Intelligent Cache Management by Exploiting Dynamic UTI/MTI Behavior

Joshua B. Fryman, Chad M. Huneycutt, Luke A. Snyder,
Gabriel H. Loh, Hsien-Hsin S. Lee
Center for Experimental Research in Computer Systems (CERCS)
Georgia Institute of Technology
Atlanta, GA 30332-0280
{ fryman, chadh, lukesn, loh, leehs }@cercs.gatech.edu

Abstract

This work addresses the problem of the increasing performance disparity between the microprocessor and memory subsystem. Current L1 caches fabricated in deep submicron processes must either shrink to maintain timing, or suffer higher latencies, exacerbating the problem. We introduce a new classification for the behavior of memory traffic, which we refer to as target behavior. Classification of the target behavior falls into two categories: Uni-Targeted Instructions (UTI) and Multi-Targeted Instructions (MTI). On average, 30% of all dynamic memory LD/ST operations come from execution of UTIs, yet only a few hundred static instructions are actually UTIs. This makes isolation of the UTI targets an avenue for optimization. The addition of a small, fast cache structure which contains only UTI data would ideally reduce MTI pollution of UTI information. By intelligently selecting between larger, slower data caches and our UTI cache, we reduce the latency problem while increasing performance.

Our distinct contributions fall in three areas, with implications to many others: (1) we present a new characterization of memory traffic based on the number of targets from LD/ST instructions; (2) we explore the underlying nature of the target division and devise a simple mechanism for exploiting regularity based on a UTI cache; (3) we explore a variety of prediction mechanisms and processor configuration options to determine sensitivity and the performance gains actually attainable under different modern processor configurations. We attain up to 42% IPC improvements on SPEC2000, with a mean improvement of 8%. Our solution also reduces L2 accesses by up to 89% (average 29%), while reducing load-load violation traps by up to 84% (average 13%), and store-load violation traps by up to 43% (average 8%).

1 Introduction

With every new generation of microprocessor, pressure due to the memory bottleneck increases. Many techniques have been implemented to avoid the penalties associated with main memory access, such as non-blocking caches, prefetching, and value prediction. Arguably we can compute results as fast as our power and real estate budget will tolerate, so long as the data necessary is present. Since obtaining the information necessary for computation, whether actual data or instructions, remains the chief bottleneck in computation, many projects have examined how to move data more efficiently. These projects have built models based on variations of dataflow principles, pre-computation [16], data decoupling [4], as well as extensive compiler analysis for better memory management [13].

As architects move forward, the near future offers die capacities on the order of one billion transistors. Active research in all fields continues to investigate how to effectively use these transistors to continue the performance improvements the industry has sustained over the past 40 years.

Conventional wisdom has been that if no better usage can be found, the conversion of unused transistors in any die can be turned into an on-die L3 cache. The assumption is that adding more cache will be a worthwhile investment of resources. Using an approximation that modern microprocessors, with L1 and L2 caches, comprise some 200M transistors, the remaining 800M transistors would make at best a 16MB L3 cache. Even using a generous 20-cycle access time for this L3 cache, Figure 1 shows just what impact such a use of resources would have on the full SPEC2000 benchmark suite. The change in geometric mean of IPC when adding the L3 cache to an Alpha 21264 is an increase of 1.5%. This small change is primarily due to memory-bound applications, with the L3 cache experiencing a high miss rate. Therefore, just adding larger caches may not be

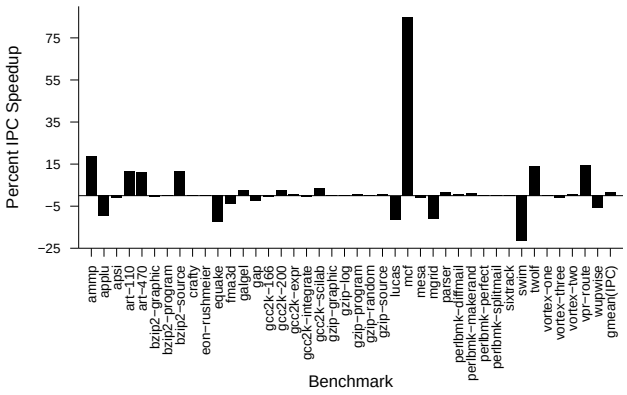

Figure 1: **Adding a 16MB L3 20-cycle cache to a 21264 has 1% average impact on IPC for SPEC2000. Based on standard Alphasim [5] with L1 I,D caches: 64KB 2-way, 1,3-cycle; L2 cache: 1MB, 2-way, 7 cycles; L3 cache: 8-way, 20 cycles.**

an ideal approach unless the working set of applications increase accordingly.

We propose in this paper a new mechanism to evaluate memory operations. Using this mechanism, we isolate 30% of all memory traffic in a special cache structure of just 2KB, increasing system performance by reducing pollution effects. Using a simple predictor off of the critical path, we choose between the normal cache and our cache to generate performance improvements and reduction in L2 cache accesses. Other groups have proposed the addition of a small, fast cache but have used it in different ways to achieve mixed results. The micro-cache [22] was used to isolate critical-path data, while the stack value file [11] isolated stack data.

In specific, we contribute three primary results in this work:

- a new mechanism to classify memory operations
- a thorough exploration of this classification
- results and sensitivity studies based on different aspects of our system

The rest of this paper is organized as follows. In Section 2, we present our new idea for memory target behavior characterization, and show how different benchmarks have different characteristics. We describe a new design to exploit this behavior in Section 3. The performance gains from our characterization and solution are presented in Section 4. Section 5 studies the performance sensitivity of a variety of microarchitecure parameters. Our results and methods have implications in many areas, and highlights of these are in Section 6. Finally, Sections 7 and 8 discuss related work and conclude.

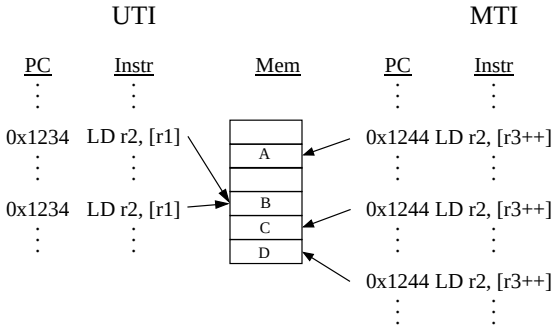

Figure 2: **Basic concept of Uni-Targeted Instructions and Multi-Targeted-Instructions illustrated.**

2 UTI/MTI Characterization

Memory operations can be classified into two primary categories, Uni-Targeted Instructions and Multi-Targeted Instructions, based on their reference behavior. A Uni-Targeted Instruction (UTI) is a memory operation that only accesses one unique memory address over a dynamic trace of instructions. A Multi-Targeted Instruction (MTI) accesses multiple memory addresses over the the same trace.

