

# Exploiting Load Latency Tolerance for Relaxing Cache Design Constraints

Ramu Pyreddy, Gary Tyson

Electrical Engineering and Computer Science,

University of Michigan,

1301 Beal Avenue,

Ann Arbor, MI 48109-2122, USA.

May 10, 2002

## Abstract

Deep pipelines and fast clocks of current day microprocessors make data cache design a challenging task. The first level data cache should be large enough to effectively hide the long memory access latencies and small enough to service memory requests in one processor clock cycle. In this paper we present a multi-lateral cache design exploiting load latency tolerance that addresses this design problem. We present a unique and computation intensive approach to study the latency tolerance of load instructions in programs. This approach provides a more thorough analysis of load latency tolerance than prior studies. We use the information gleaned from this study to classify load instructions as non-critical (latency tolerant) or critical (latency intolerant) to the program performance. The multi-lateral cache structure is a conventional DL1 cache augmented with a small but very fast “critical” cache. Only loads identified as “critical” loads will access both the critical cache and the conventional cache. While non-critical loads will only access the conventional DL1 cache.

**Results:** Our study on the SPECINT 2000 benchmarks shows that a reasonably small number of static loads in a program account for most of the dynamic critical loads during execution. The critical loads account for 12%-24% of the total dynamic loads. A small critical cache of 1KB-2KB can

result in hit rates of greater than 80% (upto 95%) for the critical loads. A critical cache augmenting a slower conventional DL1 cache performs quite comparably with a faster conventional cache of the same size. The performance of a conventional DL1 cache of size  $S$  KB augmented by a small critical cache of 1KB or 2KB is comparable or sometimes even better than the performance of a multi-cycle DL1 cache of size  $2S$  KB.

## 1 Introduction

The increasing frequency gap between main memory and the modern microprocessor is becoming the single most important bottleneck to microprocessor performance. This latency to main memory makes the implementation and efficiency of on-chip caches more important. A processor design solution to this problem has been to increase the size of the first level data cache (DL1). However, a larger cache size implies a longer access latency, so DL1 cache size is limited by the short latency it must have, in order to keep up with the microprocessor core. For example, Pentium III had a DL1 cache size of 16k whereas Pentium 4 has a DL1 cache size of 8KB. As processor pipelines increase in length to keep up with the demand for higher clock frequencies, DL1 caches must remain small to be fast and at the same time must be large enough to hide the long memory latencies. These two conflicting constraints make DL1 cache design a challenging task.

The effect of latency of a load instruction on the program performance is different from other load instructions. While an increase in the latency of certain load instructions has an adverse effect on program performance, increase in the latency of other loads might have a negligible impact or no impact on the program performance. Current microprocessors do not make any distinction between loads based on the effects of their latency on the program performance. All loads are treated equally and memory systems try to maximize the hit rates i.e., satisfy as many memory requests in the DL1 cache as possible. Loads are fetched in program order and executed as quickly as possible. As soon as all the loads source operands are ready, the load is issued to the memory system. All loads access the first level of data cache (usually DL1) and advance through the memory hierarchy until the request is satisfied. Treating all loads equally implies that all target data are contending for space in the memory hierarchy regardless of the effect of the latency of a particular load on the program performance.

Srinivasan and Lebeck [1] have shown that not all loads affect the performance of program equally.

In fact, many have significant latency tolerance. Rakvic et.al [2] presented heuristics to dynamically determine latency tolerant loads. They call these latency tolerance loads as non-vital(not important) loads. Our work proposes a more direct and thorough characterization of latency tolerance of load instructions and a new caching method to take advantage of this load classification.

This work provides two contributions. i) We present a thorough and exhaustive study to determine the latency tolerance of the load instructions that account for 80% of the total dynamic load references. We study the latency tolerance of every load instruction in a program individually. The experimentation methodology and the simulation model will be explained in sections 3 and 4. ii) We present a new cache called critical cache that is accessed only by the loads that are classified as latency intolerant. Latency intolerant loads access both the critical cache and the conventional DL1 cache, whereas the latency tolerant loads access only the conventional DL1 cache. The critical cache is a small structure designed to be fast. Our experiments show that majority of the load instructions are reasonably latency tolerant. A small number of static load instructions in the SPECINT 2000 benchmark suite are responsible for most of the latency intolerant load references. A critical cache of 1KB or 2KB can capture more than 80% of all the latency intolerant load references with our classification scheme. The critical cache reduces the penalty of having a slower DL1 cache instead of a 1 cycle DL1 cache.

