure 10: CPI for SPECint and SPECfp using 32KB L1 caches

More importantly, SpliCS outperforms the (L0 + L1) system. The features distinguishing SpliCS from the (L0 + L1) system and accounting for its superior performance are: (i) The A and B caches of SpliCS are probed in parallel. Simultaneous probing of caches A and B eliminate the need to communicate a miss in cache A to cache B. The cache A miss is automatically serviced from the cache B array and the reply is sent directly to the processor, (ii) Cache-writes, caused by stores from the processor are written to both caches when the data are present. This eliminates the need to propagate stores to the next level of memory hierarchy unlike the (L0 + L1) system and, (iii) Cache A of SpliCS does not "block". This is one of the principal contributor for the performance of SpliCS. Our results in section 4.1 show

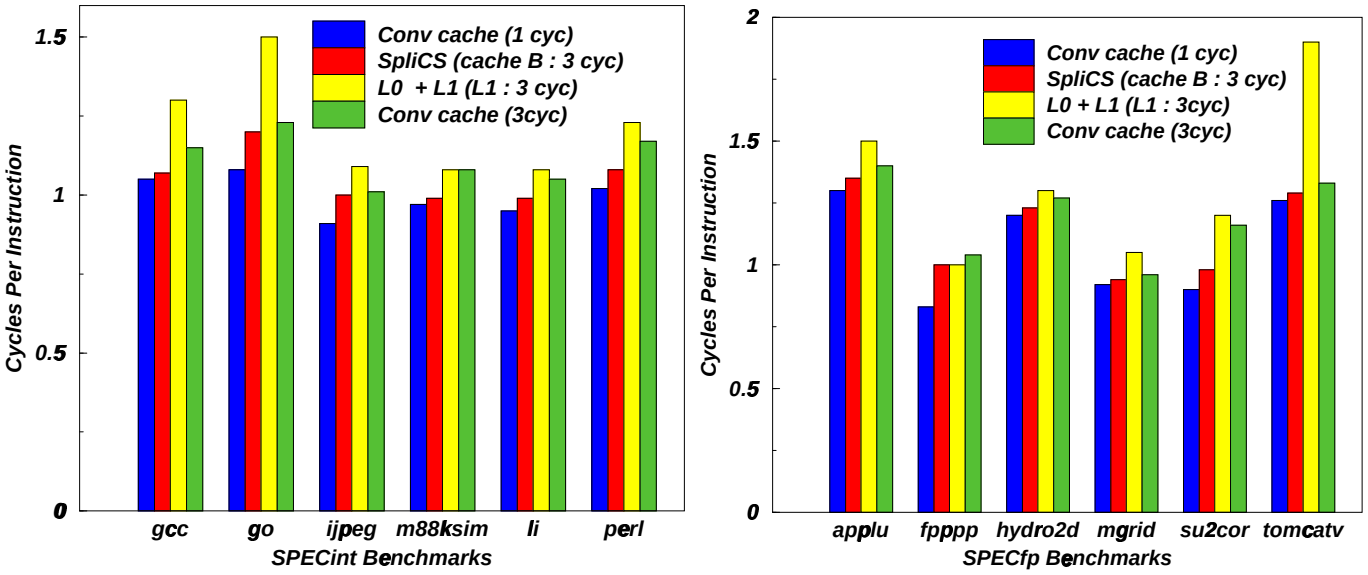

Figure 11: CPI for SPECint and SPECfp using 64KB L1 caches

that the large percentage of stalls in the (L0 + L1) system due to the L0 cache "blocking" is critical to the loss of its performance. This is completely eliminated in the SpliCS.

Similarly, we present the CPI for all the caches using SPEC benchmarks in Figures 10 and 11. The trend of the results remain the same for all the SPEC benchmarks. Results for some of the benchmarks like *go*, *gcc*, and *tomcatv* show that an L0 cache reduces the performance substantially. Moreover, the trends in the results remain the same across different cache sizes. Here again, we see that SpliCS outperforms an (L0 + L1) system and achieves a 14 to 18% reduction in CPI than a similarly configured traditional cache.