Both UTI and MTI occurrences are identified by the PC of the instruction, as illustrated in Figure 2. In the figure, the LD instruction at PC `0x1234` reads from the target address in `[r1]`, which is constant regardless of where the instruction is executed dynamically. This is an example of UTI behavior. The LD instruction at PC `0x1244` reads from `[r3]`, yet the value in this register changes as the dynamic execution progresses. This is an instance of MTI behavior. Conceptually the UTI may be using a global variable, whereas the MTI may be traversing an array or chasing pointers.

2.1 UTI/MTI Dynamic Distribution

To quantify the distribution of UTI and MTI targets in applications, Figure 3 (a) shows the breakdown for the SPEC2000 benchmark suite over the *entire* application. Figure 3 (b) shows the breakdown on 100M instruction traces from the interval chosen by early SimPoints [15].

While these results have some particularly large individual variations (bzip, gcc, etc.), the average results are similar (31% dynamic UTI for the full run, 29% for the SimPoints version). To accelerate our simulations, we use the SimPoints with the expectation that individual IPC gains may vary as suggested by these early results, yet the mean should be indicative of the result were full benchmark runs used.

The correlation between full applications and SimPoints for UTI behavior is shown in Figure 4. To avoid the more glaring error cases from skewing our results, we subset our benchmark programs by discarding any benchmark that has over 75% disagreement between the the UTI/MTI ratio of full runs compared to SimPoints. Another interesting impli-

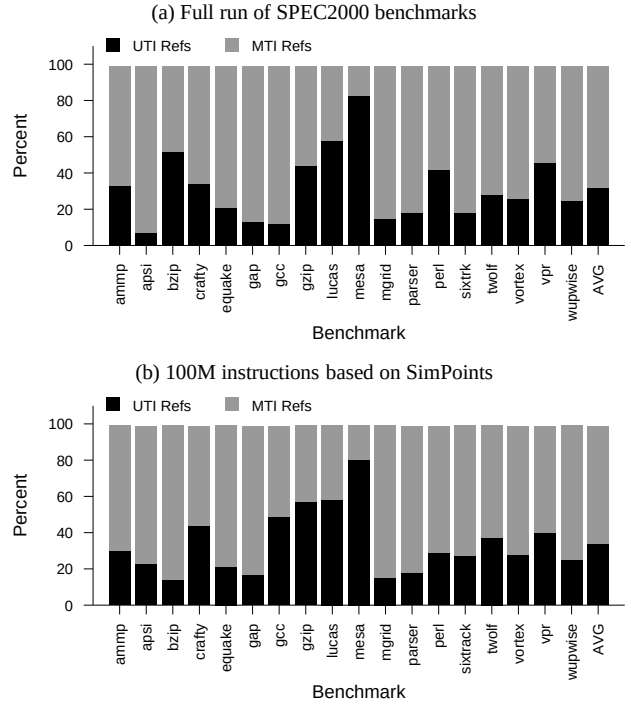

Figure 3: **Distribution of UTI/MTI dynamic instances.**

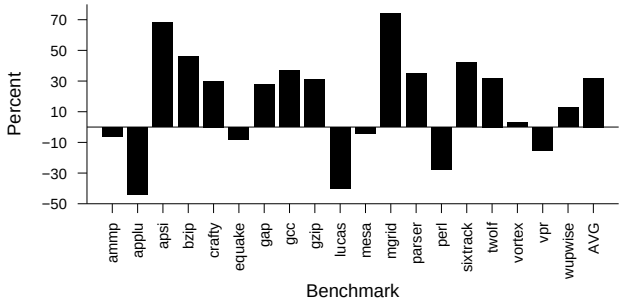

Figure 4: **The difference between full application UTI dynamic instruction percentage and the SimPoints based 100M instruction benchmark subset.**

cation is that SimPoints does not accurately reflect this type of memory behavior, which requires future exploration.

2.2 UTI/MTI Static Distribution

Building on this separation at the instruction level of UTI or MTI traffic, a detailed simulation demonstrates that the actual number of UTI targets is small despite the substantial dynamic instruction percentage. Table 1 shows the results over the SPEC2000 benchmarks in classification of the UTI-MTI behavior. The average unique UTI targets for the full run is merely 1350, and can be as few as 301. For the SimPoints execution, this average drops to 270 (not shown in Table 1).

Of the actual program LD/ST instructions, UTIs are less

SPEC 2000	Dynamic Insns.		Static Insns.		Uniq. Targets	
	UTI	MTI	UTI	MTI	UTI	MTI
ammp	56.0 B	115 B	2.57 K	43.2 K	638	2.63 M
applu	74.6 B	94.3 B	7.80 K	111 K	1262	22.7 M
apsi	12.8 B	168 B	11.0 K	139 K	1256	25.0 M
bzip2	27.2 B	25.6 B	2.27 K	28.2 K	301	78.5 M
crafty	28.1 B	55.4 B	6.49 K	102 K	1095	408 K
eon	12.7 B	24.9 B	15.2 K	127 K	5637	101 K
equake	12.1 B	46.5 B	1.92 K	29.4 K	490	6.88 M
facerec	17.1 B	53.2 B	3.76 K	72.6 K	1032	4.08 M
fma3d	58.0 B	77.6 B	11.1 K	143 K	3016	15.1 M
galgel	2.19 B	169 B	8.54 K	126 K	1478	4.71 M
gap	13.5 B	89.9 B	4.08 K	77.3 K	1551	25.0 M
gcc	2.21 B	16.1 B	19.2 K	599 K	3312	41.1 M
gzip	19.0 B	24.0 B	2.04 K	29.5 K	429	15.5 M
lucas	30.6 B	22.4 B	3.90 K	50.0 K	722	20.8 M
mcf	889 M	19.4 B	1.33 K	20.9 K	369	24.9 M
mesa	113 B	22.5 B	5.77 K	58.1 K	1478	4.97 M
mgrid	73.6 B	403 B	2.63 K	48.0 K	608	7.27 M
parser	39.9 B	187 B	37.7 K	79.9 K	508	14.7 M
perl	4.66 B	6.42 B	6.82 K	89.4 K	1436	173 K
sixtrk	626 M	2.77 B	12.0 K	159 K	2548	2.79 M
swim	58.4 B	92.6 B	3.41 K	51.4 K	770	25.0 M
twolf	41.8 B	107 B	8.10 K	111 K	1004	1.00 M
vortex	13.3 B	37.0 B	12.4 K	216 K	1573	17.2 M
vpr	533 M	623 M	3.27 K	47.6 K	720	602 K
wupw	26.8 B	81.2 B	2.92 K	46.7 K	589	23.1 M

Table 1: **The breakdown of MTI and UTI information over complete runs of SPEC2000 benchmarks. The dynamic instruction count only reflects LD/ST traffic. The static instruction counts are those actual PCs which correspond to either UTI or MIT over the program lifetime. The unique targets are unique memory addresses over the lifetime of all dynamic instructions that are classified as UTI or MTI.**

than 6% as indicated by the static instruction data in Table 1. However, these 6% of instructions comprise 30% of the dynamic LD/ST references. This trend of very few actual PCs generating a substantial amount of memory traffic should be readily identifiable. Since those same PCs only access a few hundred unique memory locations, the data for these operations will fit into very small caches. Less than 5400 bytes (1350 targets $\times$ 4 bytes) are required to hold the entire UTI data set for the full benchmark runs on average. Discussion of line sizing is in Section 5.2. Less than 1200 bytes are required for 100M instruction windows based on SimPoints. The MTI data set, however, comprises at least tens of megabytes.