## 2 Previous Work

[1] and [3] were the first to identify latency tolerance of loads exhibited by a microprocessor. These works show that loads leading to mis-predicted branches or to a slowing down of the machine are loads that are critical. However [1] used the latency tolerance information for pre-fetching the critical data and better managing the victim cache using the critical data information. They did not design any structures that would primarily exploit the criticality information. One drawback of their approach was that their study was that the criticality criteria was based on heuristics. Since their model required rollback capabilities, they sampled the simulation instead of doing a cycle accurate simulation from start to finish.

[4] introduced a buffer containing non-critical addresses. The non-critical buffer is used to store the latency tolerant loads to improve Data Cache Efficiency.

[2] present a limit study to characterize different types of non-vital loads. They define non-vital loads as loads that can be delayed without affecting performance. They identify non-vital loads as unused loads, loads leading to a correctly predicted branch, loads not instantly used, loads leading to a store and store forwarded loads. This is a dynamic scheme that identifies non-vital loads during the program execution. While there is an inherent advantage of being a dynamic scheme, the hardware structures needed to implement this scheme are not trivial. While these studies are aimed at finding the latency tolerant loads, we are primarily looking at finding the latency intolerant loads.

### 3 Simulation Model

We used SimpleScalar [5] tool kit and the underlying processor pipeline model provided as the cycle accurate simulator. The processor model is very similar to that of an Alpha 21264 processor. The processor configuration is summarized in table 1. The branch predictor is fairly aggressive and so is the RUU size. The fetch engine is capable of fetching 8 instructions in one cycle or a basic block.

The cache configurations simulated are summarized in table 2. The cache configurations are explained in section 5. All the configurations of data cache are 2 ways associative and have a 32byte lines. We used the programs from SPECINT 2000 benchmark suite. We simulated the following eight integer benchmarks - BZIP2 (graphic input), CRAFTY, EON (cook input), GAP, GZIP (graphic input), MCF, PARSER, TWOLF. GCC was not included in the suite due to the large number of load instruction in the program. For all the benchmarks, the reference data set was used and every benchmark was simulated for 500 million instructions.

### 4 Latency Tolerance

Figure 1 shows the performance of the BZIP2 program of the SPECINT 2000 suite as the latency of four of its load instructions is varied. For each load instruction, the latency of the load instruction is varied from 1 cycle to 32 cycles (we sampled the latency tolerance at latencies of 2, 4, 8, 12, 16, 20, 24 and 32 cycles) and the performance of the program as compared to performance with an ideal memory system is plotted<sup>1</sup>. Figure 1 shows that increasing the latency of load instruction for all dynamic

---

<sup>1</sup>Yes! this involved months of simulation time on a large pool of machines lying idle between conference deadlines.

Processor Configuration. (Very similar to an Alpha 21264)	
Fetch Width	8 instructions per cycle
Fetch Queue Size	64
Branch Predictor	2 Level 4K entry level 2
Branch Target Buffer	2K entry 8 way associative
Issue Width	4 instructions per cycle
Decode Width	4 instructions per cycle
RUU size	128
Load/Store Queue Size	32
Inst Cache	64 KB, 2-way, 64-byte lines
L2 Cache	1MB 2-way, 128 byte lines
L2 Latency (to CPU)	16 cycles
Memory Latency	20 cycles

Table 1: Processor Configuration

occurrences of Load1 has no effect on the performance for large latencies(upto 24 cycles). It can also be observed that Load2 is less tolerant than Load1 and Load3 is less tolerant than Load2. Note that the performance of the benchmark is decreased by 10% just by increasing the latency from 1 to 32 for all the dynamic instances of Load4, while satisfying all other memory requests in 1 cycle. Figures 3 and 4 show the load latency tolerance for all the loads that account for 80% of the dynamic load count for two of the benchmarks - BZIP2 and CRAFTY. For lack of space, the load latency tolerance curves for load instructions of other benchmarks are not shown.

This method of determining the latency tolerance of a particular load instruction entails in simulating the benchmark eight times for a particular load instruction, each time with a different load latency value for all the dynamic occurrences of the load instruction. This approach requires hundred of runs to evaluate the latency tolerance of all the loads in a program (sometimes thousands of runs, e.g., GCC). While this brute force approach is computation intensive, it can determine the latency intolerance of a load instruction more accurately than the heuristics proposed in [2] and [1]. One drawback of this approach is the absence of any modeling of interaction between load instructions.

