Non-vital Loads

†Ryan Rakvic, †Bryan Black, ‡Deepak Limaye, & †John P. Shen

†*Microprocessor Research Lab*
Intel Labs
{ryan.n.rakvic,bryan.black,john.shen}@intel.com

‡*Electrical and Computer Engineering*
Carnegie Mellon University
dlimaye@ece.cmu.edu

Abstract

As the frequency gap between main memory and modern microprocessor grows, the implementation and efficiency of on-chip caches become more important. The growing latency to memory is motivating new research into load instruction behavior and selective data caching. This work investigates the classification of load instruction behavior. A new load classification method is proposed that classifies loads into those vital to performance and those not vital to performance. A limit study is presented to characterize different types of non-vital loads and to quantify the percentage of loads that are non-vital. Finally, a realistic implementation of the non-vital load classification method is presented and a new cache structure called the Vital Cache is proposed to take advantage of non-vital loads. The Vital Cache caches data for vital loads only, deferring non-vital loads to slower caches.

Results: *The limit study shows 75% of all loads are non-vital with only 35% of the accessed data space being vital for caching. The Vital Cache improves the efficiency of the cache hierarchy and the hit rate for vital loads. The Vital Cache increases performance by 17%.*

1   Introduction

The latency to main memory is quickly becoming the single most significant bottleneck to microprocessor performance. In response to long-latency memory, on-chip cache hierarchies are becoming very large. However, the first-level data cache (DL1) is limited in size by the short latency it must have to keep up with the microprocessor core. For an on-chip cache to continue as an effective mechanism to counter long latency memory, DL1 caches must remain small, fast and become more storage efficient.

A key problem is that microprocessors treat all load instructions equally. They are fetched in program order and executed as quickly as possible. As soon as all load source operands are valid, loads are issued to load functional units for immediate execution. All loads access the first level of data cache and advance through the memory hierarchy until the desired data is found. Treating all loads equally implies that all target data are vying for positions in each level of the memory hierarchy regardless of the importance (vitality) of that data.

As demonstrated by Srinivasan and Lebeck [22] not all loads are equally important. In fact, many have significant tolerance for execution latency. Our work proposes a new classification of load instructions and a new caching method to take advantage of this load classification. We argue that load instructions should not be treated equally because many loads need not be executed as quickly as possible.

This work presents two contributions. 1) We perform a limit study analyzing the classification of load instructions as vital (important) or non-vital (not important). Vital loads are loads that must be executed as quickly as possible in order to avoid performance degradation. Non-vital loads are loads that can be delayed without impacting performance. 2) We introduce a new cache called the Vital Cache to selectively cache data only for vital loads. The vital cache improves performance by increasing the efficiency of the fastest cache in the hierarchy. The hit rate for vital loads is increased at the expense of non-vital loads, which can tolerate longer access latencies without impacting performance. Performance is also increased by processing (scheduling) the vital loads ahead of non-vital loads.

2   Previous Work

In [1] the predictability of load latencies is addressed. [15] showed some effects of memory latencies, but it was [21][22] to first identify the latency tolerance of loads exhibited by a microprocessor. These works show that loads

leading to mispredicted branches or to a slowing down of the machine are loads that are critical. This work is built on the same concept as [21][22]. In fact, part of our classification (lead to branch, see Section 4) is taken from this previous research. This work further identifies additional classes of loads and uses a different classification algorithm. Furthermore, we introduce a new caching mechanism to take advantage of them.

The work in [21] introduced an implementation based on the non-critical aspect of loads. They implemented two different approaches: using a victim critical cache, and prefetching critical data. Neither seemed to show much performance benefit. The work in [7] also introduced a buffer containing non-critical addresses. The implementation in Section 5 is based on the same spirit of [7][21], but is done in accordance with non-vital loads.

Section 5 introduces a form of selective vital caching. This selective caching is similar in concept to [9][10][14][19][25]. The goal of selective caching is to improve the efficiency of the cache. [9][10] cached data based on temporal reuse. [25] selectively cached data based on the address of loads. In particular, loads which typically hit the cache are given priority to use the cache. We also propose caching data based on the address of loads. However, we cache data based on the vitality or importance of the load instruction.

The non-vital concept should not be confused with "critical path" [24] research. Non-vital loads may or may not be on the critical path of execution. Non-vital loads become non-vital based on resource constraints and limitations. Therefore, a load that is considered "non-vital" may be on the critical path, but its execution latency is not vital to overall performance. [6] introduced a new insightful critical path model that takes into account resource constraints. [6] used a token-passing method to try to identify instructions that are critical to performance. On the other hand, our approach attempts to identify the loads that are not critical to performance and therefore do not need DL1 cache hits to maintain high performance.

Other popular research tries to design a DL1 that maintains a high hit rate with very low latency[8]. One approach used streaming buffers, victim caches [13], alternative cache indexing schemes [20], etc. [10]. Another approach attempts to achieve free associativity. Calder et al. [4], following the spirit of [12][11][2][17], proposed the predictive sequential associative cache (PSA cache) to implement associative caches with a serial lookup. The aim of their work was to reduce the miss rate of direct mapped caches by providing associativity in such a way that the cache access latency was the same as a direct mapped cache. They achieved in-creased hit rate and thereby performance through a sequential lookup of a direct mapped cache.

