Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss

James Dundas and Trevor Mudge
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
{dundas, tnm}@eecs.umich.edu

Abstract

In this paper we propose and evaluate a technique that improves first level data cache performance by pre-executing future instructions under a data cache miss. We show that these pre-executed instructions can generate highly accurate data prefetches, particularly when the first level cache is small. The technique is referred to as *runahead* processing. The hardware required to implement runahead is modest, because, when a miss occurs, it makes use of an otherwise idle resource, the execution logic. The principal hardware cost is an extra register file. To measure the impact of runahead, we simulated a processor executing five integer Spec95 benchmarks. Our results show that runahead was able to significantly reduce data cache CPI for four of the five benchmarks. We also compared runahead to a simple form of prefetching, sequential prefetching, which would seem to be suitable for scientific benchmarks. We confirm this by enlarging the scope of our experiments to include a scientific benchmark. However, we show that runahead was also able to outperform sequential prefetching on the scientific benchmark. We also conduct studies that demonstrate that runahead can generate many useful prefetches for lines that show little spatial locality with the misses that initiate runahead episodes. Finally, we discuss some further enhancements of our baseline runahead prefetching scheme.

1. Introduction

Prefetching data into the data cache is one well-established approach to reducing the detrimental effects memory latency has on processor performance. Prefetching can be accomplished via software, hardware, or a combination of both. Typical software prefetching methods [1][2][3] allow a compiler to use its static knowledge of a program's behavior to generate prefetch instructions. Typical hardware prefetching methods can observe the dynamic nature of a program, and work by either exploiting spatial locality [4][5], or by keeping track of the access patterns for earlier accesses [6][7].

Confining prefetching to software approaches means that the hardware can be kept simple and fast, but prefetch instructions may cause code bloat, and increase register pressure. Existing programs may have to be recompiled to exploit software prefetching. Furthermore, the compiler cannot observe, or take advantage of, the dynamic behavior of programs.

Hardware prefetching can take advantage of information created at run-time and execute existing binaries, however, they can require a significant amount of hardware in order to be effective. The stride and reference prediction tables in [6][7], are a case in point. The hardware requirements can be reduced if the table entries are allocated by the software. This hybrid hardware-software technique was presented in [8]. Their instruction stride table (IST) selectively generates cache miss initiated prefetches for accesses chosen beforehand by the compiler. This resulted in multiprocessor performance for scientific benchmarks comparable in some cases to software prefetching, with an instruction stride table as small as 4 entries. The IST concept was subsequently combined with the prefetch predicates of [2] in [9]. Another hardware prefetching scheme that avoids the need for significant amounts of hardware is the "wrong path" prefetching described in [10]. This actually prefetches instructions from the not-taken path, in the expectation that they will be executed during a later iteration.

Most prefetching techniques, software- or hardware-based, tend to perform poorly on an important class of applications having recursive data structures such as linked-lists. A software technique that overcomes this limitation was presented recently in [11], in which software prefetches were inserted at subroutine call sites that passed pointers as arguments. Another pointer-based approach was described in [12]. This approach uses pointers stored within the data structures to generate software prefetches.

The runahead prefetching approach presented in this paper is a hardware approach, that requires only a modest amount of hardware, because, when a miss occurs, it makes use of an otherwise idle resource, the execution logic. The principal hardware cost is an extra register file. Unlike most of the proposed hardware methods it does not rely on any spatial locality of the data or on the history of previous accesses. It attempts to generate accurate data prefetches by pre-executing future instructions while data cache misses are outstanding. Cache misses resulting from pre-executed loads and stores create the prefetches into the data cache. Because their pre-execution is simply a preview of their actual execution any resulting cache misses are highly accurate, although not perfect as we shall see. The cache misses generated during runahead are placed in a data memory access queue, DMAQ.

We examine two variants of runahead, a conservative method which allows pre-execution to continue only up to the point where branches and jumps cannot be resolved, and an aggressive method which assumes branches and jumps are correctly resolved during runahead. In either case, stores are used as cache touch instructions. They are not allowed to complete and write into memory during runahead.

