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 .1.

ew designs ([8], [15], [11], [ ], [ ]) 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) 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 C95 suite. Our design, called the plit Latency Cache ystem (C pli), uses 2 caches: a small, fast cache (A) and a larger, slower cache ( ). The contents of cache A are strictly included in cache .

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

The rest of this paper is organized as follows. In section 2 we present our design, the plit Latency Cache ystem C pli). In section 3, we describe the experimental framework and the benchmarks used in this study. In section 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 C pli . elevant prior work is discussed in section . Conclusions and promising directions for future work are discussed in section .

2 SpliCS - Split Latency Cache System

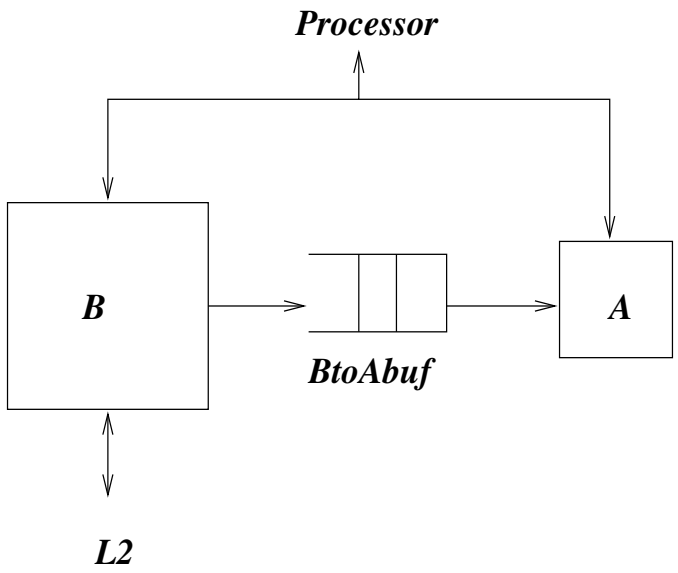


Figure 1: SpliCS Block Diagram

Figure 1 presents a high-level block diagram of C pli with 2 caches labeled A and . Cache A is smaller and faster than cache 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 . Thus, C pli has variable latency cache hits, similar to systems that use congruence class prediction, set prediction or other multiple probing techniques [1 ,].

The contents of cache A are strictly included in cache . 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 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 to cache A, there are no queues associated with cache A. Cache 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. C pli is specifically targeted at this issue. C pli 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 out of 5 hits are serviced by cache A. Thus cache is used less frequently and therefore can be larger and thus slower than a conventional primary cache. Cache 's larger capacity will result in fewer misses. Furthermore, C pli also enables a power tradeoff relative to a conventional cache so that we can design cache to operate at a lower power and more slowly than would otherwise be required.

The C pli 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 . Conversations with our circuit designers indicate that this is quite feasible. y using a single directory, the placement of data in cache A may be restricted by the organization of cache . If an implementation requires more exible mappings of data in cache A to boost utilization, a separate directory for cache A can be used. ince cache A is so small—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.

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

- Case 1: *L ine 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, the line is also present in cache . For every *store* in cache A, the line in cache is also updated. For every hit in cache A, the line in cache is also made the most recently used line in its set.
- Case 2: *L ine is in cache B, not in cache A:*
  The critical word is forwarded to the processor in 3 or 5 cycles (cache 's latency) and the requested line is made the most recently used line in cache . In addition, is promoted to cache A. A first-in-first-out buffer (toAbuf) is situated between cache A and cache , as illustrated earlier in Figure 1. When a line is promoted from cache 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 toAbuf has finite size, not all promotions from cache 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 toAbuf and never makes it to cache A, thereby maintaining strict inclusion of cache A with respect to cache

- Case 3: *L 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 and the critical word is bypassed to the processor. If in the process, we replace a line from cache to make room for line , then to maintain strict inclusion we must also invalidate the replaced line from cache (if present). Line is made the most-recently-used line in cache and is also copied to cache A.

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

- Case 5: *L is a pending miss*:
  If a prior request to line is still being serviced, the cache blocks (i.e., it does not process any new requests) until the pending access to line 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 C pli .

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

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 I T. . Watson Center for research 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 stage pipeline with two integer execution units, 1 oating-point execution unit, 1 load-store unit, and a branch unit. It has instruction buffers, each of which can hold inst The instruction decode window is capable of holding 8 instructions. LOOP can dispatch at most the first instructions in the decode window. At most operations can be executing in a given cycle. Also, 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. ranch prediction is performed using a 10 entry link-stack, an 8 entry branch target buffer and an 8 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 TL s (data and instruction) with a 0 cycle penalty for a TL 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 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 with 128 byte lines and 2-way associativity. L U replacement policy is used by all the caches. The L1-L2 bus is 32 bytes wide; transferring a cache line requires 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. imilarly, a 3-cycle cache requires one address generation cycle and 3 cache access cycles.