[3][5][11][23][26] focused on providing bandwidth and speed by distributing the cache accesses. Cho et al. [5] provided high-speed caches by decoupling accesses of local variables. Neefs et al. [16] also provided high bandwidth high speed cache memories. Their solution utilized a general multi-banked scheme interleaved based on data address. [18] provided high bandwidth and high speed by partitioning the DL1 into subcaches based on temporality use. Our implementation (Section 5) is similar in spirit, but is built around the non-vital concept.

3   Simulation Model

A cycle accurate simulator is used to gather all the data presented. The machine model employs eight instruction wide fetch, dispatch, and completion. The instruction window size is 128 instructions and the Load-Store Queue has 64 entries. The first level data cache (DL1) has 32KB and is a dual-ported 4-way set-associative cache with 32-byte blocks and 3-cycle latency. The second level unified cache is 256KB, with a 10 cycle access latency. There is a 32-entry store buffer, and the ports are assumed to be read/write. There are ten functional units that execute instructions out-of-order. A hybrid branch predictor using both a bimodal and a 2-level predictor is used. The instruction cache is set at 32KB and is also single cycle. The fetch unit is capable of fetching eight instructions or one basic block per cycle. We propose this as a relatively realistic machine.

Seven integer (compress, gcc, go, ijpeg, li, m88ksim, perl) and seven floating point (applu, apsi, fpppp, mgrid, swim, tomcatv, wave5) benchmarks from the Spec CPU95 suite are used for this study. Unless otherwise noted, numbers presented represent harmonic means of all benchmarks.

4   Non-vital Loads

This section examines the classification of non-vital loads. A classification method is presented along with a limit study that shows how many loads are non-vital. Section 5 outlines an implementation that leverages the non-vital load classification to improve IPC by increasing the efficiency of the data cache.

4.1   Classification

Loads are classified into two categories: vital and non-vital. Vital loads are loads that must be executed as quickly as possible in order to maximize performance. Non-vital loads are loads that are less vital to program execution and for various reasons do not require immediate execution in order to maximize performance. Figure 1 illustrates the classification of load instructions into vital or non-vital. We

Non-vital	Vital
Unused	
Store forwarded	
Lead to store	
Not instantly used	Instantly used
Lead to correctly predicted branch	Lead to mispredicted branch

Figure 1 Classification of loads as Non-vital and Vital.

do note that the non-vital loads are not of equal vitality, but we do not present an ordering.

Unused: A load is classified as non-vital if its result is not used before the destination register is redefined. A load can go unused when the compiler hoists it above a branch that does not branch to the basic block that consumes the load's result.

Store forwarded: Another type of non-vital loads is a load that receives its data from the store buffer via store forwarding. Such a load is classified as non-vital because the data access does not reach the cache hierarchy. Hence, the load's data value need not reside in the first level cache.

Lead to store: Store instructions are by nature non-vital. As a result any data flow leading to a store's address or data source is also non-vital. When all dependents of a load ultimately lead to a store and the data flow terminates at the store, the load can be classified as non-vital.

Instantly Used: Instructions dependent on a load's result may be in the instruction window when the load finishes

execution. However, other resource or data hazards can cause the dependent instruction to defer execution for some time. If a load's dependent is immediately ready for execution when the load finishes, the load is considered vital, due to the critical demand of the dependent. If the dependent instruction is not ready for execution as the load finishes then the load is non-vital.

Lead to branch: As demonstrated in [22], loads that lead to predictable branches are non-vital. Every data flow path beginning with the load must terminate at a predictable branch for the load to be considered non-vital. If a branch is not predicted correctly or not all paths lead to a predictable branch then the load is classified as vital.

4.2 Limit Study of Non-vital Loads

This section presents the results of a limit study on the classification of non-vital loads. Realistic hardware implementation for the load classification, outlined in Section 4.1, is described in Section 5. To properly classify non-vital loads, as per the classification in Section 4.1, a complete
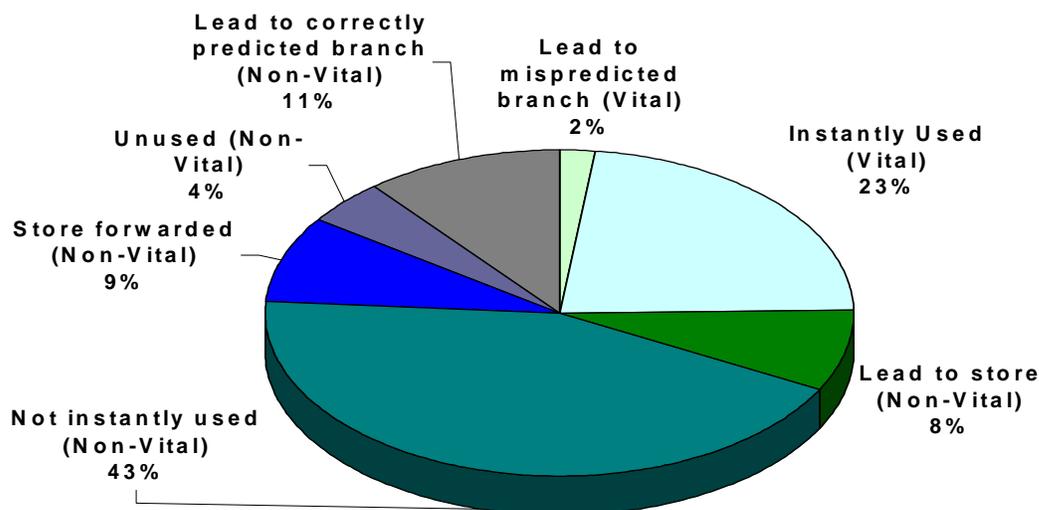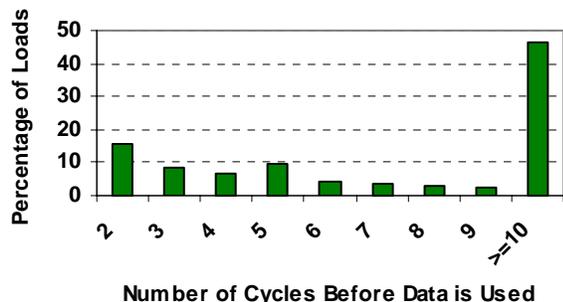

Figure 2 Load classification.