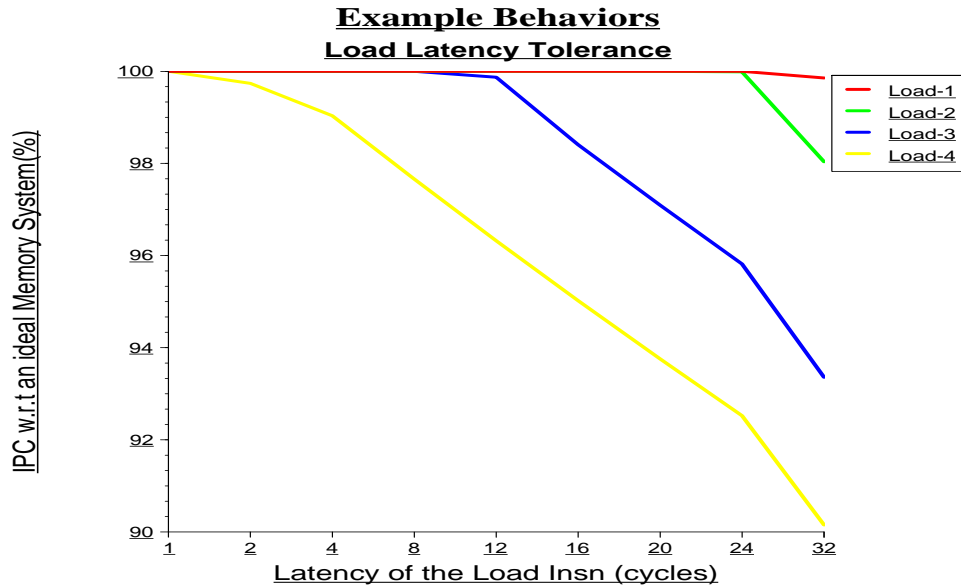


Figure 1: Load latency tolerance behavior of 4 loads from BZIP2 benchmark. Load1 is the most latency tolerant while Load4 is the least latency tolerant.

Since all other loads have a latency of 1 cycle, the effects of the interaction of loads at runtime can not be captured. For example, the latency intolerance of another load instruction could hide the latency intolerance of the load instruction under consideration. While in our study, the load instruction might be classified as a latency intolerant instruction, it might not actually affect the program performance during execution.

Figure 2 shows the region of interest in more detail. To determine the latency tolerance of a load instruction, 2 parameters - latency and a performance threshold are used. If the performance of the program as compared to the performance of the program with an ideal memory system (every load request serviced in 1 cycle) is less than the performance threshold (in percentage) at the given latency for this load instruction, the load is classified as latency intolerant for the particular pair of (performance threshold, latency). Otherwise the load is classified as latency tolerant load instruction. Keep in mind that the list of latency intolerant load instructions changes with the change in the (performance threshold, latency) pair. For example, in figure 2 Load 4 is latency intolerant for the (performance threshold, latency) pair of (99.6, 8cycles) and loads Load1, Load2 and Load3 are latency tolerant by the same measure. However if the latency was 16 cycles in the pair, Load3 would be latency intolerant as well.