Since approximately 5400 bytes can capture 30% of the dynamic LD/ST traffic, which is generated by relatively few instructions, a mechanism to capture this information in the memory hierarchy may lead to performance gains. As a minimum, isolation of such UTI information will eliminate pollution in this 30% of memory traffic. Since the actual

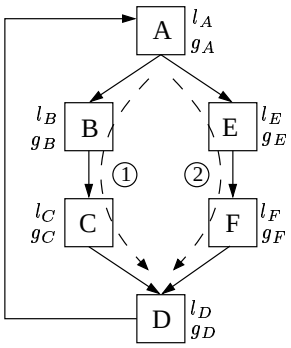

Figure 5: **Example program structure to illustrate the concept of phasing and ratio changes.**

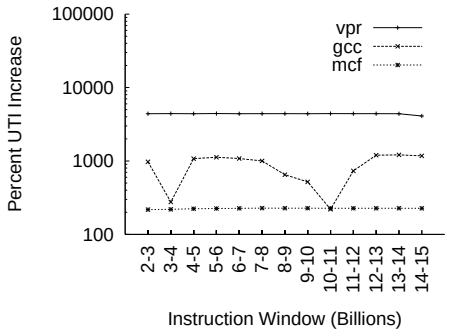

Figure 6: **By resetting all UTI-MTI state every billion instructions, the relative percentage of UTI targets increases indicating phase behavior.**

unique targets represent less than 0.01% of all memory targets, various prediction mechanisms may be useful to capture this behavior.

2.3 Phasing

The consideration of MTI targets as a series of stable UTI targets may capture call-path locality in the reference stream. Repeated chains of function calls may experience periods where local variables are actually at constant addresses in the stack. We examine a few full benchmark runs to determine whether this concept of phasing is valid. The observed MTI behavior in the stack suggests that discarding history (beyond a certain age) may reveal that MTI memory operations may be temporally reclassified as UTI. Therefore an age-based decay of history may reveal other trends inside of the UTI-MTI landscape.

To illustrate the concept of phasing, consider Figure 5. Assume that all global variables, g_i, are accessed via UTI and that all local variables, l_i, are accessed via MTI. If the program execution flow is a repeating sequence alternating between the paths $\{1, 2\}$, such that $ABCD \rightarrow AEFD \rightarrow ABCD \rightarrow \ldots$, then the local variables on the stack may change locations. Using a shorthand of ΣXi to represent the total unique addresses of type X, we can define the UTI unique target ratio R_{UTI} of this code hammock as:

$$R_{UTI} = \frac{\Sigma g_i}{(\Sigma g_i + \Sigma l_i)} \quad (1)$$

The continuous oscillation between the execution paths $\{1, 2\}$ prevents any local variable from acting as UTI. However, if the execution path were to continuously be *only* one of the paths $\{1, 2\}$, such as $ABCD \rightarrow ABCD \rightarrow \ldots$, then every pass through the loop will access local variables at the same address on the stack. Even though we assumed for this example that local variables are MTI, if this single path executes sufficiently long we can treat these variables as UTI. This would change our R_{UTI} to be:

$$R_{UTI} = \frac{(\Sigma g_i + \Sigma l_i)}{(\Sigma g_i + \Sigma l_i)} \quad (2)$$

For this trivial example, the phasing behavior reduces R_{UTI} to the constant 1 – indicating that *all* memory references are effectively UTI. In reality, the actual fluctuation based on array accesses, function call paths, and other variables will cause the R_{UTI} to fluctuate during any selected window of dynamic execution.

However, we can calculate based on Table 1 the *expected* ratio R_{UTI} over the lifetime of each benchmark. This ratio is what we expect to find if we pick a window of dynamic instructions from that benchmark and re-compute the ratio over the memory access stream in that window. By comparing the per-window R_{UTI} to the full benchmark calculation, it becomes possible to determine the relative increase or decrease of the unique UTI targets with respect to the total unique targets in that window. If the window ratio is increasing, there are more UTI available for our system to work with. If the window ratio is decreasing, there are fewer UTI for our system. Therefore, the ratio R_{UTI} can be used as an approximation to the phasing behavior in a window of dynamic instructions.

By resetting the captured UTI-MTI state information every billion instructions, we analyze a few benchmarks to determine how their UTI percentage changed compared to a full application classification. Figure 6, using a log-scale Y axis, shows the results for gcc, gzip, mcf, and vpr, skipping the first two billion instructions to avoid warm-up behavior. These few benchmarks show the trends that may be observed over all the benchmarks. Substantial phasing behavior appears with changes between 210-1200% in gcc and a nearly constant 225% in mcf and 4400% in vpr.

As the UTI occurrence increases, any scheme we design for capturing UTI behavior should have more opportunities for performance improvement. However, any decay of history which is too aggressive may result in over pressuring the isolated cache, reducing performance. To obtain the best performance possible, a careful support of phasing information should be included in any design.

3 The UTI cache

From the prior characterization and discussion of memory reference behavior, several key points suggest an opportunity for exploitation.

1. UTI references are 30% of all traffic on average.
2. There are only 1350 unique UTI targets on average.
3. Approximately 5400 bytes can store this UTI data.
4. Some MTIs may act as UTIs over 100M-1B insns.

The counting of distinct UTI targets, when considering the 100 M instruction window via SimPoints, is actually smaller – approximately 270 on average over the SPEC2000 suite. These 270 average unique targets require less than 1200 bytes to isolate the UTI data.

Our hypothesis is that the 70% of memory traffic which is MTI based continuously conflicts with the 30% that is UTI. The MTI data covers 99.99% of the unique target addresses in the dynamic program, whereas the UTI amounts to barely 0.01%. By preventing the pollution between UTI and MTI data, substantial performance gains can be attained.

3.1 Capturing UTI Behavior

We propose to add a small, 1-cycle or faster cache to a traditional processor. This small cache, inserted next to the standard L1 data cache, captures UTI data preventing MTI pollution. By removing the UTI data from the L1 data cache, the standard data cache will act as an MTI cache.

A new problem arises in how to select between the standard L1 data cache and our small UTI cache. The critical path is negatively impacted by delaying cache selection until effective address generation. This impact necessarily occurs whenever isolating certain types of data or otherwise breaking up a monolithic cache into individual caches. Prior art to hide the increase in latency includes address prediction [11], fast address calculation [1], and value prediction [20]. We propose to avoid this problem by using a predictor at instruction decode time, based only on the PC of the LD/ST operation, as shown in Figure 7. This predictor lies off the critical path, returning a prediction before the LD/ST queue attempts to access caches. Results based on this predictor are shown in Section 4, while design space of this predictor is covered in Section 5.1.