Figure 3 Dependence distance from loads.

dependency chain beginning with each load is constructed. Dependency chains are required to determine if a load leads to a store or branch. Each dependency chain begins with a load and is constructed and maintained only while the load is in the machine. Once the load completes it is classified as either vital or non-vital, and the dependency chain is discarded.

The pie chart in Figure 2 shows a complete break down of all loads and their classification as vital or non-vital. Astoundingly only 25% of loads in the SPEC95 benchmarks are vital. On the vital loads, 2% are leading to mis-predicted branches and 23% are vital due to instant demand from dependent instructions. The remaining 75% of loads are classified as non-vital. Of the five categories of non-vital loads, loads that lead to not-instantly-used instructions make up 43% of all loads. 8% of loads ultimately lead to stores, while 9% receive their data from the store buffer via store forwarding. Another 11% of loads lead to a correctly predicted branch and 4% of loads are unused.

To further analyze the 43% of loads with not instantly using dependents, Figure 3 illustrates the number of cycles between a load's execution and its dependent's execution. If the dependent executes immediately, the load is considered vital. If the dependent instruction does not execute immediately after the load, then the load is classified as non-vital. Figure 3 shows that almost 50% of the time, a load's dependent instruction executes 10 or more cycles after the load. The >=10 bar lumps all cases in which the dependent instruction did not execute within 10 cycles of the load's execution. If the dependent instruction is not in the machine at the time of load completion its execution is considered more than 10 cycles after the load. One cause of this delay is when a load precedes a mispredicted branch but the dependent instruction is following the mispredicted branch, and is stalled by the misprediction recovery.

4.3 Verification of Non-vital Load Classification

To verify the classification results in Section 4.2, the benchmarks are executed with doubling the execution latency of all loads classified as non-vital. If there is little or no performance impact then these loads are truly non-vital, otherwise they are incorrectly classified. Performance loss from increasing non-vital loads' latency can be due to resource constraints. For example, if a non-vital load's latency is increased, it may cause the Re-order Buffer (ROB) to fill up. The ROB fill-up may in turn cause instruction fetch to stall and overall performance reduction. Similarly, functional unit unavailability can also cause performance degradation. Vital loads, however suffer performance loss due to various reasons. For example, vital loads that lead to a mispredicted branch cause instruction fetch to continue down the wrong path for extra cycles.
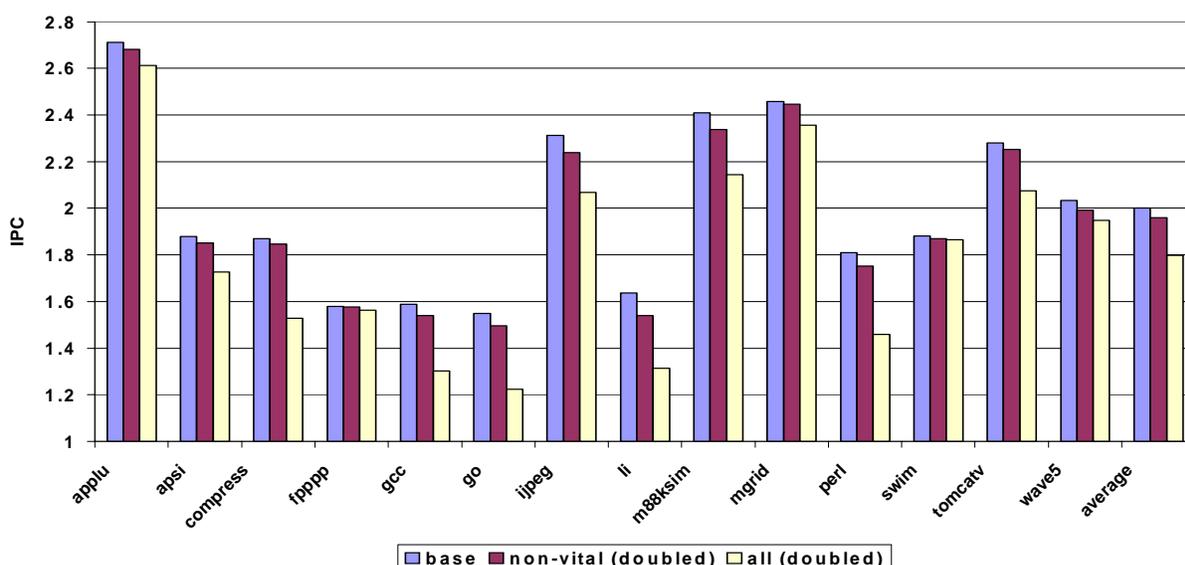

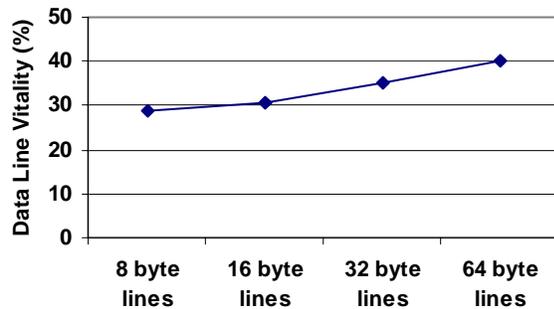**Figure 4 Performance impact of load access latency.**

Figure 5 Data vitality relative to cache line sizes.