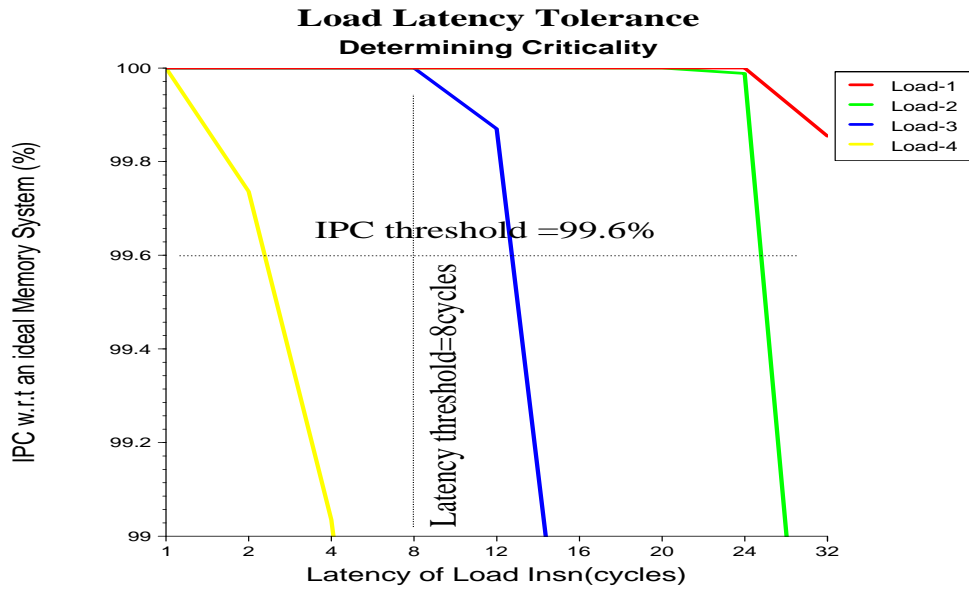


Figure 2: A look at the region of interest from Figure 1 in more detail. A (performance threshold, latency) pair determines the latency tolerance of the load. Load 4 is latency intolerant since the performance of the program is below the performance threshold at the given latency. Note that for the same parameters, loads 1, 2 and 3 are latency tolerant.

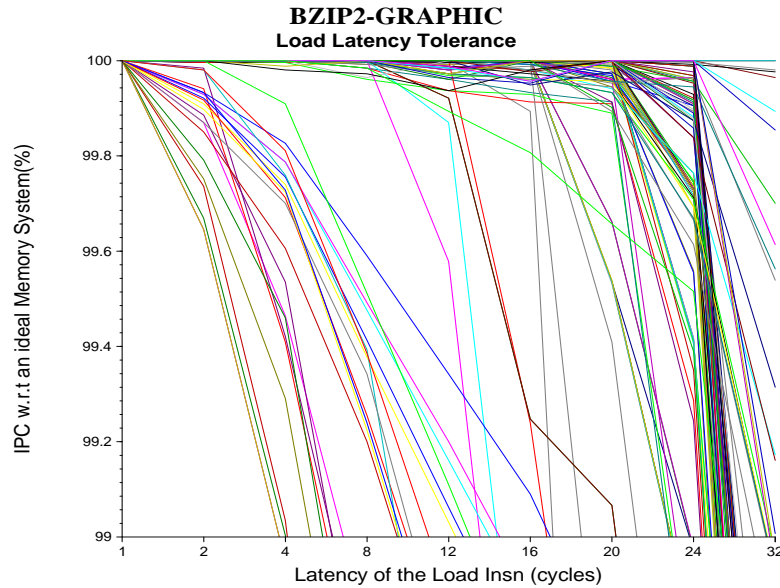


Figure 3: Load latency tolerance behavior of loads for BZIP2 benchmark.

Experimenting with different values for (performance threshold, latency) pairs, we found that the best classification was obtained for most benchmarks at (99.8, 12 cycles). Loads that resulted in a degradation of the performance by more than 0.2% when the latency of the load was increased from 1 to 12 cycles are considered as latency intolerant loads. These loads access the critical cache. The intolerant loads obtained as a result of this classification resulted in better performance gains with a critical cache than classifications with other values for (performance threshold, latency). 0.2% might seem like a small performance degradation for an increase in latency of 12 cycles. However, it should be noted that this degradation is entirely caused due to the dynamic occurrences of only one load instruction. The same degradation for all the loads in the program could cumulatively affect the performance drastically, considering the number of loads in the program.

## 5 Critical Cache

We propose to add a small but fast cache called critical cache to the conventional DL1 cache that will be accessed by the critical loads (latency intolerant) loads. All load references access the conventional cache. However, only the critical loads access the critical cache. All the stores access both the caches and the caches implement a writeback policy. Stores may bring in data that might never be accessed