By waiting until cache misses occur before generating prefetches, runahead adds a highly responsive feedback component to the memory hierarchy: the greater the cache miss penalty, the more opportunities there are for prefetching, which tend to reduce the frequency of future cache misses. Conversely, if an application enters a phase where its hit rate is high, few prefetches are generated.

The paper is organized as follows. The next section provides more details on the mechanism of runahead prefetching. Section 3 discusses the simulator, benchmarks, and experimental results. Section 4 presents some concluding remarks and discussion of future work.

2. How Runahead Works

When the processor detects an L1 data cache miss it records the instruction address of the faulting load or store instruction and enters runahead mode. It also checkpoints the register file, RF, by copying it to a backup register file BRF. The processor then pre-executes subsequent instructions while the cache miss is serviced. Because the value returned from a cache miss cannot be known ahead of time, it is possible for pre-executed instructions to be dependent upon invalid data. Rather than terminating runahead we allow registers and data cache values to have an explicit "invalid" state during runahead. Denoting this value, INV, requires an extra bit associated with each register in RF as well as with each word in the L1 data cache line (if byte addressing is allowed, each byte requires an invalid bit). Pre-execution of most instructions consists of the usual steps of fetch, decode, and execute, with some changes to deal with invalid data. Also stores are treated slightly differently. The actions associated with pre-execution can be summarized by instruction type as follows:

1. **register-to-register** instructions mark their destination register INV if any of their source registers contain an INV. They can also replace an INV value in their destination if all sources are valid.
2. **load** instructions mark their destination register INV if either of three cases arises:
   i. if the base register used to form the effective address is marked INV, or
   ii. if the base register is not INV but the load causes a cache miss, or
   iii. if the base register is not INV and a cache miss does not occur but the target word in the cache is marked INV as a result of a preceding store during the same runahead episode (see next case).

   They can also replace an INV value in their destination if none of the preceding three cases apply.
3. **store** instructions do not write data into the cache or main memory. They do, however, mark the referenced L1 data cache item INV, if the base register used in address calculation is not INV and a cache miss does not occur.
4. **conditional branch** instructions are resolved normally if their branch condition is not INV. If it is, the outcome is determined by whatever branch prediction strategy the processor employs.
5. **jump register indirect** instructions (the target of the jump is obtained from a register) in which the register contains an INV value assume that the return stack contains the address of the next instruction.

The above actions were formulated from straightforward considerations of read-after-write dependencies, however they do not always accurately anticipate what occurs during actual execution. Action 3 does not account for the case when stores cause a cache miss or cannot compute their target addresses because their base register has an INV value. Such stores cannot mark their destination word INV. It thus follows that subsequent loads have a small possibility of introducing apparently valid data into the RF, which should have been marked INV. Action 4 does not account for the case when an unresolvable conditional branch is mispredicted. Finally, action 5 does not account for the case when jump indirects are used to provide dynamically relocatable subroutine call linking rather than subroutine return linking.

To summarize, the above runahead pre-execution actions result in values in the RF that cannot be trusted with certainty during runahead: there is a small possibility that a valid register should be marked INV and vice versa. Because the values in RF during runahead are only used to create runahead data prefetches, they do not affect the sequential state of the processor. The worst thing that can happen is that some number of useless prefetches can be generated.

After the cache miss that started runahead mode is serviced the processor resumes execution at the faulting instruction, and RF is restored from its backup, BRF. Each register in the BRF need only be connected to the one RF register that it shadows. No read or write ports connect to the BRF, simplifying its implementation, keeping it off of the critical path, and allowing the RF $\leftrightarrow$ BRF transfers to be done via two global signals. At most a cycle will be added to the miss penalty. Load and store instructions, that can compute their target addresses with valid registers, can generate prefetch requests if their target lines are not in the L1 data cache. The valid bits in RF and the cache are set to the valid state when the processor leaves runahead.