We ran traces of database workloads, namely TPCC and TPC , and a set of 12 benchmarks from the C PE ite on 000, collected by by the microprocessor performance group at the I T. . Watson Center esearch. Table 1 presents the total number of instructions and the memory references of the applications. All the benchmarks in the C PE ite were run using the reference inputs and we chose the execution profile suggested in [3]. For 9 (*go, i peg, li, perl, mgrid, applu, fpppp, tomcatv* C PE 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.

	c	str ct ar r s s
TPCC	100	3
TPC	58	21
12 .gcc	53	151
099.go	500	188
132.i peg	500	1 2
130.li	500	218
12 .m88ksim	00	1 5
13 .perl	500	215
110.applu	500	225
1 fppp p	500	2 5
10 .hydro2d	500	210
10 .mgrid	500	205
103.su2cor	500	230
101.tomcatv	500	208

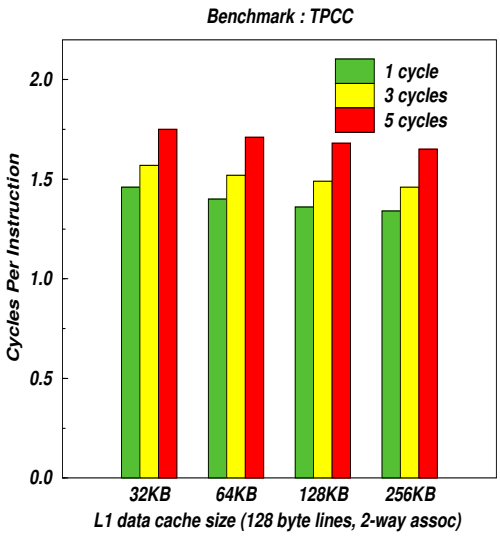
le 1: ae B m kar ar c ea ir i

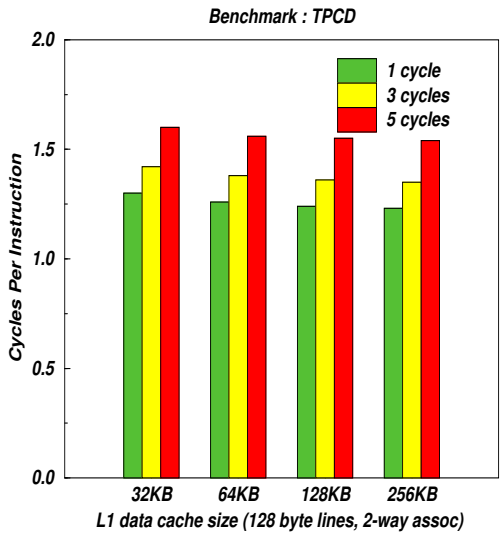
esu it lt hs r aditiona l Caches

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 32 to 25 and the access latency was varied from 1 to 5 cycles. Figure 2 shows the CPI for the database workloads.

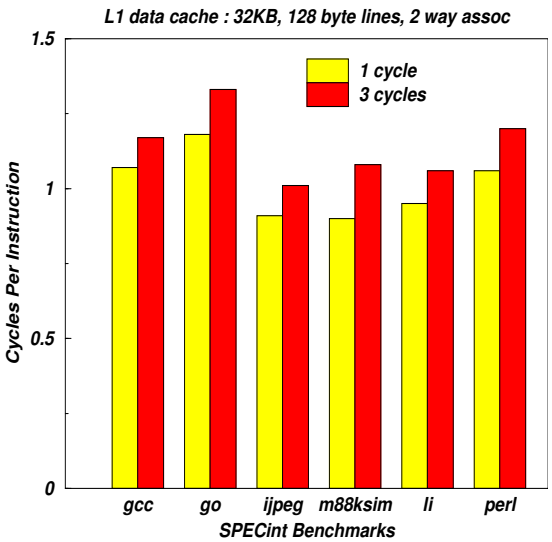
A traditional data cache with 1-cycle access latency provides a lower bound on achievable CPI. As the latency increases from 1 to 3 cycles the average CPI increases by 8%. As we further increase the latency from 3 to 5 cycles the CPI increases by 12.5%. Although the miss ratio of these caches decreases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