Figure 4 presents IPC results comparing a baseline model (base) with a model that doubles the DL1 latency for all loads (all) and a model that doubles the DL1 latency only for loads classified as non-vital. Across all benchmarks, there is little performance degradation (~2%) when the non-vital load access latency is doubled. However, an average performance loss of 10% is observed when the latency of all loads are doubled.

When considering new caching mechanisms based on load vitality it is not enough to consider just execution latency. For instance, if a vital load accesses a data line, then the line is considered vital and the data must be stored in the fastest cache. This data vitality is vital to any implementation that is designed to take advantage of non-vital loads. If all data are vital then load classification is uninteresting. Figure 5 presents the percentage of unique data that are vital as a function of cache line size. As expected, as the line size increases, the probability that a vital load will access a data element within the line increases. If any vital load accesses a data line at any point within the program execution, the line is considered and remains vital; hence as the line size increases, the data vitality increases. The data vitality ranges from 28%-40% as the line size is varied from 8 bytes to 64 bytes. For a 32-byte cache line, the average data vitality is only ~35%.

This limit study shows that 75% of all loads in the SPEC95 benchmarks are non-vital and only 35% of the accessed data space is vital. The next section outlines a new caching strategy that takes advantage of non-vital loads and data vitality to improve performance.

5 Vital Cache - An Application of Non-vital Loads

There is much pressure on the first level cache (DL1) of the memory hierarchy in today's microprocessors. Section 4.3 (data vitality) shows that not all data that are brought into the DL1 are vital to processor performance. There is the po-

tential of taking advantage of non-vital loads. One application is to redesign the data cache hierarchy. This section introduces the Vital Cache (VC). The vital cache is a realistic cache implementation that takes advantage of non-vital loads. Vital cache caches only data of vital loads. Section 5.1 describes the implementation while Section 5.2 shows initial results and their analysis.

5.1 Implementation

As discussed in Section 4.1 there are five types of non-vital loads: unused, lead to store, lead to correctly predicted branch, store forwarded, and not instantly used. Some of these types are more difficult to determine than others. In order to maintain a realistic implementation, this section will focus only on the most dominant and easiest to identify non-vital loads, the not instantly used.

We present a scheme that 1) identifies which loads are not instantly used, 2) stores this information on a per load basis, and 3) uses this information to process the load instructions. We highlight three structures needed for this scheme: 1) vitality classification via rename register file; 2) vitality storage via instruction cache; and 3) the vital cache. The following sections describe these in detail.

5.1.1 Vitality Classification via Rename Register File

As stated, we only need to identify the case when the load's value is not used immediately following its execution. We only identify the not-instantly-used non-vital loads because they constitute the largest portion of non-vital loads, and are easy to identify (the other non-vital loads can also be identified, but are left for future work). The rename register file (RRF) can collect this information (see Figure 6). Two extra bits are added to each renamed register. These bits are originally in an initial state (00). They are set to a new state (01) upon a write by a load to the renamed register. If the register is not read in the next cycle, then they go to a final state of non-vital (10). Otherwise, they go to a final state of vital (11). After identification, the vitality information is used to update the instruction cache for each static load. Every dynamic occurrence of a load causes it to be re-identified.
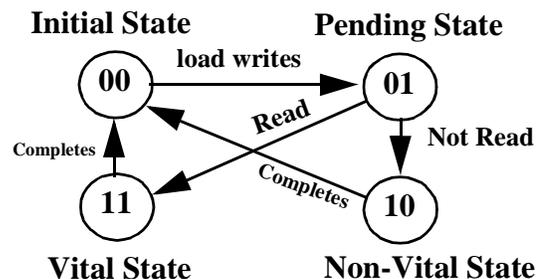

Figure 6 State machine for vitality classification.

5.1.2 Vitality Storage via Instruction Cache

The instruction cache is used to store the vitality information for each static load. Each load has only one bit that represents if it is vital or not. This added bit should not impact fetch latency, and can be attached to the instruction cache or kept in a separate table. The separate table can be accessed after decode, and indexed by the instruction pointer of the load. This bit is sent with the load to the memory hierarchy. It determines the updating of the data caches.

5.1.3 Vital Cache

The heart of our non-vital load implementation is the vital cache (VC) in the memory hierarchy (see Figure 7). We propose adding a level of cache above the DL1. This cache (DL0) should be a small cache that can be clocked in one cycle. The vital cache is only allocated/updated by vital loads; whereas the DL0 will update the cache for every load on a miss (see Figure 7). The goal is to have the non-vital loads retrieve their data from the DL1, while the vital loads get their data faster from the VC. A VC configuration also provides a port advantage by prioritizing vital loads access. For example, if a vital load is ready to execute in a given cycle, the vital load is given the available VC port (the DL0/VC is assumed to be write-through and single ported, and the DL1 is assumed to have 2 overall ports for both configurations). Prioritizing (separating) vital loads provides another performance advantage for the VC scheme.