6 Related Work

Numerous schemes exist to reduce the effective memory latency seen by the processor ([1], [2], [6], [7], [8], [9], [10], [11], [12], [13], [15], [17], [18], [19], [20], [21]). Among these, we review the prior work that is most related to SpliCS.

The *Victim Caching* approach proposed in [8] introduces a small fully-associative "victim buffer" between the primary direct-mapped(DM) cache and the next level of memory hierarchy. This method reduces conflict misses by storing lines replaced from the primary cache in the victim buffer. References that miss in the primary cache are looked up in the victim buffer.

If found in the buffer, they are served to the processor with a very low miss penalty; and are promoted to the primary cache. Since the victim buffer is fully-associative, lines that reside in the victim buffer may be associated with any set of the primary cache. The victim buffer helps reduce conflict misses in hot sets. However, the swaps between the primary cache and the victim buffer can potentially become a bottleneck. A principal difference between SpliCS and the victim cache is the strict inclusion of cache A in cache B. Inclusion avoids costly swaps between cache B and cache A. As in victim cache, the hit latencies of the two caches are different, but due to its indirect access path a hit in the victim buffer has a larger latency than the larger primary cache. Furthermore, the role of cache A is very different from a victim buffer; our goal in SpliCS is to retain the most recently used lines in cache A.

The *Non-Temporal Streaming (NTS) cache* proposed in [15] supplements the conventional direct-mapped cache with a small parallel fully associative "NTS" cache. Lines that have exhibited *temporal* locality are placed in the main cache and the rest are placed in the NTS cache. This relieves the conflicts in the main cache. However, lines from the NTS cache are not promoted to the main cache even if they exhibit temporal locality during their lifetime in the NTS cache. Subsequent conflicts among *temporal* and among *non-temporal* lines within their available cache space may lead to inefficient utilization of one or the other space. In addition, the NTS method assumes that the access latency is the same for both its caches.

The evaluation of the TPCC workload on the Victim and NTS caches (Figure 12) shows that these schemes do perform better than a single direct-mapped cache. However, a single two-way associative cache generally outperforms both these schemes.

One of the main reasons for the lower performance of NTS and victim caches than a single 2-way associative cache, is the large working set of commercial applications. In addition, the spatial and temporal locality exhibited by commercial applications is difficult to capture at word-granularity. Even though these results are on a no-timing model and therefore the swaps required in victim caches are all free and done instantaneously, victim cache performance is still worse than a single two way associative cache.

The *Assist Cache* [11] implemented in HP-7200, places a conventional primary cache in parallel with a small fully associative "assist" cache, guaranteeing a one-cycle lookup in both units. Blocks from memory are first loaded into the assist cache and are promoted to the primary cache only if they exhibit reuse (as determined statically at compile time). Unlike

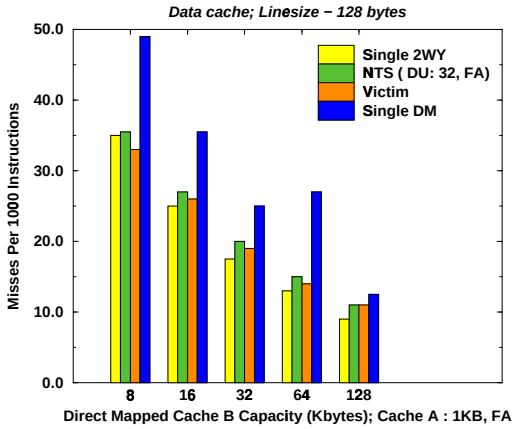

Figure 12: Misses per 1000 instructions for NTS and Victim caches

Assist Cache, SpliCS tracks the cache line reuse dynamically. In essence, the assist cache serves as a staging area for the incoming lines before those marked temporal find a place in the primary cache. The guaranteed one-cycle look-up in both caches restricts the size of the assist cache and increases the cycle time. SpliCS differs from Assist Cache again in the inclusion of cache A in cache B. More importantly, SpliCS allows different latencies for the two caches, thereby relaxing the size restrictions on the larger cache B.