The predictor selects between the UTI and MTI caches at decode time. When the LD/ST instruction executes, the memory controller uses the cache indicated by the prediction; simultaneously, the predictor uses the PC and effective address of the LD/ST operation to update its state.

To keep the caches consistent, we must safely handle missing in the predicted L1 cache while the data is live in the alternate L1 cache. We implement this during update from the L2 cache, invalidating information in the non-predicted

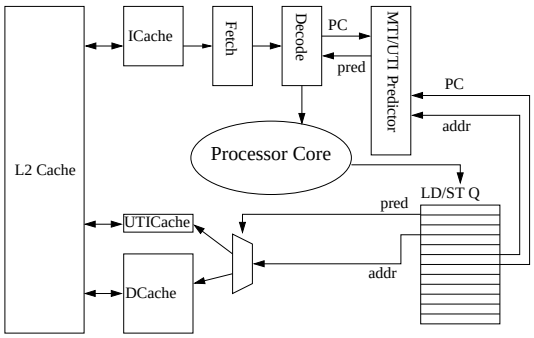

Figure 7: **Addition of a very small UTI cache to the standard processor design path to capture UTI/MTI behavior.**

Core Parameter	**Value**
Core Freq	3.2 GHz
Branch Pred	Tournament
MSHRS: Prefetch/Cache	8/8
L1 Icache	**Value**
Size/Line Size/Assoc	16KB/64B/8way
Hit Time/Prefetch	1 cycle/Enabled
L1 Dcache	Value
Size/Line Size/Assoc	8KB/128B/8way
Hit Time/Prefetch	3 cycle/Enabled
L2 (Non-Blocking)	**Value**
Size/Line Size/Assoc	1MB/128B/8way
Hit Time/Prefetch	18 cycle/Enabled
Memory (Non-Blocking)	**Value**
Type	DDR SDRAM
Rating	PC-3200
CAS/RAS/PRE/Bus Mult	3/3/3/8

Table 2: **The baseline processor model and cache configurations for this study. The Alpha 21264 uses prediction to achieve a 1-cycle hit time in the instruction cache. All values not shown are the default 21264 model parameters in Alphasim.**

cache. This type of handling is similar to snooping on a bus between L1 and L2 caches, but without the benefit of faster access times possible to neighbor caches.

3.2 Experimental Framework

To evaluate our UTI cache system, we use Alphasim [5]. Alphasim was calibrated against an aggressive out-of-order Compaq Alpha 21264 processor. We anticipate our results should be representative of real gains that may be achieved in contemporary and future processors. The changes from baseline parameters are shown in Table 2. These parameters were chosen to be representative of a recent high frequency processor, such as the Intel Pentium 4 [3].

We extend Alphasim to track LD/ST operations with a prediction bit in the reorder buffer. As soon as an instruction is decoded, the predictor is consulted and the predic-

tion bit set. The commit and writeback stages in Alphasim update the predictor using the computed target address and generating PC.

The outcome of the prediction is true for MTI cache access, or false for UTI cache access. The actual algorithms and types of predictors are more fully explored in Sections 4 and 5. An always-taken predictor should behave exactly the same as an unmodified copy of Alphasim with the same parameters as shown in Table 2.

4 Experimental Results

To determine what performance impact our new cache structure may have, we must fix a design for both the UTI cache and our predictor. The actual configuration of the UTI cache is a result of the sensitivity analysis for this cache. As discussed in Section 5.2t he maximum efficiency of the UTI cache occurs when the 2KB unit is divided into 4-byte lines with 32-way associativity. Dropping from 32-way associativity to 8-way reduces performance by 1%. For our discussion we consider only 32-way.

The predictor for our study is based on the common work of many branch predictor efforts. While we implemented a few basic predictor types, the rest of which are covered in Section 5.1, we did not make a comprehensive effort to optimize our predictor performance. Instead we attain substantial performance gains, after trying only a few variations of basic parameters. Our results thus far indicate that a more aggressive predictor will achieve even better results.

The end result is an IPC improvement over SPEC2000 of up to 42%, with a geometric mean of IPC improvement of 8%, shown in Figure 10. Individual improvements vary from 0% on lucas to 42% on perl, but importantly no benchmark experiences a negative impact to IPC. While IPC improvements are beneficial, they are not the entire picture. Changing the cache structure will be sensitive to many other factors, such as MSHR entries, memory bandwidth, associativity, etc. Section 5 explores these issues.

The basic design of our predictor uses a total of 8KB of state space. The predictor is arranged into 4,096 slots with 16-bit entries at each slot. Each entry consists of four subfields: (1) a PC tag of 5 bits; (2) an address tag of 6 bits; (3) a counter of 3 bits; and (4) steady-state of 2 bits.

The PC is used to generate both an index into the 4,096 slot table from the lower bits as well as a small tag from the upper bits, as shown in Figure 8. On a lookup, if the tag in the slot matches the tag from this PC, then the prediction is UTI if the counter is fully saturated to the maximum value. Under any other condition – tag mismatch or unsaturated counter – the prediction is MTI.

The update logic breaks down into two basic cases: tag match and tag mismatch. The block diagram of updating the

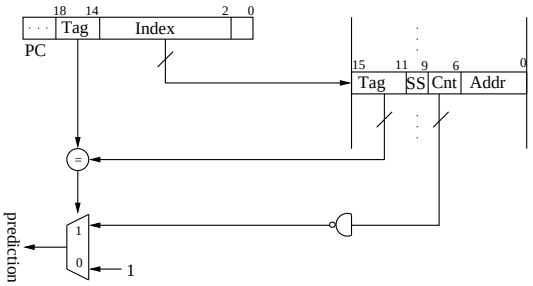

Figure 8: **The basic logic of obtaining a prediction.**

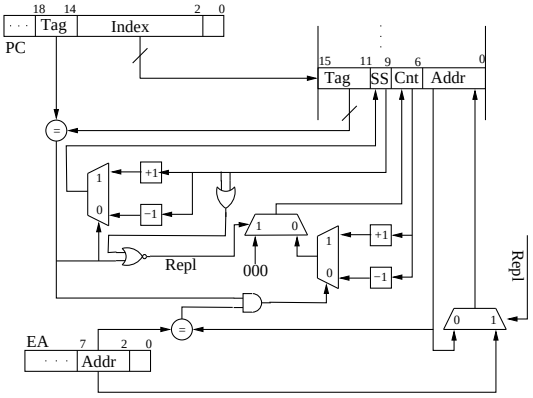

Figure 9: **The basic logic of updating predictor state.**

predictor is shown in Figure 9. In the case of tag mismatch, the steady-state (SS) counter is decremented unless it is already at zero. When the steady-state is zero in a tag mismatch, then both the tag and address subfields are replaced with the new tag and effective address from this operation. All arithmetic is saturating.

