A Case for Exploiting Memory-Access Persistence

Kim M. Hazelwood Mark C. Toburen Thomas M. Conte

Department of Electrical and Computer Engineering
North Carolina State University
{kim_hazelwood, mark_toburen, conte}@ncsu.edu

Abstract

Memory access latencies have become a major bottleneck in the performance of modern computer systems. It has been observed that the SPECint2000 benchmark suite suffers particularly high performance penalties from instruction and data cache misses. Current hardware-based and software-based prefetch mechanisms cannot always learn access patterns early enough to avoid compulsory data cache misses. Memory-access persistence is based on the idea that long-latency memory accesses tend to repeat themselves over subsequent executions of a program, despite changing inputs. By logging long-latency memory accesses during execution and correlating those accesses and their patterns across multiple executions driven with varying input sets, a program can be dynamically optimized to predictively prefetch these memory locations during subsequent executions. In this paper, we determine the extent to which memory-access persistence exists in modern applications.

1. Introduction

Memory access latencies have become a major bottleneck in the performance of modern computer systems. Over the past ten years, advances in process technology, circuit design and architecture have produced an annual increase in processor speed of 60%, whereas DRAM speed has only increased at an annual rate of approximately 10% [1]. Until recently, processor designers have been able to compensate for this annual doubling in the memory gap by simply building larger on-chip caches. However, the effectiveness of this solution is degrading as applications become larger, more complex, and operate on increasingly larger data sets. This places the burden on either predicting compulsory misses or improving the cache miss penalty. Therefore, outside of dramatic improvements in DRAM process technology or the use of SRAM structures for main memory, architectural solutions for bridging the memory gap must continue to be aggressively pursued.

The memory wall problem [1] is exacerbated by streams of compulsory misses that occur when an executing program enters a new section of code, begins processing a large un-cached data structure, or processes a data structure with a non-uniform access pattern such as a dynamically allocated linked list. Current techniques such as software and hardware prefetching can eliminate a large portion of the miss penalty in these cases by obtaining the necessary data from memory in advance of the time the processor needs it [2], [3], [4], [5], [6], [7], [8]. Both hardware and software prefetching provide varying levels of success in overcoming memory access penalties within single executions of a given program. However, the problem with prefetching techniques is that they need predictable access patterns to be successful. When the access patterns are not easy to predict, it is often difficult to speculate the prefetches far enough in advance to cover the memory latency.

Consider the case where clusters of misses are not isolated to a single run of the program with a specific input set. It is possible that a program may generate similar sequences of misses at the same point in execution regardless of the input data set driving the execution. We call this phenomenon *memory-access persistence*. If memory-access persistence truly exists, then there also exists a tremendous opportunity for exploiting it in a manner that will help bridge the memory gap by providing a complementary approach to current latency tolerance mechanisms.

The primary goal of this paper is to demonstrate the degree of persistence that exists within and across program runs using varying input sets. Our second goal is to motivate the idea of exploiting memory-

access persistence through dynamic optimization to help alleviate the memory wall problem. The remainder of the paper is organized as follows. Section 2 discusses related work. Our simulation environment is described in Section 3. Section 4 describes the ideas surrounding memory-access persistence and presents our analysis of the level of persistence that exists in the SPECint2000 benchmarks. Section 5 provides some motivation for using dynamic optimization to exploit memory-access persistence based on the nature of persistence and trends observed in our simulations. Finally, Section 6 concludes the paper and discusses future work.

2. Background

In addition to cache design techniques to increase hit rates, hardware and software prefetching mechanisms are the most widely studied and implemented approaches to avoiding load latencies from main memory. The goal of software prefetching is to insert explicit prefetch instructions at compile time in order to avoid misses of data elements that the compiler feels have a high miss probability based on static analysis and/or statistical profile data [7], [8]. Software prefetching is limited in a number of ways, however. The primary limitation is that it operates with no knowledge of dynamically allocated data structures and where they will reside in memory. While the software prefetching mechanism can accurately insert prefetch operations for statically allocated data structures, such structures are typically not as abundant as dynamically allocated data structures in modern non-numeric applications. The second limitation is that, in a feedback-directed optimization framework, software prefetching is dependent on statistical profiling and is very susceptible to profile drift. The problem with statistical profiling is that prefetch instructions may be inserted into the static code at places where they may only be useful for a relatively small proportion of the dynamic instances of their corresponding load.