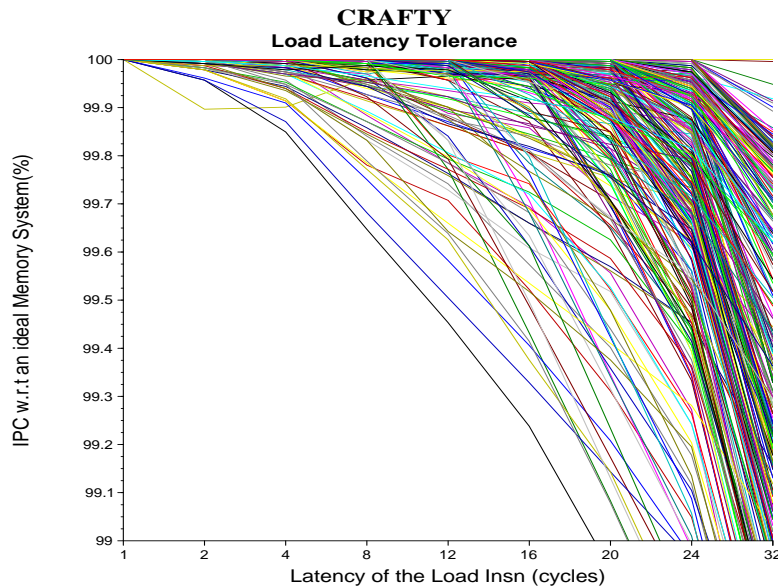


Figure 4: Load latency tolerance behavior of loads for CRAFTY benchmark.

by a critical load into the critical cache. This might cause contention for the data accessed by critical loads in the small critical cache structure. However, we chose this policy as it is simple in guaranteeing data coherency among the cache structures.

Figure 5 shows the schematic of a conventional cache structure augmented with a small critical cache structure. Table 2 summarizes all cache configurations simulated in this study. Since we used critical cache sizes of only 1KB and 2KB, we can safely assume that these caches can be designed to have a one cycle hit latency. So for the experiments in this study, critical caches always have a latency of 1 cycle.

Loads are classified statically using the latency tolerance behavior of load instructions collected (as described in section 4. If the penalty on performance of the program due to a particular load instruction at a latency of 12 cycles, is more than or equal to 0.2%, the load instruction is classified as a critical load instruction. If the penalty is less than 0.2%, the load instruction is classified as non-critical load instruction. Once statically classified, a load instruction remains either critical or non-critical for the complete duration of the program.

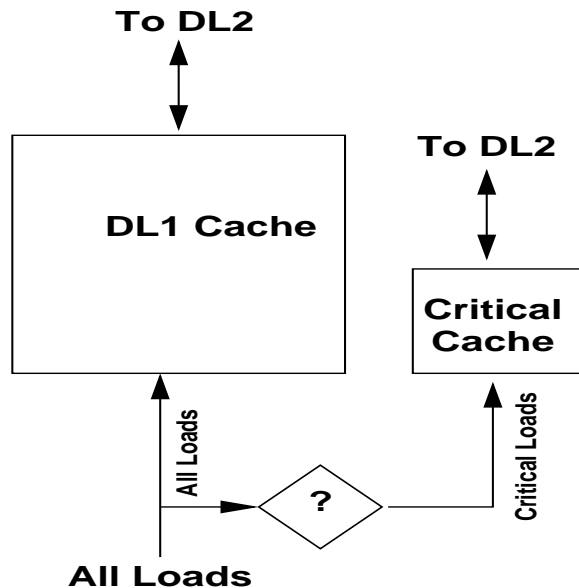


Figure 5: Schematic of the Cache design. Only critical loads access the critical caches. Critical Loads also access the conventional DL1 cache. Criticality information of a load instruction can be either encoded in a bit in the instruction or can be implemented as a table look up at decode time.

## 6 Simulation Results

To show the effectiveness of the critical cache in relaxing the space and cycle time constraints on cache design, we present the following two comparisons. We compare the performance of a DL1 cache of size  $2S$  KB to that of a DL1 cache of size  $S$  KB. We show that the performance penalty due to the reduction in cache size can be offset effectively by augmenting the DL1 cache of  $s$  KB with a small and fast (1 cycle) critical cache of 1 KB.

Figures 6 - 9 illustrate the complexity effectiveness of this cache memory hierarchy. Figures 6 and 7 show that a smaller cache with a critical cache augment performs as good as or better than a cache twice its size. For example, in figure 6 all the IPCs are normalized to that of a 16kdl1-2cycle configuration. For CRAFTY, EON, GAP, MCF and PARSER programs of the benchmark suite, adding a 1K critical cache with 1 cycle access latency will reduce the penalty of a 8K cache as compared to a 16K cache from 20%(CRAFTY) to 80% (GAP). In the case of BZIP2, GZIP and TWOLF programs in the benchmark suite, an 8K DL1 cache augmented with a critical cache will perform better than a 16K DL1 cache. Figure 7 shows the same comparison for a 16K DL1 and