The goal of the above multi-lateral caches (Victim, NTS, or Assist) is to improve the miss rate of the L1 cache using an associated buffer. In essence, the size of the L1 cache is increased to alleviate conflicts and capacity misses. This in turn improves the overall performance. SpliCS is orthogonal to these caches because the main motivation of our design is to provide a faster access to a subset of the cache lines. Since the contents of cache A are strictly included in cache B, SpliCS does not augment the size of L1 using cache A. Therefore, the performance benefits of SpliCS comes from the variable access latency and not due to reduction in the L1 miss rate.

The most related prior work is the Most-Recently-Used(MRU) cache proposed by So and Rechtschaffen at IBM [18]. The MRU cache assumes that the most-recently-used(MRU) cache entry of each set is placed in a fast MRU cache and the rest are placed in a relatively slow primary cache. In this approach, for every change in the MRU we have to move a cache entry from the primary cache to the MRU cache. The caches are accessed in parallel and provide

faster access if the datum is found in the MRU cache. However, unlike the cache A of SpliCS, the MRU cache is always direct-mapped and the number of sets of the MRU cache is the same as the number of sets in the primary cache. Thus the MRU cache helps retain only one line per set and does not alleviate the misses of a "hot" set. SpliCS does not restrict the capacity, associativity or line size of cache A.

The Fetch Target Buffer (FTB) presented in [14], is a design similar in spirit to SpliCS. However, the role of the FTB is to improve the branch throughput by having multiple levels of branch target buffers. Similar to the parallel probing of the caches A and B of SpliCS, the multiple levels of FTBs are probed in parallel to speed-up the access to branch outcomes.

7 Conclusions

In this paper we introduced a new approach to manage the primary cache and improve its performance. SpliCS is a design to optimize the performance of a primary cache using a small, fast cache (A) in association with a larger, slower cache (B). Our results show that, relative to a similarly configured traditional cache, SpliCS achieves a 12 to 13% improvement in CPI on database workloads and 14 to 18% improvement in CPI on some benchmarks from the SPEC suite.

In modern microprocessor designs it is desirable to have the largest primary cache possible without increasing the access latency. Figure 13 shows the alternative designs for the L1 cache. Our results using traditional L1 caches show a big gap in performance between a 1 cycle and 3 or 5 cycles L1 cache. This inherent trade-off between access latency and cache sizes motivates design alternatives to increase the cache size without being penalized by larger access latency. The obvious solution of adding another level of memory hierarchy (L0 cache) does not alleviate the access time constraint. On the contrary, the small size of the L0 cache necessitated by the faster access time requirement, is detrimental to the performance. SpliCS is a solution that combines the best of both worlds by providing faster access without the associated problems of a memory hierarchy and a larger and slower cache to decrease the miss rate. The key features distinguishing SpliCS from a traditional hierarchy are the parallel probing of the cache directories (or using a single directory) for the two partitions to enable faster access to data and minimum communication between the two partitions by using simultaneous writes to

both caches. By careful evaluation using a very detailed timer we have demonstrated that these design choices lead to the superior performance of SpliCS across different sets of applications.

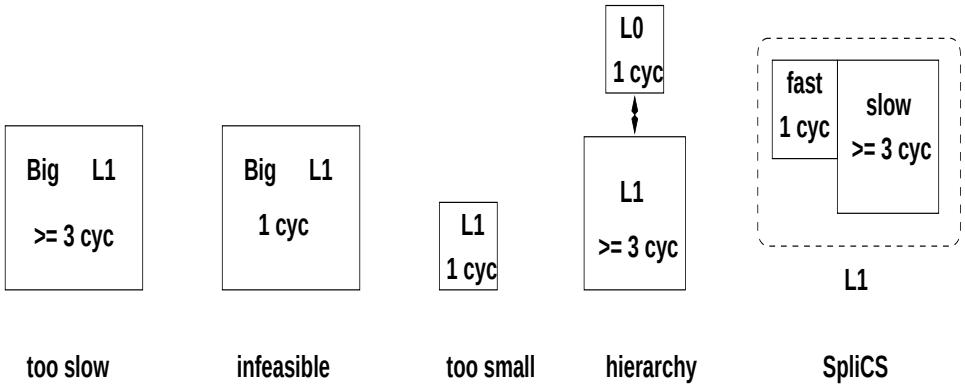

Figure 13: Alternative Designs for L1 cache

SpliCS can be applied to both instruction and data cache. In addition, SpliCS does not place any restriction on the size or associativity of the two caches. Each cache is normally designed to be as large as its access latency allows, subject to space limitation on the die. This flexibility in its design parameters and the better balance it achieves between effective latency and size distinguishes SpliCS from other proposed designs.