2.1 Two runahead policies

We consider two runahead policies in this paper: a conservative policy and an aggressive one. If a conditional branch or jump is pre-executed that is dependent upon an invalid register, the processor can do one of two things.The conservative policy halts runahead until the runahead-initiating miss is serviced. The aggressive policy assumes that the branch prediction or subroutine call return stack is good enough to accurately resolve the branch or jump. The conservative policy should generate fewer useless prefetches, but the aggressive policy can potentially deliver superior performance if it can continue to pre-execute the proper instructions past unresolvable branches and jumps. Our simulations of the aggressive runahead policy assume that the processor always stays on the proper path of execution during runahead.

3. Experiments

The simulator was created using ATOM [13]. The simulated processor fetches, decodes, and executes one instruction per cycle. It did not model pipeline stalls or penalties due to instruction cache misses, just the effects of data cache misses and any resulting prefetching. Thus the CPI figures obtained in the experiments is simply the contribution due to data cache effects, dcache-CPI, plus one. Effects such as page faults and context switches were not modeled, and the data caches were cold started.

A total of six benchmarks from the Spec95 suite were simulated, and are described in Table 1. All of the benchmarks were compiled using version 2.7.2 of gcc with the -O optimization level.

In the simulations an 8 entry DMAQ is used to send miss and prefetch requests, as well as store-throughs, from an on-chip L1 data cache to an off-chip L2 data cache strictly in the order in which they were generated. The DMAQ thus provides the functionality of a store queue, outstanding request list [6], and miss status holding registers [14]. It is further assumed that the DMAQ cannot coalesce store-throughs, or allow demand fetches or store-throughs to either pass or squash outstanding prefetches. The processor stalls for store throughs and demand fetches when the DMAQ is full, while prefetches that are generated when the DMAQ is full are dropped. In summary, we simulate a model of the interface to the off-chip memory hierarchy that is both simple and conservative. A description of the data memory hierarchy is given in Table 2.

3.1 Results of using runahead for data prefetching

Plots of data memory CPI versus L1 data cache size are shown in Figure 1 through Figure 5. These simulations were allowed to run for the first 500 million instructions of each benchmark.

There are three data sets shown in each of the figures. No prefetch corresponds to the simulated processor without any data prefetching. The other two data sets correspond to the conservative and aggressive runahead policies. The percentage reduction of dcache-CPI from that of the no-prefetch scheme is provided next to each data point. (Recall that the data memory portion of the CPI figure is simply one less than the overall CPI that was measured.)

Conservative runahead reduced performance by 4% (0.016 CPI) for the larger 8KB L1 data cache on the Compress benchmark (Figure 1). This can be attributed to the simple model assumed for the memory interface which allows store-throughs, demand fetches, and a small number of runahead prefetches to compete for DMAQ entries on an equal basis. Interestingly, while the aggressive runahead strategy was able to consistently improve performance over the entire range of cache sizes, conservative runahead was able to outperform aggressive runahead for the 1KB L1 data cache size. The small cache results in increased opportunities for prefetching, particularly for aggressive runahead, resulting in more competition for DMAQ entries. Conservative runahead generates fewer prefetches, resulting in less competition. A more advanced interface to the off-chip memory hierarchy which prioritizes the off-chip memory references would improve performance by reducing the competition between prefetches, demand fetches, and store-throughs.

The rest of the benchmarks were able to benefit significantly from runahead prefetching. Note that the dcache-CPI for the no prefetch scheme is significant: all of the simulations have a dcache-CPI of about 2 for a 4KB L1 data cache, with the exception of Ijpeg. Go (Figure 2) benefited the most from the aggressive runahead scheme, with reductions in dcache-CPI of 58 to 43% for the aggressive runahead scheme. The conservative runahead scheme delivered performance reductions half that of the aggressive runahead scheme, indicating that a significant number of unresolvable branches and jumps were encountered. Perl (Figure 3)had reductions of dcache-CPI of 49 to 22% for the aggressive scheme and 35 to 15% for the conservative scheme. The dcache-CPI reductions for Vortex (Figure 4) were similar with 47 to 26% for the aggressive scheme, and 33 to 18% for the conservative scheme. Ijpeg (Figure 5) was perhaps the most interesting of the integer benchmarks, with reductions of dcache-CPI of 44 to 31% for the aggressive scheme, and nearly identical reductions of 43 to 30% for the conservative scheme. This indicates that virtually all of the branches and jumps encountered during runahead could be resolved with valid registers.