5.2 Results

As stated in Section 3, we assume that the 32KB cache has multi-cycle latency. Therefore, we will show that it is advantageous to add a smaller, faster data cache (DL0) to the memory hierarchy. Instead of treating all loads equally like the DL0, the VC gives priority to vital loads. We begin by comparing the overall performance impact that a non-vital load implementation can have. Figure 8 shows IPC results for each individual benchmark for three different configurations: 1) DL1 with 3 cycle latency (DL1); 2) DL1 and a 256B DL0 (DL1+DL0); and 3) DL1 and 256B vital cache (DL1+VC). As graphed, adding a small (256B) single-cycle cache to the hierarchy (DL1+DL0) adds performance for almost all the benchmarks, averaging a 4% increase in overall performance. But by using a vital cache configuration, we gain an additional 12% compared to the DL0 configuration. The IPC performance gain is noticeably higher for the floating point benchmarks. The vital cache does present a performance gain for all the benchmarks, and averages a 17% increase over the configuration with just the DL1.

The performance gain is derived from an increase in the efficiency of the highest level cache (DL0 or VC) and by prioritizing the processing of vital loads (allocating the VC port to vital loads). However, increasing the latency of non-vital loads by pushing their access time to that of the slower DL1 may also have a negative effect on performance (see Figure 4). If the ROB (or other resource constraints) is strained by the increased load latency, then the machine may stall, and overall performance is reduced. But the IPC results in show that this negative effect of the vital cache is more than compensated for by the increase in priority for the vital loads.

The percentage of loads (non-vital) that do not update the vital cache is high and is shown in Figure 9. Recall: 43% of
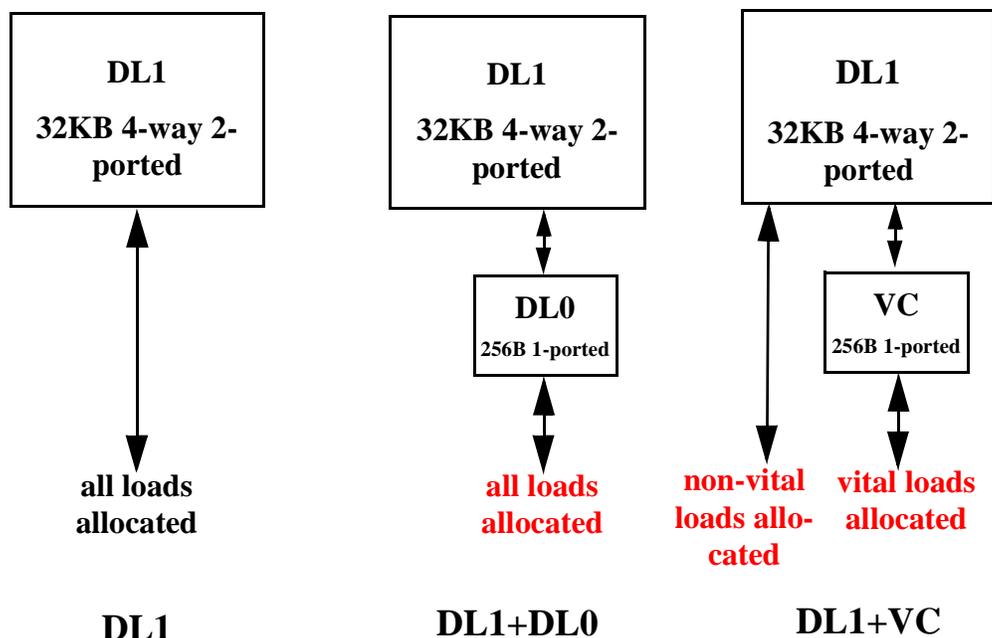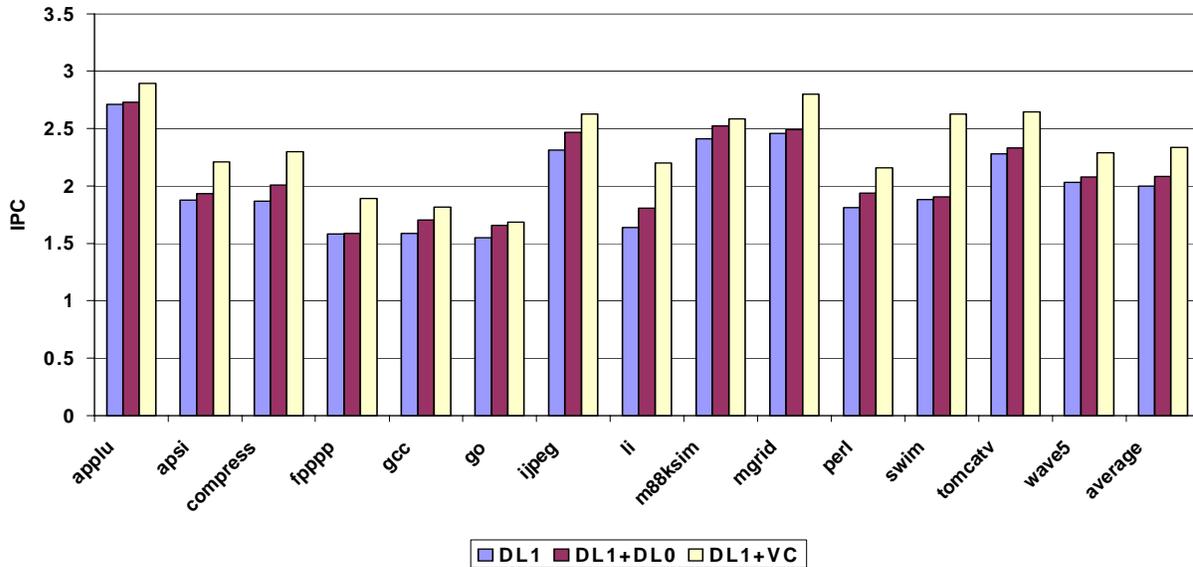

Figure 7 Different cache configurations.