References

[1] A. Agarwal, S. D. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct Mapped Caches", *Proceedings of the 20th Int'l Symposium on Computer Architecture, May 1993 pp 179-190.*

[2] Arimilli, R.K. Clark, L.J. Dodson, J.S. Lewis, J.D, "Programmable Unified Split Caching Mechanism for Instructions and Data", *IBM Patent AT997158, Nov. 1997.*

[3] M. Charney, T. Puzak, "Prefetching and Memory System Behavior of the SPEC95 benchmark suite," *IBM Journal of Research and Development, Vol 41, Number 3, May 1997.*

[4] M. Check, T. J. Slegel, "Custom S/390 G5 and G5 microprocessors," *IBM Journal of Research and Development, Vol 43, Number 5/6, September 1999.*

[5] P. Emma, "Understanding Some Simple Processor-Performance Limits," *IBM Journal of Research and Development, Vol 41, Number 3, May 1997.*

[6] A. Gonzalez, C. Aliagas, M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality", *Proceedings of International Conference on Supercomputing, 1995, pp 338 - 347.*

[7] T. Johnson, W. W. Hwu, "Run-time Adaptive Cache Hierarchy Management via Reference Analysis", *Proceedings of the 24th Int'l Symposium on Computer Architecture, June 1997 pp 315-326.*

[8] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", *Proceedings of the 17th Int'l Symposium on Computer Architecture, May 1990 pp 364-373.*

[9] D. Kim, J. Lee, "A Partitioned On Chip Cache For Reducing Memory Access Latency on Shared Memory Multiprocessors", *Proceedings of 11th International Symposium on High Performance Computing Systems, July 1997, pp. 95-104.*

[10] J. Kin, M. Gupta, W. H. Mangione-Smith, "The File Cache: An Energy Efficient Memory Structure", *Proceedings of 11th International Symposium on Microarchitecture, Dec 1997, pp. 184-193*

[11] G. Kurpanek, K. Chan, J. Zheng, E. DeLano, W. Bryg, "PA7200: A PA-RISC Processor With Integrated High Performance MP Bus Interface", *COMPCON Digest of Papers, Feb 1994, pp. 375-382.*

[12] R. L. Mattson, "Partitioned Caches: A New Type of Cache", *IBM Patent SA987016, June 1989.*

[13] Milutinovic, V. Tomasevic, M. Markovi, B. Tremblay, "A New Cache Architecture Concept: The Split Temporal/Spatial Cache", *Proceedings of the 8th Mediterranean Electrotechnical Conference on Industrial Applications in Power Systems, Computer Science and Telecommunications, vol.2 Bari, May 1996.*

[14] G. Reinman, T. Austin, and B. Calder, "A Scalable Front-End Architecture for Fast Instruction Delivery", *26th International Symposium on Computer Architecture, pages 234-245, May 1999.*

[15] J. A. Rivers, E. S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design", *Proceedings of the 1996 International Conference on Parallel Processing, Vol I, August 1996, pp 151-162.*

[16] A. Seznec, "A Case for Two-Way Skewed-Associative Caches", *Proceedings of the 20th Int'l Symposium on Computer Architecture, May 1993 pp 169-178.*

[17] A. J. Smith, "Cache Memories", *Computing Surveys, Sept. 1982, pp 473-530.*

[18] K. So, R. N. Rechtschaffen, "Cache Operations by MRU change", *IBM Report RC-11613, Nov 1985.*

[19] D. Stiliadis, A. Varma, "Selective Victim Caching: A Method to Improve the Performance of Direct Mapped Caches", *IEEE Transactions On Computers, Vol 46, No. 5, May 1997 pp 603-610.*

[20] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, E. S. Davidson, "Evaluating the Performance of Active Cache Management Schemes", *Proceedings of the 1998 IEEE International Conference on Computer Design, Oct 1998.*

[21] A. M. Weiner, "Biased Partitioned LRU Cache Replacement Control", *IBM Disclosure TDB 05-77 p4697, May 1977.*