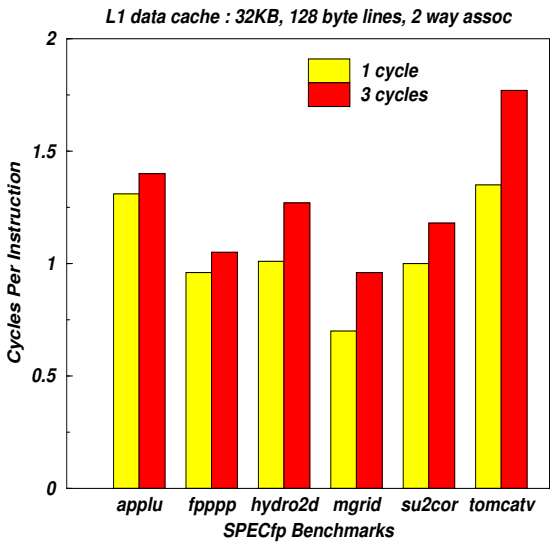
Figures 3 and show the CPI for the C B Benchmarks. Our results show that the average miss ratio of C B Benchmarks is less than 3% for a L1 data cache. ince the performance was not significantly affected by increasing the cache size further, we present results only for 32 and caches. Figures 3 and shows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



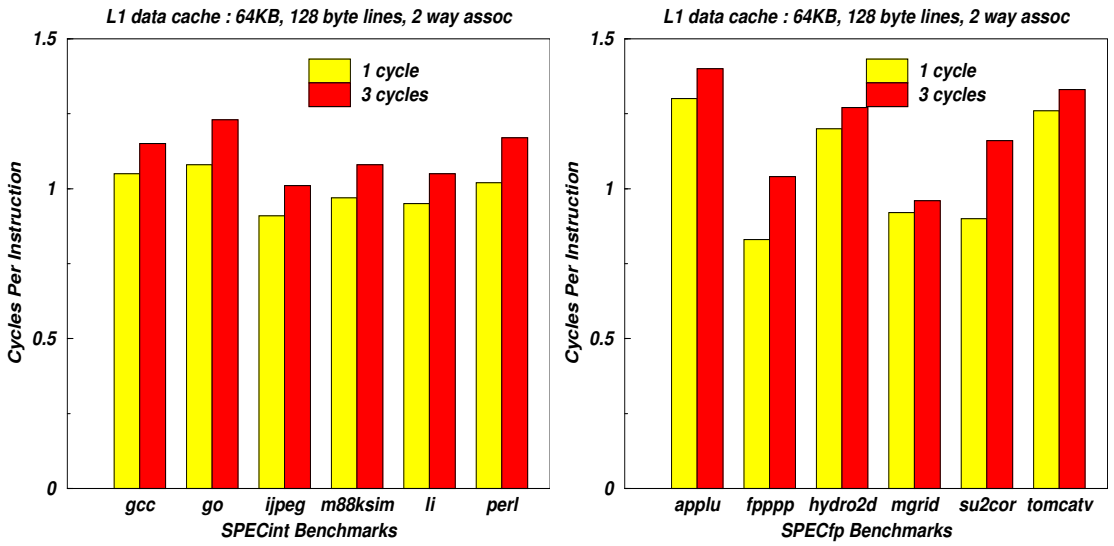


F igu Cre: or D tC Cra a io i l i agcac a C





F igu3: Cre or S a S C p iu 3 C ri inoB l g ia a cac a



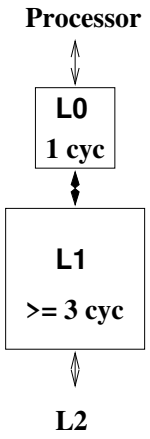
F igu Cre: or S a S C p iu C r i io l g Bda cac e

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 25 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.

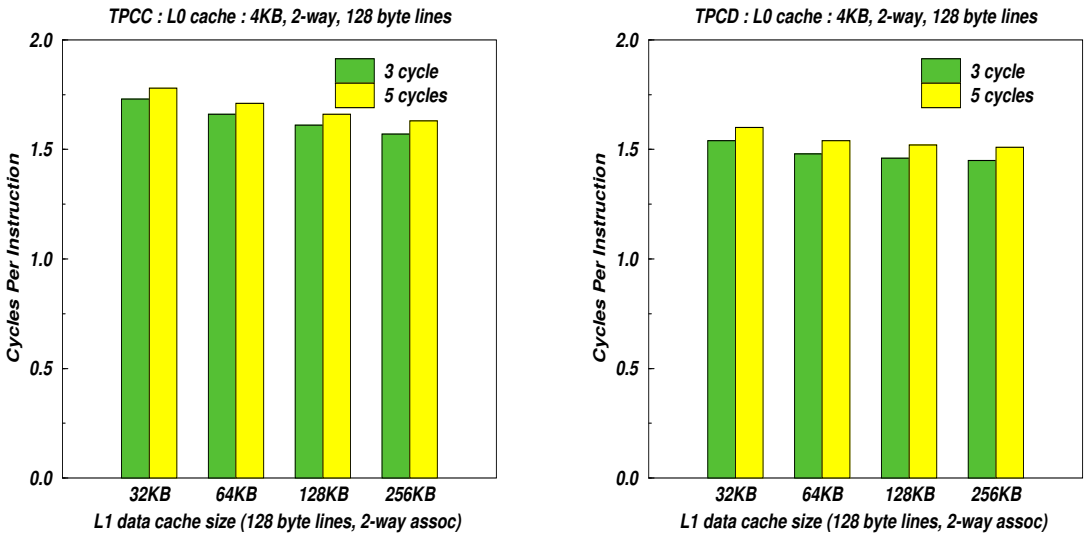
In our simulations, the L0 cache is a write-through cache All the updates to the items in the L0 cache are propagated to the L1 via the L0-L1 bus. The L0 cache model is similar to the detailed L1 cache model.