3.2 Comparison with sequential prefetching

The following simulations were performed with the L1 data cache size fixed at 4KB, and were only simulated for the first 100 million instructions of each benchmark.

Simulations of the *prefetch-on-miss* and *always-prefetch* sequential prefetching techniques described in [4] were performed. Prefetch-on-miss generates a prefetch for line $i+1$ whenever an access of line i misses in the L1 data cache. Always-prefetch generates a prefetch for line $i+1$ whenever line i is accessed. Both techniques were also simulated with the ability to generate prefetches for line $i+2$ in addition to line $i+1$, giving us a total of 4 sequential prefetching schemes. Prefetches that hit in the L1 data cache were dropped.

Bar graphs are shown in Figure 6 through Figure 11, which compare the dcache-CPI reducing ability of the four types of sequential prefetching to that of aggressive and conservative runahead prefetching. The percentage reduction of dcache-CPI over the baseline no-prefetch scheme is provided on top of each bar. Performance degradation is indicated by enclosing the percentage in parentheses. In general, sequential prefetching works well for scientific benchmarks but poorly for integer benchmarks. Because the benchmarks we are using are integer benchmarks, we added a scientific benchmark, Tomcatv, to provide a comparison with at least one benchmark for which sequential prefetching schemes generally work well.

Figure 6 shows results for Ijpeg. This is the one case where both prefetch-on-miss schemes were able to improve performance to a small extent (dcache-CPI decreased by 7 and 3%), however, this was greatly overshadowed by that of conservative and aggressive runahead prefetching where the dcache-CPI was decreased by 26 and 27%, respectively. The always-prefetch schemes significantly degraded performance (dcache-CPI increased by 34 and 49%).

As we expected, Tomcatv (Figure 7) benefited from sequential prefetching. The always-prefetch and prefetch-on-miss schemes were able to improve performance (dcache-CPI decreased by 7 and 8%) when prefetches were generated for only the immediately sequential line. When up to two lines could be prefetched, performance was reduced for both always-prefetch and prefetch-on-miss (dcache-CPI increased by 28 and 25%). The conservative and aggressive runahead schemes were able to reduce data dcache-CPI significantly (dcache-CPI decreased by 25 and 24%), outperforming all of the sequential prefetching schemes.

In the remainder of the benchmarks (Figure 8 through Figure 11) the conservative and aggressive runahead schemes were able to reduce dcache-CPI significantly, with the exception of compress where gains were modest. Sequential prefetching resulted in reduced performance for all of these benchmarks.

3.3 Measurements of miss-prefetch spatial locality

In order to get a better picture of why runahead is effective for integer benchmarks we measured the miss-prefetch spatial locality provided by runahead for our benchmarks. Again our simulations ran for the first 100 million instructions of each benchmark with the L1 data cache fixed at 4 KB.

Plots of the miss-prefetch spatial locality for the aggressive runahead policy are shown in Figure 12 through Figure 17. The x = 0 point on the x-axis represents the cache line address of each runahead-initiating L1 data cache miss. The other points on the x-axis represent the distance in cache lines between the address of each runahead-initiating miss and any runahead prefetches that are generated during the corresponding runahead episode. For example, x = 1 represents the address of the next sequential line after the address that caused the runahead-initiating miss. The y-axis is a cumulative value. Its value at a particular x, say 35, represent the fraction of all runahead prefetches that result from pre-executed accesses to any of the 35 sequential line addresses after the address causing the miss that initiated runahead. The negative part of the x-axis represents addresses before the runahead-initiating miss. The extreme data points at x = +/- 1000 lines represent the fraction of runahead prefetches whose addresses correspond to lines that are at least 1000 lines distant from their corresponding runahead-initiating misses. The plots in Figure 12 through Figure 17 are aggregates of all the runahead prefetches that are generated over the course of running each benchmark.