Figure 8 Implementation performance comparison.

all loads are classified as not instantly used. But this number changes as the cache hierarchy management changes. For example, when a non-vital load's data does not move to the vital cache, the load (with higher latency) may actually become a vital load. The interactions and changing of vital/non-vital loads is outside the scope of this paper and left for future work. For some of the benchmarks there is correlation be-

tween this percentage (loads not updated), and the overall IPC performance impact that is shown in Figure 6. For example, mgrid has a high percentage of loads which do not update the vital cache (in theory making it more efficient), and the IPC's positive impact reflects this.

Hardware identification of a load's vitality is done via prediction and is described in Section 5.1. The scheme pre-
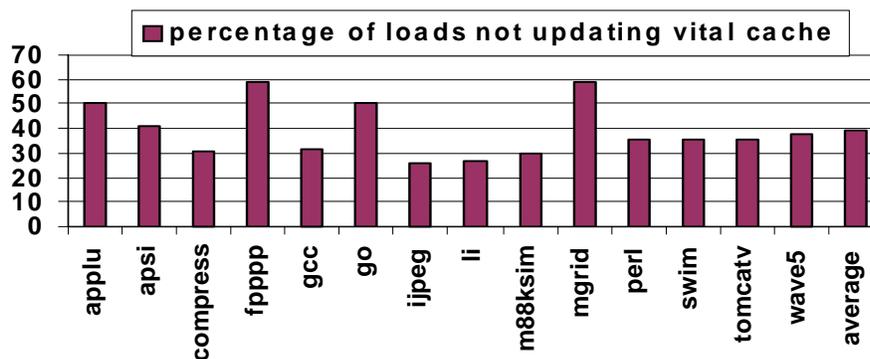

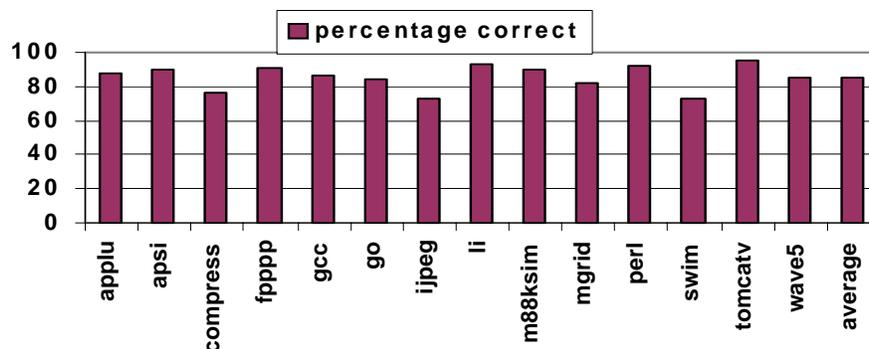**Figure 9 Loads that do not update the vital cache.**

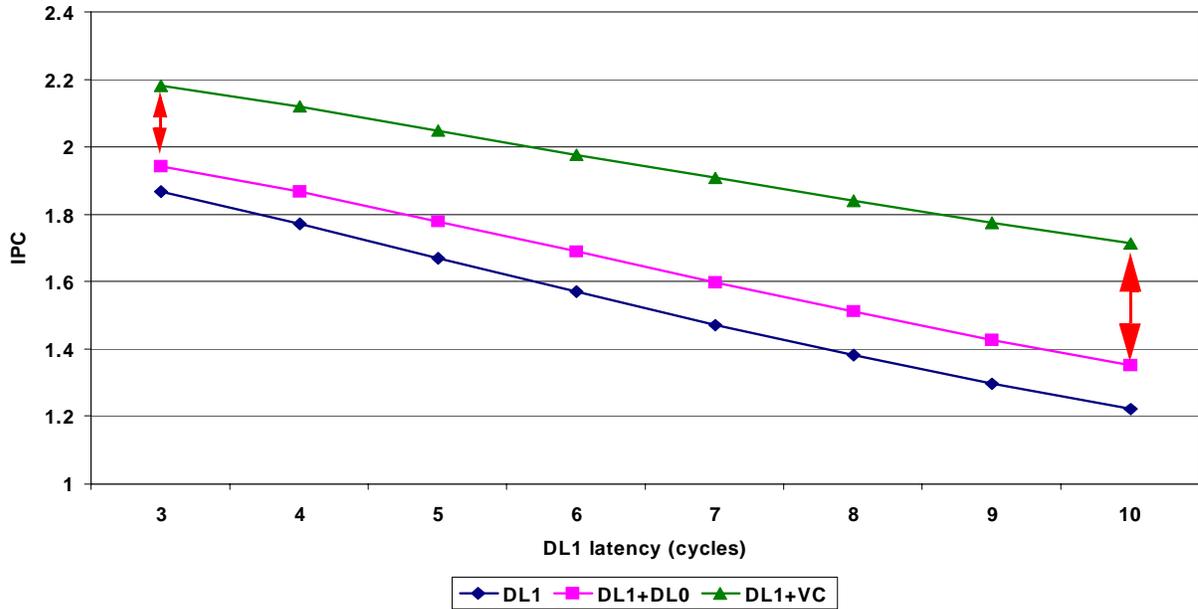**Figure 10 Vitality predictability.**

Figure 11 DL1 latency analysis.