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 is being serviced, the L0 cache processes hits or misses to other lines, but blocks on another reference to line .



F igu r e Cac e S e m 1

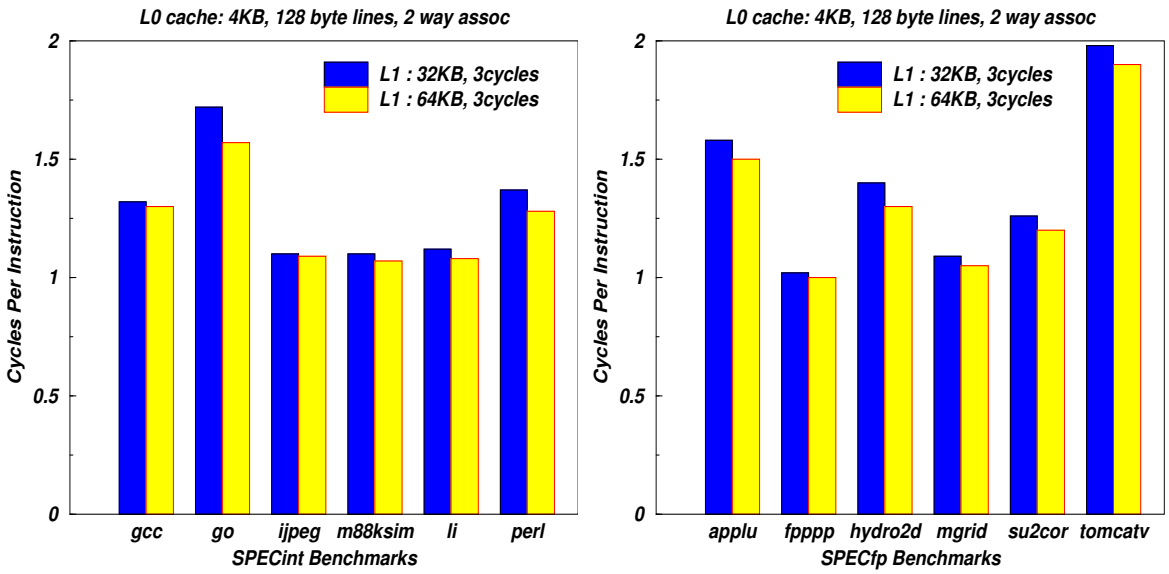
We simulated an L0 data cache with 128 byte lines and 1 cycle access latency varying its size from 1 to . We varied the associativity of this cache from 1 to . As the trends of the results remained the same, we present results only for , 2-way associative L0 cache with 128 byte lines and 1 cycle access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



F igu Cr oe D tC C a i a cacg e a 1 C

Figure shows CPI for the database workloads. As in section , we present results for C Benchmarks using only 32 and L1 caches with 3 cycle access latency. Figure

shows the CPI for the CintPE and CfpPE benchmarks.



F igu : Cre or S a S C p i c m Ck ar u ia cac e a g 1

Contrary to intuition, we observe that the (L0 L1) system performs worse than the traditional L1 cache system shown in Figures 2, 3, and . It is important to note that although the accesses that hit in L0 take 1 cycle, those that miss in L0 take or 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 % for the C PE 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 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

	c		c	s	sta	c s
			+			+
32	.	10.0		32	2.21	.2
	3. 3	9. 8			1. 9	5.5
128	2.89	8.		128	1.28	5.3
25	2.33	8.18		25	1.12	5.25

le a : %C co S all or a 1 e m u 3 ce 1 aicle cc gla
(TPCC and TPC workloads)

	c		c	s	sta	c s
			+			+
32	2.	8.01		32	12.2	19.32
	1.01	.99			11.3	18.52

le 3a : % oc C c S all or a 1 e m u 3 ce 1 aicle cc gla
(CinE and CfpE workloads)