In the case of a tag match, the deciding factor is whether the address fields match. If the address fields match, then the steady-state and counter values are incremented. If the address fields fail to match, then the counter is decremented while the steady-state increments due to the tag match. These steady-state bits coupled to the counter catch phasing behavior, as discussed in Section 2.3.

5 Sensitivity Analysis

There remain many unexplored aspects of the results presented in Section 4. Specifically, our system is designed to improve the latency of memory operations. To study how other changes impact our results, we next examine a variety of parameters that may interfere with the behavior the UTI cache captures. In Section 5.1, we look at how our predictor compares to simple alternatives. In Section 5.2 we explore the locality of the UTI cache as alternate arrangements of the 2KB structure are evaluated. Section 5.3 considers the performance changes as the non-blocking MSHR count is varied from 1 to 16. Section 5.4 explores how our solution changes the memory pressure itself in terms of main mem-

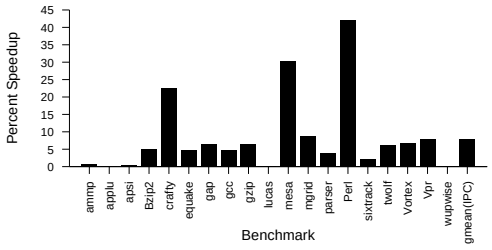

Figure 10: **IPC improvements over the baseline data cache model with a simple predictor for our UTI cache.**

ory accesses, L2 accesses, and other considerations. Another concern is the amount of state space storage required for our predictor, which is discussed in Section 5.5. For comparison, we illustrate how our scheme compares to alternate uses of our increase in storage for modern systems in Section 5.6.

5.1 Basic Predictors

The predictor presented in Section 4 is straightforward, but lacks any basis for comparison to other strategies. For comparison, we implemented two trivial predictors: random and always UTI. The random predictor is exactly as named. The always UTI case corresponds to always predicting the UTI cache. The third type of predictor we implemented for comparison was what we refer to as the history predictor. This kept every PC and every address referenced from that PC in a large tree, using the entire past history to make an absolute decision as to whether the instruction is UTI or MTI.

The results of these predictors are shown along with our real predictor in Figure 11. These results are relative to the baseline system. As expected, the always UTI case is a substantial performance penalty, yet it outperforms all others on crafty, perl, and vpr. The random prediction is almost entirely a negative result, failing to perform well since thrashing can occur between the two L1 data cache structures.

The infinite history predictor also outperforms the real predictor, attaining the best results on several benchmarks: ammp, applu, apsi, gcc, gzip, mgrid, sixtrack, twolf, and wupwise. The infinite history has slightly higher performance than our real predictor in just a few cases, demonstrating that our predictor is clearly not optimal.

None of these predictors represent a ceiling or floor on performance. Future work is to improve upon these results with more intelligent prediction mechanisms.

What is somewhat surprising at first glance is that both the always UTI and the random predictor can actually outperform the baseline data cache. For these benchmarks, the limitation is not cache capacity but the latency of cache access (i.e., critical path). The UTI cache, at one cycle to access, meets the underlying needs of the application. As expected, however, the general case is that using the UTI

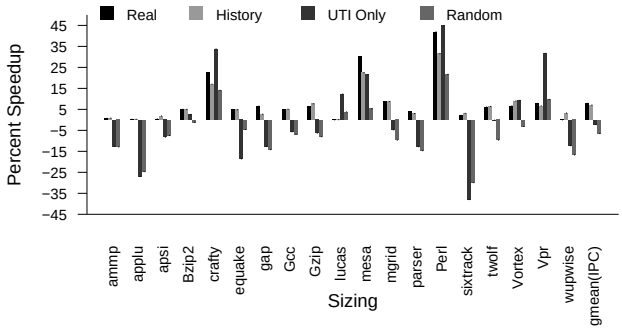

Figure 11: **Percent speedup of Not-Taken (UTI) and Random predictors over the Always-Taken case (baseline data cache).**

cache by itself or unintelligently (random) translates to a substantial performance loss.

Comparing Figure 10 to Figure 11 shows a correlation between those benchmarks which are dependent on latency and the magnitude of gains seen with our UTI cacheselection with a real predictor. With the real predictor, we find that these benchmarks all exhibit good performance increases, with mesa and perl the top two. However, we also see that intelligent management of the UTI cache pays off. With poor selection, sixtrack was the single worst performer for random and always UTI predictors. With our simple predictor, we attain 2.2% IPC gain for sixtrack. Even those applications which are not obviously latency dependent, but rather seem dependent on cache size, can benefit from our technique.

5.2 UTI cache Structure

The optimal design parameters for the UTI cache requires a sweep of several variables. Our critical concern of limiting the access time to 1 cycle at the core processor frequency limits how large our cache can be. We chose the 2KB data limit for the UTI cache to address this concern. To fully explore the options of line size, associativity, and number of sets, we did a sweep of cache parameters over the SPEC2000 suite. Keeping our UTI cache size fixed, we vary line size from 4B to 64B, and associativity from 1-way to 32-way. Figure 12 shows the results of this sweep.

As intuition suggests, the isolation of UTI targets into a separate cache effectively destroys locality. Small line sizes with moderate to high associativity outperform long cache lines traditionally used to capture spatial locality. The 32-way, 4B line size option achieves 8% IPC speedup on average. While this configuration nearly doubles the storage space due to the tag overhead on 4B line size, it is still accessible within 1 cycle. If we instead settle for an 8-way UTI cache, our performance improvement drops to just under 7.5%.

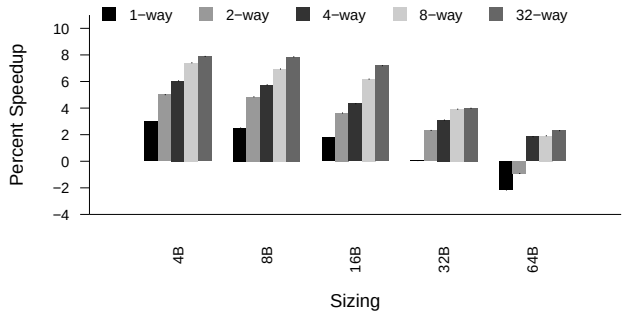

Figure 12: **Percent geometric mean IPC speedup for the SPEC2000 average result over the baseline data cache as the UTI cache configuration is varied. The line size, in bytes, is represented by the x-axis, while each bar corresponds to a different associativity.**

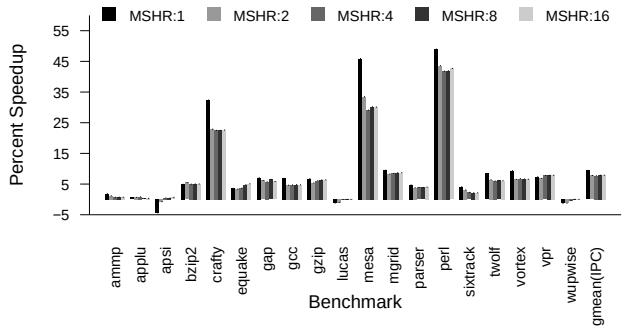

Figure 13: **Percent IPC speedup for the SPEC2000 average result over the baseline data cache as the number of MSHRs vary.**

5.3 MSHR Sensitivity

Thus far, we have explored our UTI cache as a primary solution for improving application performance. The Alphasim system also includes MSHRs for the prefetch system and regular cache MSHRs. The basic purpose of the MSHR in any context is to reduce the blocking due to limited ports of the memory subsystem. To explore the impact of varying MSHR capacity, we analyze the spectrum of IPC gains as all MSHR slots were uniformly run through the range of 1, 2, 4, 8, and 16. The results of this are shown in Figure 13.