sented is a simple initial scheme, and implements a last-value prediction algorithm, i.e. if a static load is identified as non-vital the last time it executed, then we will predict that it will be non-vital the next time it executes. A simple last-value prediction mechanism is accurate and the results are illustrated in Figure 10. Figure 10 graphs the results for the prediction of the not-instantly-used loads. A misprediction is counted if the load is fluctuating between vital and non-vital. The vitality of this class of loads is predictable, with the mean prediction accuracy above 85%.

The 32KB 4-way associative 2-ported cache that acts as the DL1 is assumed to be 3 cycles in all simulation results so far. As microprocessor clock frequency increases, the latency of DL1 will also increase. We now vary the DL1 from 3 cycles to 10 cycles. Figure 11 presents harmonic means of the IPC for the three configurations again. As the DL1's latency becomes greater, the impact of the vital cache becomes greater (slope of the vital cache configuration is not as steep as the other configurations). Assuming 10-cycle DL1 latency, a VC configuration outperforms the

DL0 configuration by 27% and a DL1-only configuration by 40%.

Preceding simulation results had the DL0/VC fixed at 256B. We now present simulation results varying the size of this highest-level cache from 256B to 8KB (assuming single cycle DL0/VC for all cases). Figure 13 compares the performance between a DL0 and VC configuration for three variations representing different DL1 and DL2 latencies: see Figure 12. For example, for medium-DL0, the latencies for the data caches are 20, 6, and 1 cycle(s) for the DL2, DL1, and DL0 respectively. These variations (fast, medium, slow) represent cache latency scaling with future anticipated microprocessor clock frequencies (DL2 and DL1 are scaled by the same factor). As the size of the DL0 is increased, the performance difference between the VC configuration and a normal DL0 configuration is reduced. The efficiency of the VC becomes less of a factor as the size of the cache is increased. However, for the three different schemes, a 1KB VC consistently outperforms an 8KB DL0.

	DL1 Latency	DL2 Latency
fast	3	10
medium	6	20
slow	9	30

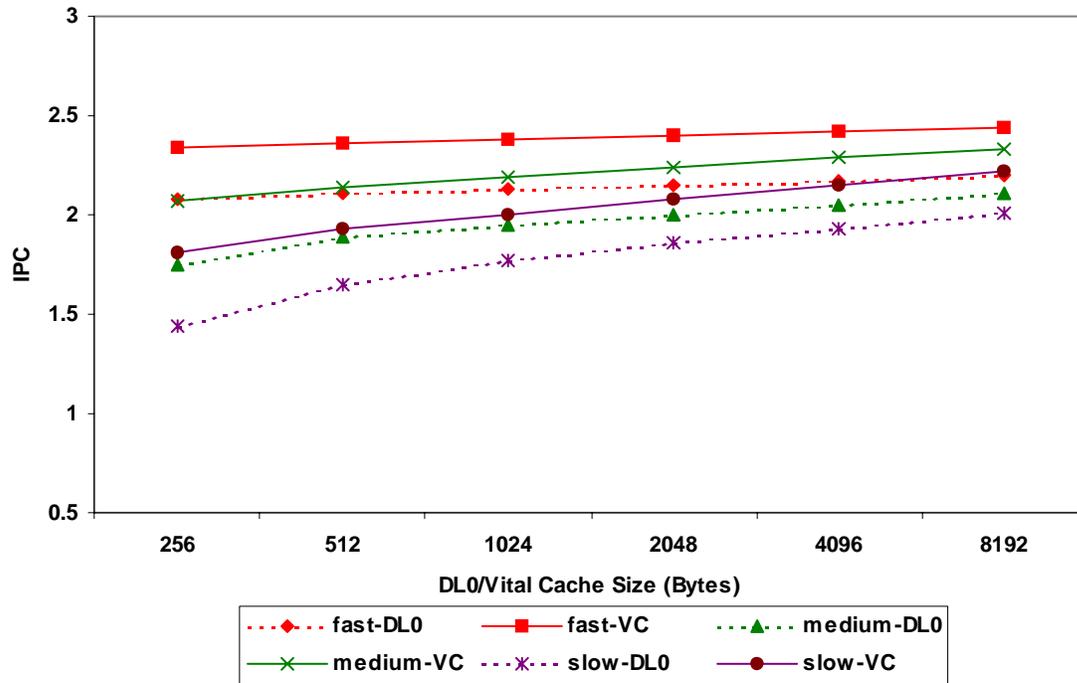**Figure 12 Three variations of varying cache latencies.**

Figure 13 DL0 vs. vital cache with varying cache latencies.

Figure 13 also shows that as the latencies of the caches become greater, the importance of VC becomes greater. For example, assuming "slow" caches, a 256B VC outperforms a 256B DL0 by 26%. It can also be noted that a slow-VC configuration performs equivalent to a fast-DL0 configuration. As microprocessor clock frequencies continue to increase, the importance of the VC will also increase.