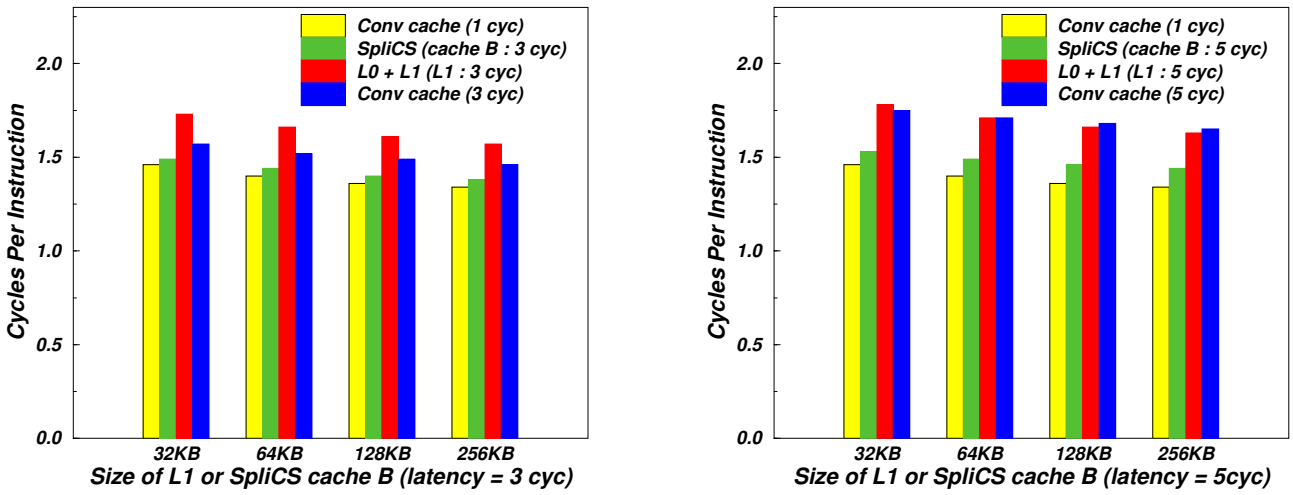
which the requests/replies occur between L0 and L1 is higher than the available bandwidth.

ulti-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 C pli where the L0 blocking is eliminated.

5 esu it lt Es pliCS Cache s

In this paper we present results for a system with only the L1 data cache designed as C pli . 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 1 to and its associativity from 1 to . As the trends of the results remained the same, we present results for one of the cache A configurations, namely, , 2-way associative cache with 128 byte lines and 1 cycle access latency. Cache 's configuration is the same as the L1 data cache configuration in section .

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



F igu Cr : oe C C

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 for the C pli cache. Within each group of bars, we present from left to right the lower bound CPI using a 1 cycle L1 cache, CPI for C pli with 3 or 5 cycle cache 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.