Cache Config. Name	Conventional DL1		Critical Cache	
	Size	Latency	Size	Latency
08kdl1-2cycle	8KB	2	None	-
08kdl1-2ycle+1k-critical	8KB	2	1KB	1
08kdl1-2ycle+2k-critical	8KB	2	2KB	1
16kdl1-1cycle	16KB	1	None	-
16kdl1-2cycle	16KB	2	None	-
16kdl1-2ycle+1k-critical	16KB	2	1KB	1
16kdl1-2ycle+2k-critical	16KB	2	2KB	1
32kdl1-1cycle	32KB	1	None	-
32kdl1-3cycle	32KB	3	None	-
32kdl1-3ycle+1k-critical	32KB	3	1KB	1
32kdl1-3ycle+2k-critical	32KB	3	2KB	1

Table 2: Cache configurations and their nomenclature.

32K DL1 cache. Excepting two benchmarks (CRAFTY and PARSER), a 16K DL1 augmented with a critical cache outperforms a 32k DL1 cache.

The second comparison shows how a critical cache can relax the design constraints on cycle time. Figures 8 and 9 compare the performance of a slower DL1 cache augmented with a small and fast critical cache with that faster DL1 cache of the same size. For example in Figure 8 the performance penalty due to a 2cycle access latency as compared to a 1 cycle access latency for BZIP2 program of the benchmark suite is approximately 3.1%. Adding a 1K critical cache to the slow 16K DL1 cache reduces this penalty to 0.6% (80% reduction in penalty). Figure 9 shows the same comparison for a 32k DL1 cache comparing a slow cahce (3 cycle access latency) and a fast cache (1 cycle access latency).

Column 2 of table 3 shows the number of static loads in the program that account for 80% of all the dynamic load references in the program. Column 3 of shows the the number of these static loads that have been identified as critical using our classification scheme. Column 4 shows the percentage of total dynamic load memory references accounted for by the loads in Column 3. Column 5 shows the

Benchmark	Static Loads accounting for 80% total dyn. loads	Loads identified as Critical	%age of Total Loads issued to Critical Cache	Miss rate for Critical Cache(%)
bzip2	130	23	21.63	9.50
crafty	905	107	16.81	16.34
eon	550	52	17.55	8.40
gap	100	26	12.43	5.72
gzip	95	17	15.48	11.94
mcf	115	32	22.68	9.87
parser	305	33	20.93	12.46
twolf	185	42	15.32	3.53

Table 3: Load Statistics; Coverage and Miss rates for the Critical Cache.

miss rates for these critical loads accessing the critical cache. It can be observed that a small critical cache of 1KB or 2KB can achieve hit rates of at least 80% and upto 96%.

## 7 Conclusions

In this paper, we presented a very direct and thorough classification of loads based on their latency tolerance. We exploit this classification to design a cache design that can service the critical load requests in one processor clock cycle and yet have a large DL1 cache that can minimize the long memory access latencies effectively. We have shown that a S KB DL1 cache augmented with a critical cache can perform as good as (or even outperform in some cases) a 2S KB DL1 cache. We have also shown that the penalty performance due to a multi-cycle DL1 cache when compared to a one cycle DL1 cache can be reduced by almost 80% by augmenting the multi-cycle DL1 with a small and fast critical cache. The small size of the critical size guarantees that design can keep up with fast processor clocks. This small cache can store the critical data whereas the bigger cache can store the non-critical data and can have the luxury of slower clock frequencies since it is primarily servicing the non-critical loads. This allows cache designers to satisfy the one cycle latency required by the processor