Hardware prefetching can complement software prefetching because the hardware mechanisms are capable of learning the access patterns of dynamically allocated data structures. Current hardware prefetching mechanisms can predict what addresses should be prefetched based on the miss address and either a stride- [2], [3] or a history-based context [4], [5], [6]. While the hardware approaches provide excellent speedups in some cases, there are disadvantages. First, they often cannot learn an access pattern well enough ahead of the usage to provide tolerance for long latencies. Thus they can be ineffective against the memory wall. Second, they are forced to relearn access patterns, due to cold starts and drifts in the program's access patterns, each time the program is executed. As a result, they have no ability to track persistence across program runs.

The concept of using dynamic optimization to exploit persistence is aimed at creating a more robust latency tolerance mechanism for data-cache misses. Using dynamic optimization allows us to complement existing prefetching mechanisms by providing the ability to: 1) collect non-statistical profile information that defines the degree of persistence that exists for a given program, and 2) speculatively prefetch well in advance of the usage site, thus hiding longer latencies than existing techniques. Correlation between program runs using non-statistical profiling is similar to cross-program

Table 1 - Dynamic instruction count of benchmarks used.

	test	train	ref
164.gzip	3,703,576,265	63,916,996,696	95,060,008,733
175.vpr	956,521,416	11,098,847,568	84,886,064,416
176.gcc	1,913,767,146	4,828,859,973	42,735,089,367
181.mcf	203,745,510	8,769,967,776	58,711,516,379
197.parser	4,075,314,199	13,121,967,919	531,751,362,392
253.perlbmk	5,238,179	38,306,235,750	42,868,346,148
255.vortex	10,142,404,013	18,641,263,169	124,422,185,624
256.bzip2	15,809,040,703	86,708,522,798	119,414,951,303
300.twolf	296,066,299	15,968,048,185	426,999,118,281

redundancy, a phenomenon exploited in *Slipstream* processors [9]. Slipstream processors use two versions of a program running concurrently but out of synchrony to obtain speedup.

3. Simulation Environment

Results were gathered using the Simplescalar 2.0 *sim-cache* simulator on a Sun Microsystems Ultra-60 running Solaris 7. The *sim-cache* simulator was augmented to produce a trace containing the address and cycle time of all L2 data-cache misses. The SPECint2000 benchmarks shown in Table 1 were run to completion using the official test, train, and reference inputs.

4. Memory-access persistence

Cache misses are a large factor in the performance of modern programs. For example, the cache miss latency of a 500 MHz Alpha 21264 microprocessor is currently 128 cycles [1]. Because of this long latency, it is important to focus research efforts on eliminating as many cache misses as possible. We present one method that uses the notion of persistence to predictively prefetch data from memory. If programs tend to access the same pattern of memory addresses during execution, we say that the access pattern is *persistent*. Studying the persistence of cache accesses throughout and between program executions can help us gain insight into techniques for exploiting this persistence such as dynamic optimization. Because dynamic optimizers can store profile information between program executions, we can leverage these profiles to avoid certain high-latency cache misses in subsequent program executions. Unlike hardware-based prefetching methods, no warmup or training period needs to occur at the start of execution, resulting in immediate benefits. Furthermore, unlike static compiler inserted prefetch instructions, the decision to prefetch can be dynamically changed to adjust for a program phase shift or profile drift.

Memory-access persistence can exist in two forms – *interprogram* persistence and *intraprogram* persistence. A program that tends to access data from the same memory locations (or an offset of the same memory locations) across multiple executions, regardless of the input set, is said to exhibit interprogram persistence. Programs may also exhibit interprogram persistence resulting from accesses to memory locations containing dynamic data structures that are simply allocated to different areas of memory each time the program is executed. In this case, interprogram persistence is determined by the static instructions that result in the data-cache misses. Dynamic optimization and feedback-directed optimization can leverage information regarding interprogram persistence during the optimization process. Intraprogram persistence, on the other hand, occurs when sequences of memory accesses repeat themselves during the course of a single execution of a program. By recognizing intraprogram persistence, a dynamic optimizer or a hardware-based prefetch mechanism can avoid similar miss clusters later in the current execution.

Table 2 – Dynamic miss coverage of the top 10% of miss addresses.