Typical modern processors such as the Pentium4 implement 4-6 MSHR slots. By running through this large spectrum, we demonstrate that while our system reaches nearly 10% IPC improvement when the MSHRs are set to 1, it is still a win of 8% IPC improvement with MSHRs of 8. Even with a very aggressive 16-entry MSHR system, our UTI cache still results in noticeable IPC gains. In some cases, our results show no impact due to the increased MSHR capacity.

Another trend in Figure 13 is that some benchmarks

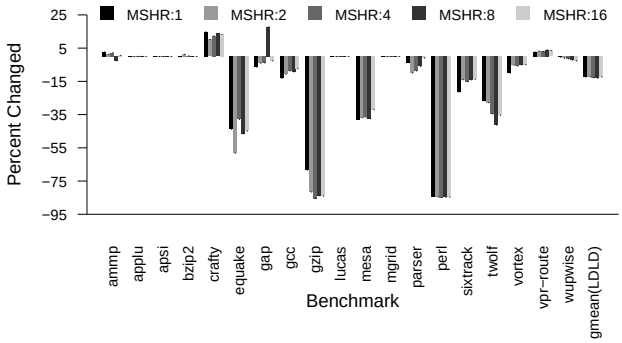

Figure 14: **Percent change in load-load violations of the memory ordering for SPEC2000, with varying MSHR levels.**

display rising performance after the initial dropoff (as the MSHR count is increased from 1). The benchmark gzip, for example, nearly attains the same IPC if the MSHR count is 1 or 8+.

5.4 Memory Pressure

Another side effect of our UTI cache addition is the reduction of memory pressure. We measure memory pressure in four metrics: (1) the load-load memory violations; (2) the store-load memory violations; (3) L2 cache accesses; and (4) DRAM accesses. The first two pressures are directly related to how long memory operations take. If a load has blocked, waiting for memory, the likelihood of a subsequent load or store interfering with it is directly proportional to the duration of the first operation blocking. Therefore, any reduction is a boost to the overall efficiency of the processor.

Similarly, by reducing the amount of accesses into the L2 cache, we are improving the utilization of our existing cache storage. Reducing DRAM utilization would be a further improvement, but is unlikely to occur with such a small UTI cache.

With our UTI cache again sweeping the range of MSHR values, Figures 14, 15, 16, and 17 respectively show the results of our four metrics under simulation.

Our load-load metric exhibits a range of behavior, peaking with a reduction by 85% in perl with 8 MSHR slots. Conversely, our load-load violations on gap gets worse by 17%. The geometric mean load-load violation reduction over SPEC2000 is approximately 13%. The store-load violation behavior is even better, with a general reduction everywhere except for ammp with an MSHR of exactly 2. The general reduction of store-load violations is approximately 8%. While this suggests that use of a UTI cache may facilitate reduction of load-store queues by some amount, we have not yet simulated the effect this would have on overall performance.

Our L2 access metric is almost a uniform reduction in total bandwidth. While our reduction is up to 89%, the ge-

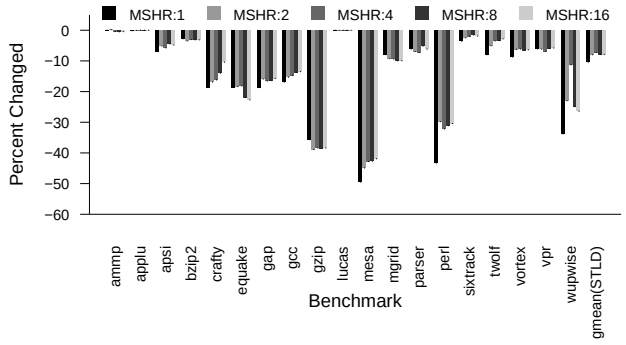

Figure 15: **Percent change in store-load violations of the memory ordering for SPEC2000, with varying MSHR levels.**

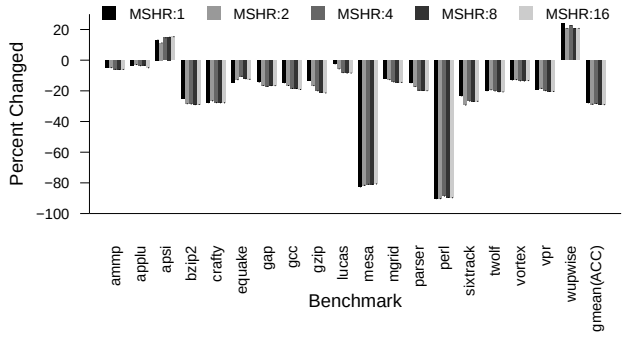

Figure 16: **Percent change in L2 accesses for SPEC2000, with varying MSHR levels, compared to baseline data cache with equal MSHRs.**

ometric mean impact is 29% reduction in L2 accesses for 8 MSHR slots. The DRAM changes are relatively small, within the range of $\pm 5\%$. The geometric mean reduction in DRAM accesses is 0.04% with a 8 MSHR slots. Since misses are primarily compulsory, not capacity, this is unsurprising.

5.5 State Space

Our real predictor, while off the critical path, uses a moderate amount of state space storage totaling 8KB. To understand how the performance of our real predictor varies with respect to the amount of state space storage, we fix the basic algorithm as shown in Section 4 except we increase the number of slots to meet the total desired space.

By varying the state space from $\frac{1}{2}$KB to 512KB, we find that more than 16KB of storage is wasteful as shown in Figure 18. At 16KB, our predictor achieves nearly 8.5% IPC improvement, whereas at 8KB it achieves 8%. Therefore, the results presented in Section 4 assume the 8KB predictor space even though 16KB would perform slightly better.