6   Conclusion

In this paper, a new load classification method is proposed, that classifies loads as those vital to performance and those not vital to performance. A study is presented that analyzes different types of non-vital loads and quantifies the percentage of loads that are non-vital. 75% of all loads are non-vital with only 35% of the accessed data space being vital. A realistic implementation of the non-vital load classification method is presented and a cache called the vital cache is proposed that makes use of the non-vital load classification. The vital cache is designed to cache data for vital loads only, deferring non-vital loads to slower caches. The vital cache improves the efficiency of the cache hierarchy and hit rate for vital loads. The vital cache increases performance by 17%, relative to a design without the vital cache.

As microprocessors continue to push towards even higher frequencies, the efficiency of the first and second level caches will become more and more critical. In order to make more efficient use of these caches, we must leverage the dynamic behavior of memory accessing instructions. This paper has shown that the vitality attribute of load instructions can be effectively leveraged to enhance cache efficiency and hence overall processor performance.

References

[1] S.G. Abraham, R.A. Sugamar, B.R. Rau and R. Gupta, "Predictability of Load/Store Instruction Latencies", Proc. 26th International Symposium on Microarchitectures, Dec 1993, 139--152.

[2] A. Agarwal and S. D. Pudar. "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches", In Proceedings of the 20th International Symposium on Computer Architecture, pages 169--178, San Diego, CA, May 1993.

[3] T. Alexander and G. Kedem. "Distributed Prefetch-buffer/Cache Design for High Performance Memory Systems", In Proceedings of the 2nd International Symposium on High-Performance Computer Architecture, pages 254--263, February 1996.

[4] B. Calder, D. Grunwald, and J. Emer, "Predictive sequential associative cache," in Proceedings of the Second International Symposium on High-Performance Computer Architecture, Feb. 1996.

[5] S. Cho, P. Yew, and G. Lee. "Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor," Proc. of the 26th Int'l Symp. on Computer Architecture.

[6] Brian A Fields, Shai Rubin and Rastislav Bodik. "Focusing Processor Policies via Critical-Path Prediction", In proceedings of 28th International Symposium on Computer Architecture.

[7] B. Fisk and I. Bahar. "The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency", In IEEE International Conference on Computer Design, October 1999.

[8] M. D. Hill. "A Case for Direct-Mapped Caches," IEEE Computer, pp. 25 - 40, Dec. 1988.

[9] Johnson, Connors, Merten, & Hwu. "Run-Time Cache Bypassing", in IEEE Transactions on Computers, Vol. 48, No.12, December 1999.

[10] L. John and S. A. "Design and performance evaluation of a cache assist to implement selective caching." In International Conference on Computer Design.

[11] T. Juan, T. Lang, and J. Navarro. "The Difference-bit Cache." Proc. of the 30th Annual Int'l Symp. on Microarchitecture.

[12] N. P. Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers." In 17th Annual International Symposium on Computer Architecture.

[13] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. "Inexpensive implementations of set-associativity," Proceedings of the 16th Annual International Symposium on Computer Architecture, 17(3):131--139, 1989.

[14] Kin, Gupta, and Mangione-Smith. "The Filter Cache: An Energy Efficient Memory Structure", International Symposium on Microarchitecture, 1997.

[15] L. Kurian, P. T. Hulina, and L. D. Coraor. "Memory latency effects in decoupled architectures." IEEE Transactions on Computers, 43(10):1129--1139, October 1994.

[16] H. Neefs, H. Vandierendonck, and K. De Bosschere. "A Technique for High Bandwidth and Deterministic Low Latency Load/Store Accesses to Multiple Cache Banks," Proceedings of the 5th International Symposium on High-Performance Computer Architecture.

[17] B. Rau. "Pseudo-Randomly Interleaved Memory," 18th International Symposium on Computer Architecture, May 1991.

[18] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. "On High-Bandwidth Data Cache Design for Multi-Issue Processors," Proc. of the 30th Annual Int'l Symp. on Microarchitecture, pp. 46 - 56, Dec. 1997.

[19] Sanchez, Gonzalez, & Valero. "Static Locality Anlysis for Cache Management", In the Proceedings of PACT, November 11-15, 1997.

[20] A. Seznec. "A case for two-way skewed-associative caches." In 20th Annual International Symposium on computer Architecture.

[21] Srinivasan , Ju, Lebeck, & Wilkerson, "Locality vs. Criticality", in Proceedings of the 28th International Symposium on Computer Architecture, 2001.

[22] Srinivasan and A. Lebeck, "Load latency tolerance in dynamically scheduled processors.", in Proceedings of the Thirty-First International Symposium on Microarchitecture, pp. 148--159, 1998.

[23] G. S. Sohi and M. Franklin. "High-Bandwidth Data Memory Systems for Superscalar Processors," Proc. of the Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 53 - 62, April 1991.

[24] Tune, Liang, Tullsen, Calder. "Dynamic Prediction of Critical Path Instructions". In the Proceedings of the 7th High Performance of Computer Architecture.

[25] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. "A modified approach to data cache management." In Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture, pages 93--103, December 1995.

[26] K. M. Wilson and K. Olukotun. "Designing High Bandwidth On-Chip Caches," Proc. of the 24th Int'l Symp. on Computer Architecture, pp. 121 - 132, June 1997.

IEEE
COMPUTER
SOCIETY