	Test	Train	Ref
164.gzip	37.89%	26.90%	32.28%
175.vpr	39.13%	73.44%	37.44%
176.gcc	39.46%	53.65%	52.66%
181.mcf	10.04%	10.03%	10.02%
197.parser	45.06%	31.39%	47.28%
253.perlbmk	20.39%	82.57%	67.60%
255.vortex	62.54%	59.73%	70.58%
256.bzip2	47.17%	50.48%	52.64%
300.twolf	10.01%	21.21%	23.96%

In order to ascertain whether intraprogram persistence really exists, we studied data cache misses to determine if a certain subset of the miss addresses tended to dominate the total cache misses for a program. We analyzed the top 10% most frequently missed memory addresses and determined the percentage of total data cache misses that were associated with those addresses. Our results are shown in Table 2. In general, Table 2 shows us that there does exist varying amounts of intraprogram persistence within common benchmarks and that there is an opportunity for mechanisms beyond those previously mentioned that can exploit this phenomenon to help bring down the memory wall. As an example, we can see from the table that for 176.gcc using the reference input set, 10% of data miss addresses accounted for 52.66% of the total number of data cache misses. Overall, the dynamic miss coverage ranged from 10% in 181.mcf to 82.57% in 253.perlbmk for the top 10% of miss addresses per benchmark.

Table 3 - Percentage of static address matches across varying inputs.

	test vs. train	test vs. ref	train vs. ref
164.gzip	7.30%	3.69%	50.47%
175.vpr	17.36%	62.40%	27.03%
176.gcc	30.27%	34.20%	81.15%
181.mcf	98.98%	97.06%	98.04%
197.parser	41.33%	27.48%	66.23%
253.perlbmk	0.44%	47.32%	0.21%
255.vortex	99.62%	73.47%	73.47%
256.bzip2	47.32%	23.96%	25.55%
300.twolf	14.38%	7.05%	48.47%

Furthermore, in order to motivate interprogram persistence, we studied the invariance of data cache misses between subsequent executions of a program with varying input sets. Our first test determined the percentage of static miss addresses that two runs of a program had in common. We analyzed the following combinations of the SPEC input sets: (1) test vs. training, (2) training vs. reference, and (3) test vs. reference.

The results of the static address comparisons are shown in Table 3. Table 3 shows significant variation in the percentage of common static miss addresses across runs of the same program. Programs such as 181.mcf and 255.vortex show promising results across all input set combinations. However, the remaining benchmarks show large variations in the number of common addresses as the input set changes, and several benchmarks, 253.perlbmk in particular, show extremely limited amounts of commonality. In these cases however, it is important to note that this test is a conservative indicator of the true amount of interprogram persistence that exists between program executions.

Dynamically allocated data structures are often located at a constant offset to their location from a previous run. Consider the case of a simple program that allocates an array in memory, and then traverses that array. During one execution, that array may be allocated at address x in main memory. A subsequent execution may allocate the array at address $x+n$. While the actual data addresses are not the same, they are at a constant offset to one another. Therefore, interprogram persistence still exists in this case, but in a form called *constant-offset persistence*, which must be detected by a means other than a strict address comparison. The results in Table 3 do not account for constant-offset persistence, but this form of persistence can be clearly seen in Figure 1.

Figure 1 shows memory access patterns over time for each benchmark and input set. Each individual graph provides a snapshot of the memory access patterns for each input set during an isolated time period. It should be noted that the graphs shown in Figure 1 do not reflect the memory access patterns of the entire execution time of each benchmark. Figure 1 is meant to demonstrate the varying forms of persistence and to aid in interpreting the data in Table 2.

Several interesting observations can be made from Figure 1. First, we can plainly see the existence of intraprogram persistence in the graphs for 176.gcc, 253.perlbmk, and 255.vortex. Each of these benchmarks shows repeated misses to the same addresses, especially for the training and reference inputs. This data helps corroborate the data from Table 2. Second, simple interprogram persistence is evident in the graphs for 175.vpr, 176.gcc, 181.mcf, 197.parser, 253.perlbmk, and 300.twolf. Interprogram persistence is especially noticeable in these graphs between the training and reference inputs. This tends to agree with the data in Table 3 with the exceptions being 175.vpr and 253.perlbmk. However, even in these cases the existence of interprogram persistence is clear. Finally, the trend we termed constant-offset persistence is clearly shown in the graphs of 164.gzip and 255.vortex. Both of these graphs show similar miss patterns between input sets but at different addresses.

5. Motivating Dynamic Optimization

In order to exploit memory-access persistence, we need a framework capable of unobtrusively collecting

and analyzing non-statistical profiles during execution and optimizing applications based on this analysis. Dynamic optimizers are well suited for this purpose. In this section we will discuss how the fundamental aspects of interprogram memory-access persistence motivate the use of a dynamic optimization framework to exploit it.