For example, by looking at the data points for positive x close to x = 0 in Figure 12, one can see that 18% of all runahead prefetches generated for Vortex are for the 12 immediately sequential cache lines after their runahead-initiating misses. Similarly, 8% are generated for the 12 lines preceding their runahead-initiating misses. The cumulative distributions are relatively flat until they reach x = 1000, at which point there are lage spikes. The spike at x = +1000 indicates that 27% of the runahead prefetches are generated for lines whose addresses are more than 1000 lines ahead of their runahead-initiating misses. Another 38% are generated for lines whose addresses are more than 1000 lines before their runahead-initiating miss.

Go (Figure 13) exhibited even less miss-prefetch spatial locality than Vortex. Only 15% of the runahead prefetches are generated for addresses within 100 lines of their runahead-initiating miss, and only 29% of the runahead prefetches are within 999 lines. The majority of the prefetches are for lines far removed from their runahead-initiating misses. Perl (Figure 14) exhibited somewhat more miss-prefetch spatial locality than that for Go. 16% of the runahead prefetches are for lines within 10 lines their runahead-initiating misses, two-thirds of which are for lines located before their corresponding misses. The overwhelming majority of the prefetches, 73%, are for lines far removed from their runahead-initiating misses. Compress (Figure 15) exhibited by far the best miss-prefetch spatial locality of the benchmarks that we examined, with 85% of the prefetches within 25 lines of their runahead-initiating misses. Note that two-thirds of these prefetches were for lines immediately preceding their runahead-initiating misses. Ijpeg (Figure 16) exhibited the least miss-prefetch spatial locality of the benchmarks that we examined, with very few prefetches generated close to their runahead-initiating misses. However, 85% of the prefetches were generated within 500 lines.

Tomcatv (Figure 17) shows somewhat more miss-prefetch spatial locality than Ijpeg for lines close to their runahead-initiating misses. The majority of the prefetches generated for Tomcatv were for lines far removed from their runahead-initiating misses.

These results indicate that runahead is very effective at generating cache-miss initiated prefetches that exhibit little spatial locality with respect to their prefetch-initiating misses. Sequential prefetching can only generate prefetches for lines that exhibit spacial locality.

3.4 What happens to potential prefetches

Many of the pre-executed load and store instructions that would otherwise miss in the L1 data cache are unable to generate prefetches for a variety of reasons:

- a demand fetch or prefetch for the missing line is already in progress, or
- the pre-executed load or store instruction could not generate its target address with valid registers, or
- the DMAQ was full, causing what would have been a useful prefetch to be dropped.

A bar graph illustrating what happened to the potential prefetches for each benchmark is shown in Figure 18. Only the aggressive runahead policy is shown. The numbers on top of each bar represent the total number of potential runahead prefetches (pre-executed loads and stores whose actual target addresses would cause a miss in the L1 data cache) for the benchmark in question. Note that since instructions can be pre-executed multiple times in successive runahead episodes it is possible for the number of potential prefetches to exceed the number of instructions actually executed in the simulation (100 million).

From the figure it can be seen that 23 to 46% of the potential runahead prefetches were for lines that were already being fetched by an earlier prefetch or runahead-initiating miss. Another 12 to 37% of the potential runahead prefetches could not compute their target addresses with valid registers. Of the remaining potential prefetches, up to 38% were dropped due to a lack of available DMAQ entries. The remainder of the potential runahead prefetches, 19 to 40%, were actually able to become useful runahead prefetches. This strongly suggests that separate queues are needed for runahead prefetches and store-throughs, and that a runahead processor should avoid dropping prefetches, perhaps with a longer prefetch queue.