The trend of Figure 18 is unusual. With too little information, continuous interference harms performance. At the

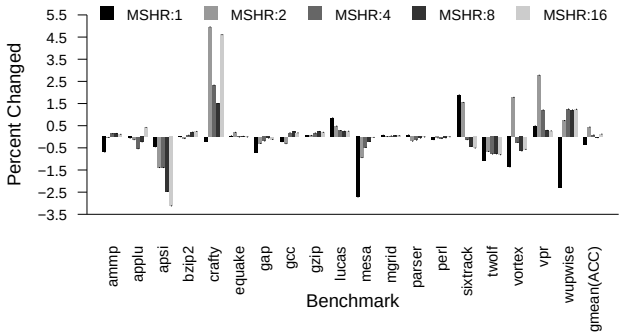

Figure 17: **Percent change in DRAM accesses for SPEC2000, with varying MSHR levels.**

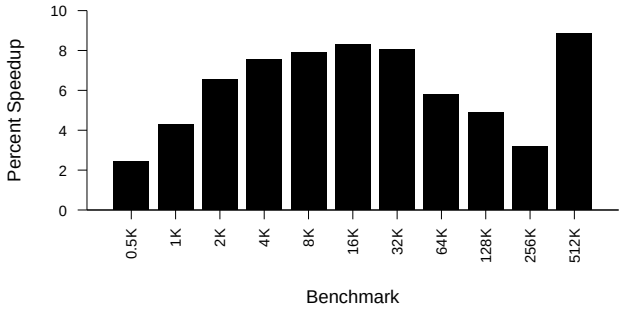

Figure 18: **Percent speedup in geometric mean IPC for the real predictor as state space storage is varied, when compared to the baseline data cache model.**

peak of 16KB, the best results are attained of almost 8.5%. At the extreme end of 512KB storage space, the predictor is able to achieve 9% IPC gain. The drop-off from 32KB through 256KB is an artifact of the tag-bits and steady-state bits being less useful in our predictor. The address replacement, as discussed in Section 4, only occurs when the tag mismatches and the steady-state is empty. When the number of entries in our predictor is sufficiently large, the tag is a series of bits that seldom change causing much thrashing of the steady-state but not actually replacing the address bits of the slot.

However, as shown in Section 5.1, even other simple predictors can beat our real predictor by large margins. This strongly suggests that the limiting agent in performance is our predictor algorithm. Since we can arbitrarily increase the number of slots without substantially improving performance, we are likely not storing the optimal information.

5.6 Alternate Comparisons

While our solution achieves appreciable IPC gain as well as other benefits, we are altering the baseline storage by our addition of the UTI cache. As example alternatives with modern systems, we now show two points of comparison –

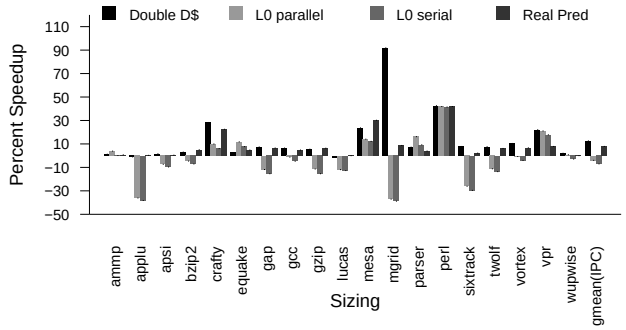

Figure 19: **Percent speedup in IPC for alternate cache designs: the UTI cache as an L0 data cache in parallel or serial to L1, and doubling the base line data cache while keeping access time constant. The real predictor is shown for comparison.**

(1) using an L0 data cache with the same capacity as our UTI cache, and (2) doubling the size of the baseline data cache while keeping the access time fixed. For brevity, we only consider IPC over the SPEC2000 benchmarks here, with the results in Figure 19.

The actual parameters for the L0 cache were swept in a set of simulations, as the optimal UTI cache configuration was determined in Section 5.2. An interesting side effect of this sweep is that the optimal L0 configuration is exactly the UTI cache configuration, 32-way associative with 4-byte line sizes. Optimality was determined by geometric mean IPC over the SPEC2000 benchmarks. On average, the use of an L0 cache performs 8% worse than our predictor.

The drawback to this L0 design is that it implements a serial lookup as a traditional cache structure would. By assuming the L0 is small and low-power, regardless of the real configuration, parallel lookups would be a more efficient solution. Rather than modify Alphasim to support the notion of parallel cache lookups, we instead model a 2KB L0 by reducing the access times to all caches in the serial chain by 1 cycle. The end result is an average IPC reduction of 3% compared to our baseline.

The doubling of the baseline data cache, while keeping access time constant, averaged a 12.5% IPC improvement over baseline. The surprising result is that doubling the data cache size does not uniformly improve the performance of all benchmarks over our UTI cache predictor. We conclude that our "intelligent" management of the cache space achieves better results than just larger caches.

6 Future Directions and Implications

While this work is encouraging, it is far from the end of possibilities for additional research. We lack an "oracle" predictor for a limit study, nor do we study other possible uses for exploiting this UTI cache. Further study may reveal a breakdown on criticality of UTI versus MTI, a new dimension that could provide interesting results.

6.1 Active Research

Presently, we are examining phase behavior over full benchmark lifetimes, as well as studying causes of the access patterns to Stack, Global, and Heap regions in memory. Work on the UTI cache itself is pursuing predictors beyond the most basic, to evaluate whether higher precision may be obtained by more advanced schemes such as combined predictors or the inclusion of additional information such as branch history. Given that the UTI cache is a simple construct, and that the UTI-MTI characterization has shown a strong set of trends based on memory region in accesses, it may be more practical to further break up the monolithic cache structures. Rather than have relatively large L1 caches, a series of small, predicted-access caches that serve specific types of operations may perform better overall while reducing the access times and energy consumption.

6.2 Cache Peeking

Our work assumed that there is no cross-communication between the UTI cache for UTI traffic and the regular L1 data cache. A miss in either directly accesses the unified L2 cache for information; the only "communication" is an invalidate signal that propagates to the corresponding other L1 data cache.

An initial study has shown that up to 85% of UTI cache misses in SPEC2000 are contained in the regular data cache. Instead of always going to the L2 cache, two alternate strategies are: (1) to parallel lookup the L1 data cache and the L2 cache; and (2) to serially lookup in the L1 data cache before checking the L2 cache. Complete analysis of the hit-on-miss rate for the UTI cache to the L1 data cache will reveal which strategy to use. The primary drawback to parallel lookups is the increased power requirements as well as ports required.

6.3 Compilers and Dynamic Optimizations

Many studies in compilers have explored the idea of on-die RAM management, either via scratchpad [2, 19] or such systems as the code cache [9] and software cache [10]. Due to the inability to statically determine where a LD/ST operation will go in memory, such systems have classified data cache management as impractical since the overhead due to instrumentation of LD/ST operations is excessive. By using the concept of UTI/MTI behavior, it may become possible to implement data caching support in a relatively efficient manner. By further exploiting the phasing behavior, full instrumentation can be replaced by simple address tests that

cause exceptions in undesired scenarios.