In a dynamic optimization framework, there are two ways to trigger optimization events. One method involves initiating optimization at given execution times. Another method involves correlating optimizations with trigger events such as the processor/system state. The temporal- and event-based nature of interprogram persistence makes the use of dynamic optimization very intriguing. Figure 1 provides insight into the effectiveness of each option based on the trends in persistence of individual benchmarks.

From Figure 1, it can be seen that optimizations triggered by both time and state change events are applicable for reducing data cache misses in SPECint2000. In particular, 175.vpr and 197.parser are very well suited for time-based triggers because their misses to common address between input sets occur at similar points in the dynamic instruction stream. This is evident by the overlapping of all three input sets for these benchmarks. Similarly, 164.gzip and 255.vortex appear to be well suited for time-based optimizations due to the constant-offset nature of their persistence. On the other hand, 181.mcf appears to be a good candidate for state-based optimizations because it exhibits a temporal phase shift in its common miss addresses between the test input set and the other two sets, which overlap.

Despite the trends that are evident in Figure 1, we have seen that the degree and type of persistence that exists within and across benchmarks varies significantly. Therefore, a combination of time- and event-based optimizations will likely be needed to fully exploit memory-access persistence in all cases.

6. Conclusions and Future Work

Reducing memory access latencies has become an important area of research in the microarchitecture community. Prefetching is an effective technique for reducing memory access latencies, but its effectiveness is limited to covering low to moderate latencies. Current hardware- and software-based prefetch mechanisms cannot always learn access patterns early enough to avoid compulsory data cache misses.

Memory-access persistence is based on the idea that long-latency memory accesses tend to repeat themselves over subsequent executions of a program, despite changing inputs. In this paper, we have demonstrated that memory-access persistence exists in two forms: interprogram persistence and intraprogram persistence. Persistence can potentially be exploited by dynamically optimizing programs to predictively prefetch these persistent memory locations. In this manner, we can provide a means for complementing software and hardware-based prefetch mechanisms to subsequently reduce the effects of the memory wall.

Much future work exists in the area of persistence. While this paper points out means for recognizing memory-access persistence, developing a set of algorithms that effectively leverage this persistence information is an important task currently under development.

7. Acknowledgments

This research was supported through equipment and cash donations from Hewlett-Packard Company, Intel Corporation, Sun Microsystems, Compaq Computer Corporation and an NSF CAREER award. We are also grateful to the Slipstream group for their ideas and input on this paper.

8. References

[1] Wilkes, M.V., *"The Memory Gap."* Keynote address, *Workshop on Solving the Memory Wall Problem*, in conjunction with the *27th International Symposium on Computer Architecture*, June 2000.

[2] Palacharla, S. and R. Kessler, "Evaluating Stream Buffers as Secondary Cache Replacement." *Proc. of the 21st International Symposium on Computer Architecture,* April 1994.