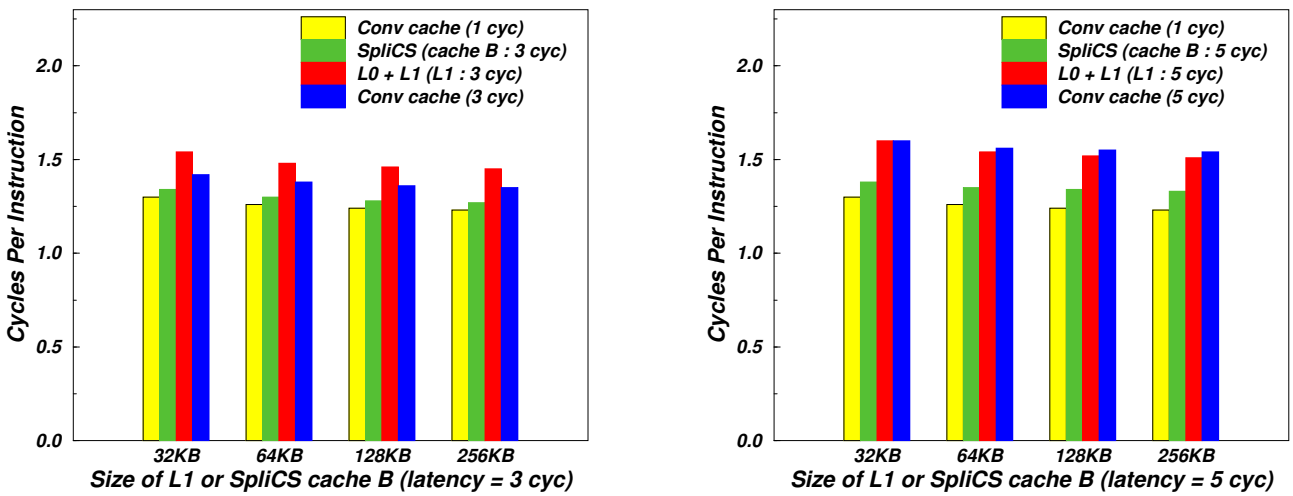
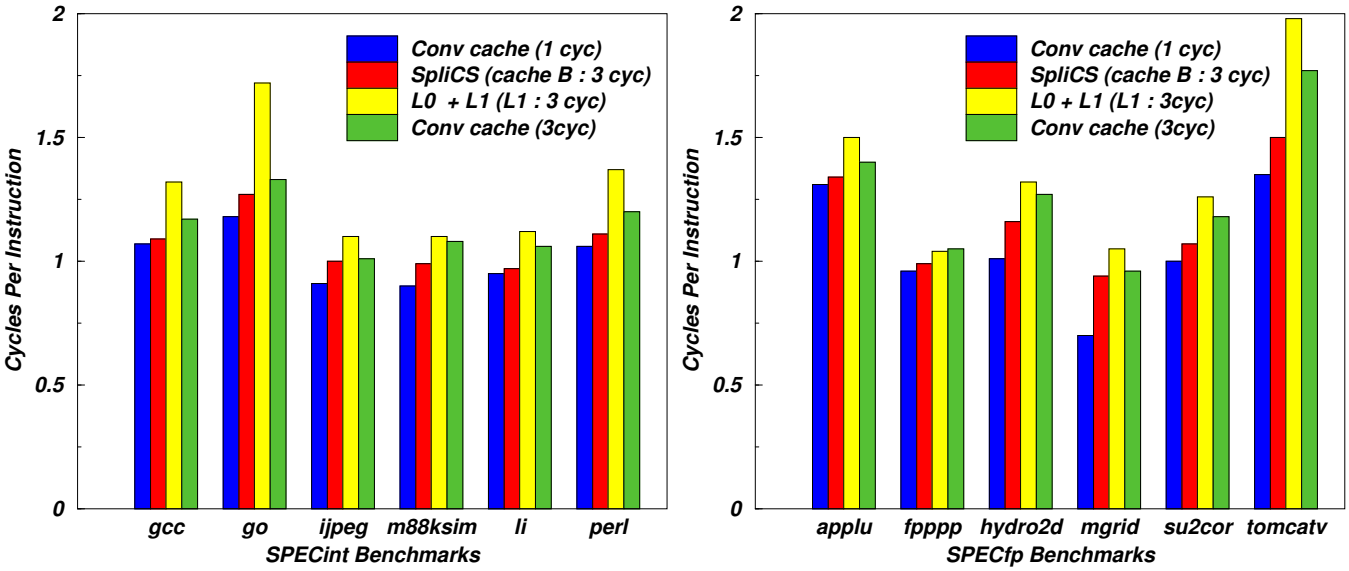