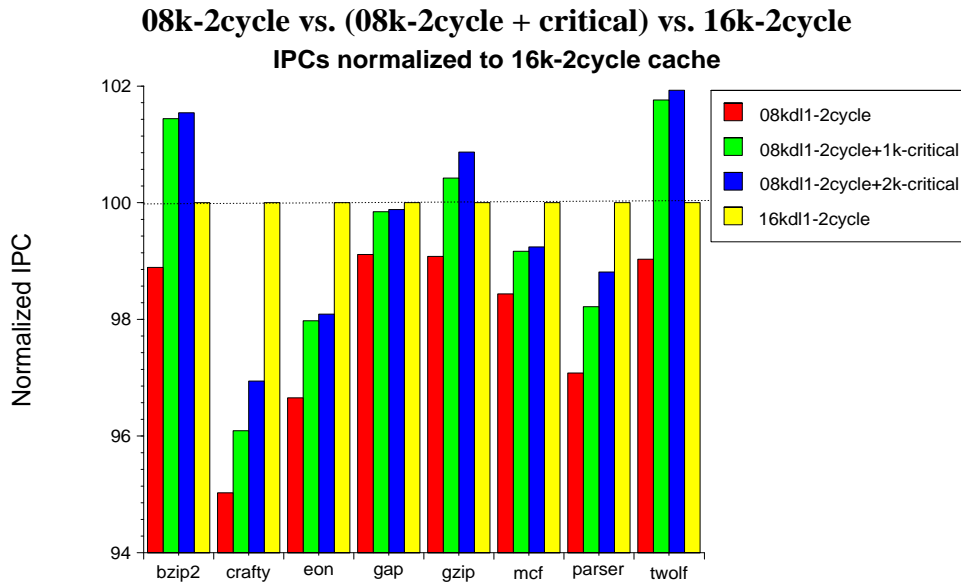


Figure 6: Performance of two cache configurations (08KB and 32KB) compared with the smaller cache configuration augmented with critical cache. A 08KB augmented with critical cache will perform as good as a 16KB cache in some cases and even better than the 16KB cache in some cases.

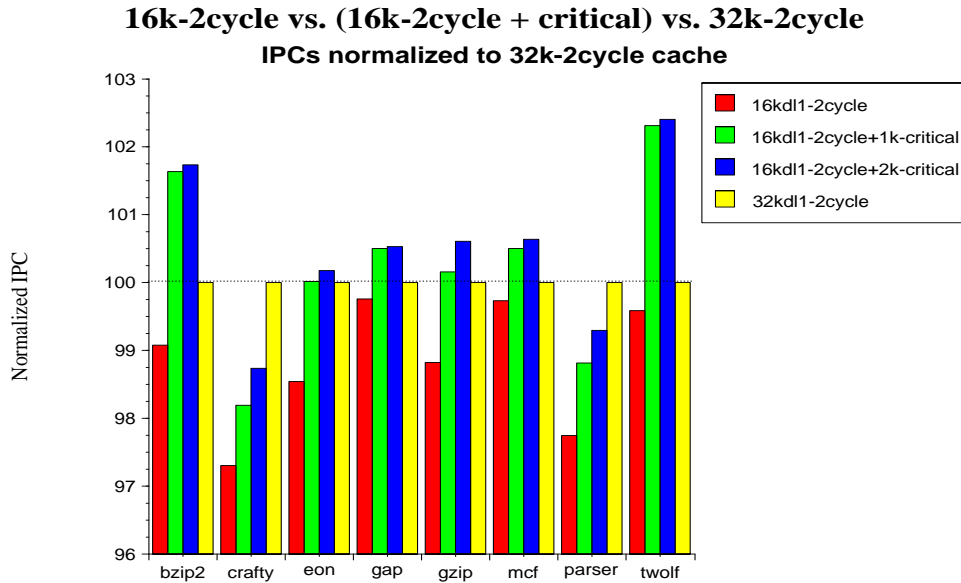


Figure 7: Performance of two cache configurations (16KB and 32KB) compared with the smaller cache configuration augmented with critical cache. A 16KB augmented with critical cache will perform as good as a 32KB cache in some cases and even better than the 32KB cache in some cases.