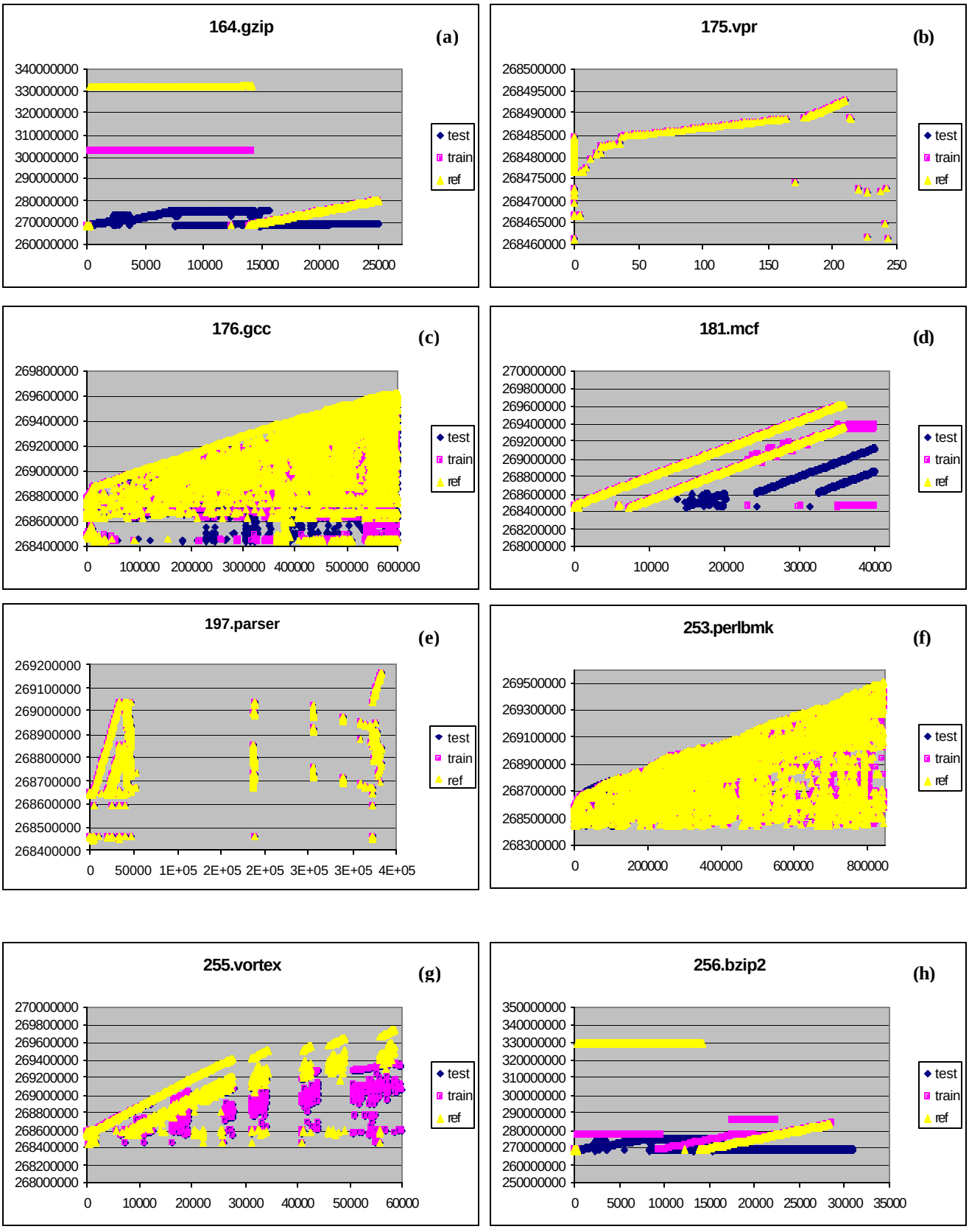

Figure 1 – Memory access patterns for changing inputs of SPECint2000. The x-axis is time measured in cycles. The y-axis shows the data cache miss addresses.

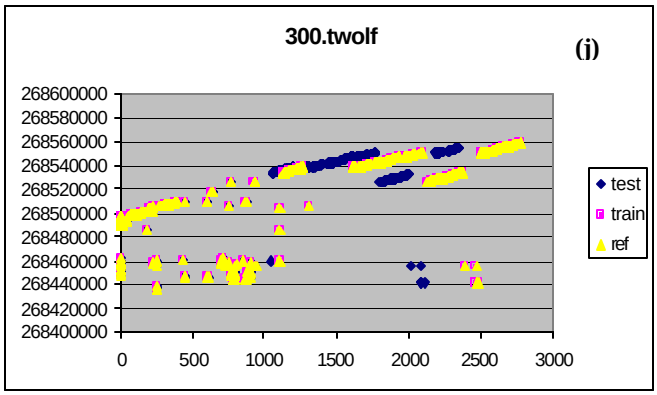

Figure 1 (continued) – Memory access patterns for changing inputs of SPECint2000.

[3] Farkas, K., P. Chow, N. Jouppi and V. Vranesic, "Memory-System Design Considerations for Dynamically-Scheduled Processors." *Proc. of the 24th International Symposium on Computer Architecture*, June 1997.

[4] Charney, M. J. and T.R. Puzak, "Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite." *IBM Journal of Research and Development*, 41(3), May 1997.

[5] Joseph, D. and D. Grunwald, "Prefetching Using Markov Predictors." *Proc. of the 24th International Symposium on Computer Architecture*, June 1997.

[6] Sherwood, T., S. Sair and B. Calder, "Predictor-Directed Stream Buffers." *Proc. of the 33rd International Symposium on Microarchitecture*, December 2000.

[7] Mowry, T.C., M.S. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching." *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[8] Ranganathan, P., V.S. Pai, H. Abdel-Shafi and S.V. Adve, "The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems." *Proc. of the 24th International Symposium on Computer Architecture*, June 1997.

[9] Purser, Z., K. Sundaramoorthy and E. Rotenberg, "A Study of Slipstream Processors." *Proc. of the 33rd International Symposium on Microarchitecture*, December 2000.