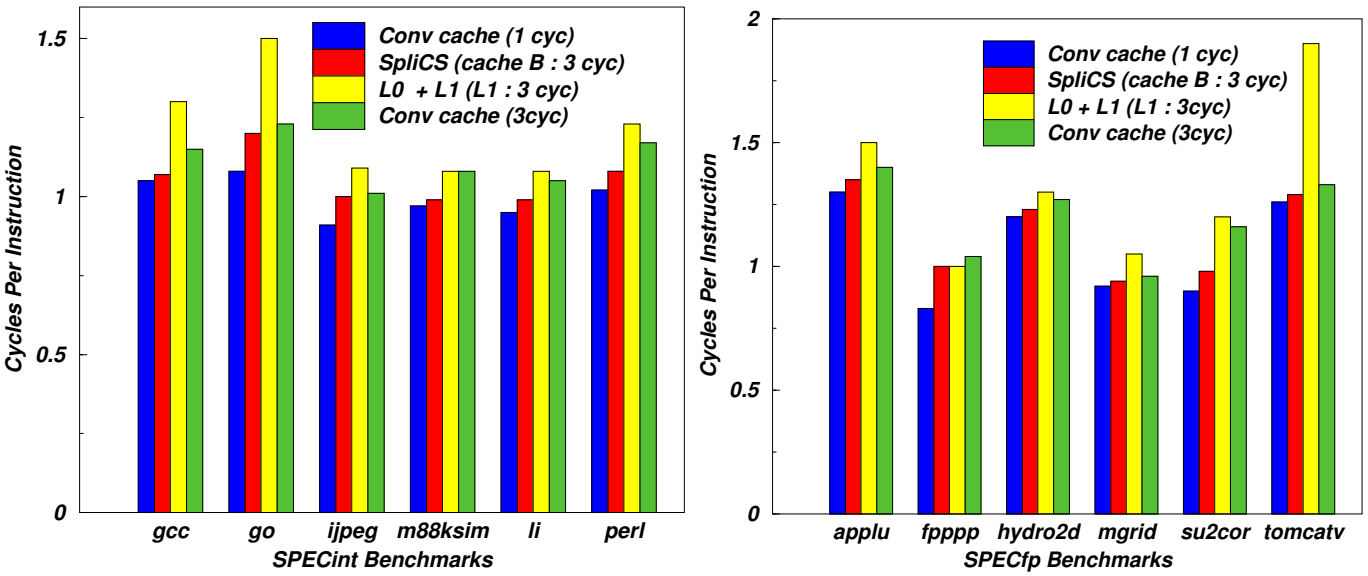
Figure C or D C

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



F igu 1 C or S : a S C p iu 3 C i ac l eg B c

ore importantly, C pli outperforms the L1) system. The features distinguishing C pli from the L1) system and accounting for its superior performance are: (i) The A and caches of C pli are probed in parallel. imultaneous probing of caches A eliminate the need to communicate a miss in cache A to cache. The cache A miss is automatically serviced from the cache 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, (iii) Cache A of C pli does not This is a one of the principal contributor for the performance of C pli . Our results in section .1 show



F igu 1: C er S a S C p iu C i acl eg c B

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

imilarly, we present CPI for all the caches using C B benchmarks in Figures 10 and 11. The trend of the results remain the same for all the C B benchmarks. esults for some of the benchmarks like *go*, *gcc*, and *tomcatv* show that an L0 cache reduces the performance substantially. oreover, the trends in the results remain the same across different cache sizes. Here again, we see that C pli outperforms an (L0 L1) system and achieves a 1 to 18% reduction in CPI than a similarly configured traditional cache.

el ated

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

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

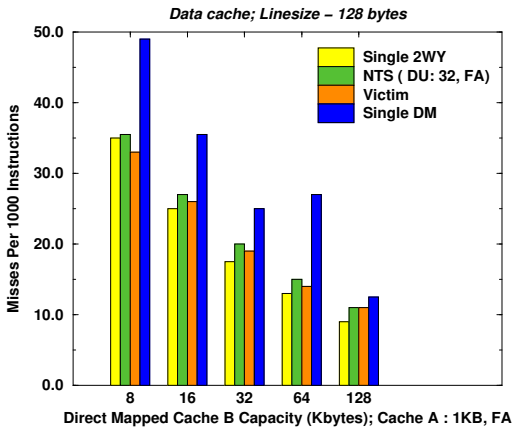
If found in the buffer, they are served to the processor with a very low miss penalty; and are promoted to the primary cache. ince the victim buffer is fully-associative, lines that reside in the victim buffer may be associated with any set of the primary cache. The victim buffer helps reduce con ict misses in hot sets. However, the swaps between the primary cache and the victim buffer can potentially become a bottleneck. A principal difference between C pli and the victim cache is the strict inclusion of cache A in cache . Inclusion avoids costly swaps between cache 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 C pli is to retain the most recently used lines in cache A.

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

The evaluation of the TPCC workload on the victim T 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 T 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 di cult 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- 200, places a conventional primary cache in parallel with a small fully associative assist cache, guaranteeing a one-cycle lookup in bo units. locks 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



F igu 1 rei : pe 1 r i e ruc io or S a c im cac e i

Assist Cache, C pli 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. C pli differs from Assist cache again in the inclusion of cache A in cache . ore importantly, C pli allows different latencies for the two caches, thereby relaxing the size restrictions on the larger cache .

The goal of the above multi-lateral caches (ictim, T , 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 con icts and capacity misses. This in turn improves the overall performance.

C pli 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. ince the contents of cache A are strictly included in cache , C pli does not augment the size of L1 using cache A. Therefore, the performance benefits of C pli comes from the variable access latency and not due to reduction in the L1 miss rate.

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

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

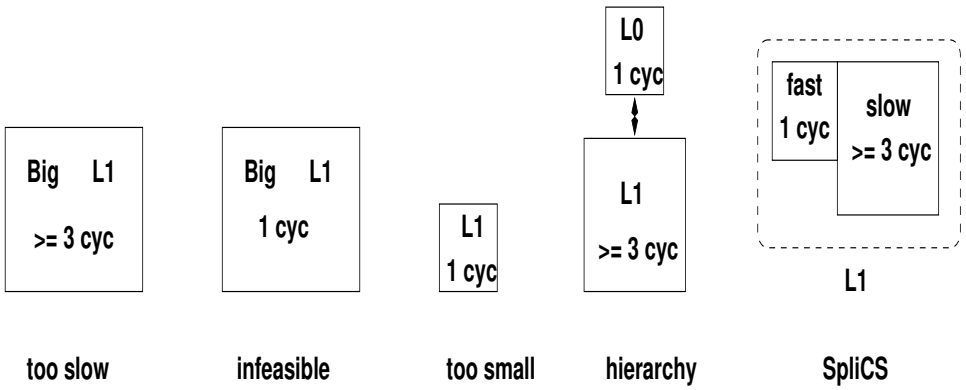
The Fetch Target uffer (FT) presented in [1], is a design similar in spirit to C pli. However, the role of the FT is to improve the branch throughput by having multiple levels of branch target buffers. imilar to the parallel probing of the caches A and C pli, the multiple levels of FT s are probed in parallel to speed-up the access to branch outcomes.