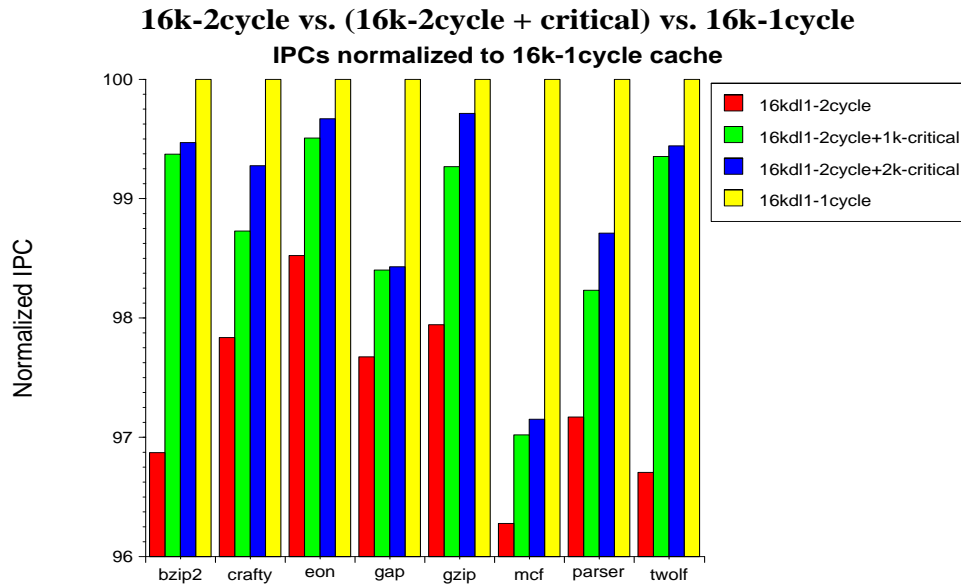


Figure 8: Performance of a 16k multi-cycle (2 cycle) cache compared to a 16k 1 cycle cache with the slower cache augmented with critical cache. A 16KB with 2cycle hit latency augmented with a 2k critical cache will recover upto 80% of the penalty due to the slower clock.

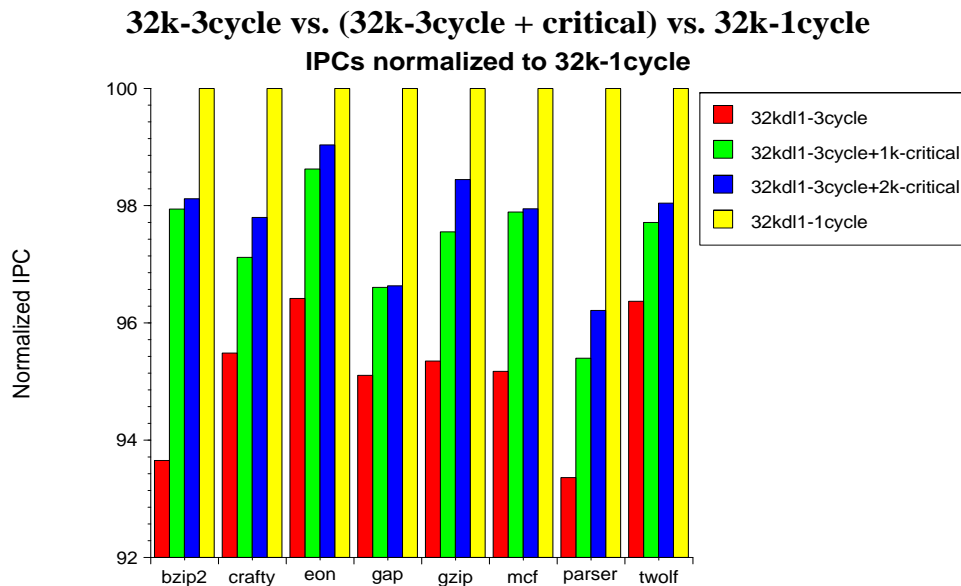


Figure 9: Performance of a 32KB multi-cycle (3 cycle) cache compared to a 32KB 1 cycle cache with the slower cache augmented with critical cache. A 32KB with 2cycle hit latency augmented with a 2k critical cache will recover upto 60% of the penalty due to the slower clock.

## References

- [1] S. T. Srinivasan and A. R. Lebeck, “Load latency tolerance in dynamically scheduled processors,” in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, (Los Alamitos), pp. 148–159, IEEE Computer Society, Nov. 30–Dec. 2 1998.
- [2] R. Rakvic, B. Black, D. Limaye, and J. Shen, “Non-vital loads,” in *International Symposium on High-Performance Computer Architecture*, Feb. 2002.
- [3] S. T. Srinivasan, R. D.-C. Ju, A. R. Lebeck, and C. Wilkerson, “Locality vs. criticality,” in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, (Göteborg, Sweden), pp. 132–143, IEEE Computer Society and ACM SIGARCH, June 30–July 4, 2001.
- [4] B. Fisk and R. Bahar, “The non-critical buffer: Using load latency tolerance to improve data cache efficiency,” in *International Conference on Computer Design (ICCD '99)*, (Washington - Brussels - Tokyo), pp. 538–545, IEEE, Oct. 1999.
- [5] D. Burger and T. M. Austin, “The simplescalar tool set, version 2.0,” *Technical Report TR 1342*, University of Wisconsin, June 1997.