The spatial locality of UTI targets appears to be quite poor, as explored in Section 5.2. If compilers can keep track of UTI behavior, with or without profiling, they may provide better cache utilization by packing UTI data together. Such a technique would be even more valuable with the ability to "pin" cache lines, such that once the UTI targets are loaded, they are not evicted by polluting MTI traffic. This requires careful analysis of dataflow graphs in the intermediate representation of a compiler, with optional profile information used to capture phasing behavior.

7 Related Work

The architecture community has invested much effort into reducing the memory bottleneck. More outstanding achievements are branch predictors, value prediction, and non-blocking caches. However, we are far from the first group to consider the information content of the reference stream to memory directly. In addition to those works cited earlier in this paper, there are several other groups which have covered some aspects of this work.

Perhaps one of the earliest studies on memory reference behavior, by Hammerstrom and Davidson [8], considered the theoretical amount of information that could be gleaned by data-dependent behavior in reference streams. Using the ideas of entropy and statistical analysis, they found that the addressing overhead is much higher than the actual data content, a result that still holds today as various address compression techniques are now used to reduce power consumption.

Farkas and Jouppi [7] considered the benefits of non-blocking loads, which is the baseline for non-blocking caches. Using a variety of different designs, they were able to reduce miss stalls by up to a factor of 2 for integer applications. Other numeric applications had more substantial gains.

These earlier works are the foundation behind the classification of delinquent load operations – those operations which cause a miss in such a way that performance is dramatically impacted. Even today, the identification and elimination of delinquent loads is a pressing issue [14]. Industry is also exploring mechanisms to avoid these penalties, including Intel's Virtual Multithreading [21] to automatically begin prefetching during delinquent stalls hoping to avoid future stalls.

Tyson et al [18] considered the reference pattern from LD/ST operations, and found that by controlling cache line allocation, memory traffic could be reduced up to 60%. However, Tyson et al were unable to turn this memory pressure reduction into measurable performance improvement.

Tyson and Austin [17] later considered memory renaming, which uses a similar concept to our UTI/MTI predictor. They predicted, also based on the PC, an index to speculative values to accelerate memory operations. Their idea of a *load-store cache* is similar to our isolation of the UTI information, yet our technique is complementary such that combining both methods should attain better results than either alone.

Moshovos and Sohi [12] designed a system to predict and capitalize on dependent memory operations with memory cloaking and bypassing. Their system of reducing the memory latency attained between 3.2 - 4.3% IPC improvements. Our system does not interfere with the cloaking/bypassing technique, yet achieves nearly twice the improvement.

All of these techniques focus on reducing the memory bottleneck from modern processors. Our methods offer a new avenue for exploration, by highlighting the potential exploitation of the dynamic behavior in LD/ST operations. Our methods also appear to be complementary to existing techniques, such that additional gains are possible when our system is applied on top of other methods.

One branch predictor study by Driesen and Hölzle [6] used a similar approach to enumerating the actual targets of instructions. Their scheme relied on preventing the pollution of second-stage predictors by easily-predicted branches. This is similar to our desire to not mix UTI and MTI data, since UTI is stable.

8 Conclusion

We presented a new classification of memory reference behavior based on the target address of each LD/ST operation. We broke our classification into two broad categories: Uni-Targeted Instruction (UTI) and Multi-Targeted Instruction (MTI).

We presented a system using a UTI cache, a small 2KB cache in parallel with the normal L1 data cache. The determination of whether to access the primary data cache or the UTI cache is made by a predictor, based on a decision about the expected UTI or MTI behavior of the generating instruction PC. By using a PC-based (non-critical path) prediction system, we are able to capture the memory target behavior of our study generating substantial IPC gain.

We attain up to 42% IPC improvements over SPEC2000, with an average improvement of 8%. Our solution also reduces L2 accesses by up to 89% (average 29%), while reducing load-load violation traps by up to 84% (average 13%), and store-load violation traps by up to 43% (average 8%).

We attain these results using simplistic finite space predictors. As many of the results suggest, a more sophisticated predictor design may attain even better performance gains.

References

[1] Todd M. Austin, Dionisios N. Pnevmatikatos, and Gurindar S. Sohi. Streamlining Data Cache Access with Fast Address Calculation. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[2] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *10th International Workshop on Hardware/Software Codesign*, May 2002.

[3] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and K.S. Venkatraman. The Microarchitecture of the Pentium 4 Processor on 90nm Technology. In *Intel Technology Journal*, Vol. 8, Issue 1 2004.

[4] Sangyeun Cho, Pen-Chung Yew, and Gyungho Lee. Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor. In *Proceedings of the International Symposium on Computer Architecture*, 1999.

[5] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the International Symposium on Computer Architecture*, 2001.

[6] Karel Driesen and Urs Holzle. Improving Indirect Branch Prediction With Source- and Arity-based Classification and Cascaded Prediction. Technical report, UC-Santa Barbara, March 1998.

[7] Keith I. Farkas and Norman P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. Technical report, DEC, March 1994.

[8] Dan W. Hammerstrom and Edward S. Davidson. Information Content of CPU Referencing Behavior. In *Proceedings of the International Symposium on Computer Architecture*, 1977.

[9] Kim Hazelwood and James E. Smith. Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[10] Chad M. Huneycutt, Joshua B. Fryman, and Kenneth M. Mackenzie. Software Caching using Dynamic Binary Rewriting for Embedded Devices. In *Proceedings of the International Conference on Parallel Programming*, 2002.

[11] Hsien-Hsin S. Lee, Mikhail Smelyanskiy, Chris Newburn, and Gary S. Tyson. Stack Value File: Custom Microarchitecture for the Stack. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2001.

[12] Andreas Moshovos and Gurindar S. Sohi. Speculative Memory Cloaking and Bypassing. In *International Journal on Parallel Programming*, 1999.

[13] Erik Nystrom, Ronald Barnes, Matthew Merten, and Wen mei Hwu. Code Reordering and Speculation Support for Dynamic Optimization Systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[14] Vlad-Mihai Panait, Amit Sasturkar, and Weng-Fai Wong. Static Identification of Delinquent Loads. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[15] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[16] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov 2001.

[17] Gary S. Tyson and Todd M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proceedings of the International Symposium on Microarchitecture*, 1997.

[18] Gary S. Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the International Symposium on Microarchitecture*, 1995.

[19] Manish Verma, Stefan Steinke, and Peter Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Asia South Pacific Design Automated Conference*, January 2003.

[20] Kai Wang and Manoj Franklin. Highly Accurate Data Value Prediction Using Hybrid Predictors. In *Proceedings of the International Symposium on Microarchitecture*, 1997.

[21] Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B. Yunus, Terry Sych, Stephen F. Moore, and John P. Shen. Helper Threads via Virtual Multithreading On An Experimental Itanium 2 Processor-based Platform. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[22] Youfeng Wu, Ryan Rakvic, Li-Ling Chen, Chyi-Chang Miao, George Chrysos, and Jesse Fang. Compiler Managed Micro-cache Bypassing for High Performance EPIC Processors. In *Proceedings of the International Symposium on Microarchitecture*, 2002.