Conc lu sion s

In this paper we introduced a new approach to manage the primary cache and improve its performance. C pli is a design to optimize the performance of a primary cache using a small, fast cache (A) in association with a larger, slower cache ( ). Our results show that, relative to a similarly configured traditional cache, C pli achieves a 12 to 13% improvement in CPI on database workloads and 1 to 18% improvement in CPI on some benchmarks from the C PE 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. C pli 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 C pli from 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. y careful evaluation using a very detailed timer we have demonstrated that these design choices lead to the superior performance of C pliacross different sets of applications.



F igu 1 re 3 e : i De ælor ig 1 e c

C pli can be applied to both instruction and data cache. In addition, C pli 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, sub ect to space limitation on the This. exibility in its design parameters and the better balance it achieves between effective latency and size distinguishes C pli from other proposed designs.

e er enc es

[1] A. Agarwal, . Pudar, Column-Associative Caches: A Technique for educing the iss ateirect apped Caches *Proceedings of the 2 l Symposium on Computer Architecture, May 1 . 3 pp 1 -1*

[2] Arimilli, Clark, L. odson, . . Lewis, Programmable Unified plit echanism for Instructions and ata *IBM Patent AT ov. 1 . 1 8,*

[3] . Charney, T. Puzak, Prefetching and ystem emory ehavior of the CPE benchmark suite, *IBM ournal of esearch and ol 41, ember 3, May 1 .*

[.] Check, T. . Custom egel, /390 5 and *IBM* 5 microprocessors, *esearch and ol 43, ember ber 1 . , Septem*

[5] P. Emma, Understanding ome imple Processor Performance *IBM Journals of esearch and ol 41, ember 3, May 1 .*

[ ] A. C. Alonso, . alera, Cache with A ultiple Caching trategies Tuned to ifferent Types of Locality *Proceedings of International Conference on Supercomputing, 1 . , pp 338 - 34*

[T.] ohnson, W . W . Hwu, Cache Hierarchy Adaptive Management via efer-ence Analysis *Proceedings of the 24th Int l Symposium on Computer Architecture, une 1 . pp 31 -32*

[8] . P. ouppi, Improving Cache Performance by the Addition of a mall Fully-Associative Cache and Prefetch uffers *Proceedings of the 1 l Symposium on Computer Architecture, May 1 . pp 3 4-3 3*

[9] . im, . Lee, Chip Cache Ed On educing emory Access Latency on hared emory ultiprocessors *Proceedings of 11th International Symposium on igh Performance Computing Systems, uly 1 . , pp.*

[10] . . in, upta, W. H. mith, The File Cache: An Energy E ecient tructure *Proceedings of 11th International Symposium on Microarchitecture, ec 1 pp. 184-1 3*

[11] . Chan, . . Lahong, W. E. ryg, PA 200 Processor-I With Integrated High Performance P us Interface *COMPCO, igest of Papers, e 1 . 4, pp. 3 -382*

[12] . Littson, Particles: A ew Type of Cache *IBM Patent SA 8 une.1 8*

[13] ilutinovic, Tomasevic, . arkovi, . Tremblay, ew A Cache Architecture Concept: The Temporal / Cache Partial *Proceedings of the 8th Mediterranean Electrotechnical Conference on Industrial Applications in Power Systems, Computer Science and Telecommunications, vol.2 Bari, May 1 .*

[1] T. Austin, and . Calder, A Fast Front-End Architecture for Fast Instruction elivery *2 , th International Symposium on Computer Architecture, pages 234-24 , May 1*

[15] . A. iversa, Edson, . Con educing gts in irect- apped Caches with a Temporality- ased esign *Proceedings of the 1 International Confere allel Processing, ol I, August 1 . , pp 1 1-1 2*

[1] A. ezase, Case, for Two-Way kewed-Associative Caches *Proceedings of the 2 th Int l Symposium on Computer Architecture, May 1 . 3 pp 1 -1*

[1] A. . Cache ith, emories *Computing Surveys, Sept. 1 82, pp 4 3- 3*

[18] .. o, echtschaffen, Cache Operations by U change *IBM , eport C-11 ov 1 . 8*

[19] . tiliadis, A. arma, Caching: A ethod to Improve the Performance of irect apped Caches *IEEE Transactions On Computers, ol 4 o. , , May 1 pp . 3- 1*

[20] E. Tam, . A. ivers, Tyson, .E. ividson, Evaluating the Performance of Active Cache anagement chemes *Proceedings of the 1 In- 8 IEEE ternational Conference on Computer esign, Oct 1 8*

[21] A. . Weiner, iased Partitioned L Cache eplacement Control *IBM isclosure T B . - p4 , May 1*