The benchmarks that had the most potential runahead prefetches (Vortex, Perl, and Tomcatv) also lost the largest proportion of them to a lack of DMAQ entries. The rest of the benchmarks lost very few potential prefetches in this way. Tomcatv also lost the smallest fraction of its potential prefetches to invalid registers, with just over 33 million prefetches generated during the first 100 million instructions of the benchmark. This huge number of prefetches for Tomcatv caused by far the largest absolute reduction in dcache-CPI among the benchmarks that we examined. This suggests that runahead may be very effective for both integer and scientific benchmarks in general.

3.5 The effect on off-chip fetch traffic

Off-chip fetch traffic can lead to performance problems if the memory hierarchy does not have adequate bandwidth to service the requests in a timely fashion, or if a significant fraction of the fetch traffic is composed of useless prefetches. Runahead prefetches are by nature quite accurate, so any performance degradation attributable to runahead is likely to arise from the sheer quantity of additional fetch traffic that may occur. A bar chart is shown in Figure 19, which depicts the number of fetches that are generated during the simulation of the first 100 million instructions of each benchmark. The number of fetches for each benchmark is broken down into three bars, which depict the number of fetches for the no-prefetch, conservative runahead, and aggressive runahead schemes. Each bar is broken down further into demand fetches (those that are the result of a conventional load or store miss) and runahead prefetches.

The percentage increase in fetch traffic for each runahead scheme over that of no-prefetch is indicated as a number on top of each bar. Note that all of the integer benchmarks incurred small increases in fetch traffic due to runahead. Only one of the integer benchmarks, Perl, suffered an increase in fetch traffic of more than three percent for either of the runahead prefetching schemes. Furthermore, the number of demand fetches was significantly reduced for most of the benchmarks when runahead was employed.

The only scientific benchmark that we simulated, Tomcatv, incurred large increases in overall fetch traffic as a result of runahead (112.6 and 114.3%). However the number of demand fetches was dramatically reduced, and Tomcatv was able to derive the largest absolute reduction in dcache-CPI of all the benchmarks that we examined.

4. Concluding Remarks

In this paper we present runahead pre-execution, a new method of improving several aspects of processor performance. Our simulations indicate that runahead data prefetching is effective at improving the performance of integer benchmarks. Simulations of a scientific benchmark, Tomcatv, indicate that runahead data prefetching may also be effective for benchmarks that work well with traditional hardware and software data prefetching methods. The experiments were conducted for a range of L1 data cache sizes that are quite small, as this was the range that we were interested in as part of our studies of the design of a very high-clock rate GaAs processor. Clearly, experiments need to be done for L1 caches larger than 8 KB.

There are a number of areas where runahead has the potential to improve performance that were not examined in this study. For example, runahead can improve branch prediction performance by pre-executing conditional branches. A shift register can be used to record the outcomes of pre-executed conditional branches. A count register is needed to keep track of how many valid outcomes are in the shift register. A second count register and two bits of state are needed to determine when to allow the processor to add or remove outcomes to or from the shift register. When the processor leaves runahead mode any outcomes that were recorded in the shift register are used as branch predictors by shifting them out one at a time for each conditional branch that is encountered. A similar approach can be used to record pre-executed indirect jump targets. Pre-executed conditional branch outcomes can also be used to train a traditional dynamic branch prediction scheme during runahead episodes, which is used to predict conditional branches when the shift register is empty.

Some preliminary runahead branch prediction simulation results were reported in [15]. Although the initial results were encouraging, the fact that the simulator could not explore wrong paths during runahead meant that the results were somewhat optimistic. We will present more accurate results once we have a simulator that can explore wrong paths.

Some performance is lost since runahead stores cannot modify the L1 data cache, which can cause read-after-write dependencies with subsequent loads during the same runahead episode. A small fully associative store data cache may be used to capture store data in runahead, which could then be used by subsequent runahead loads. This cache can be quite small yet still deliver a noticeable improvement in performance. This approach would be most useful when runahead crosses procedure boundaries or whenever registers are spilled, both of which can cause a significant number of stack-based loads and stores.

Finally, a runahead processor needs a steady supply of instructions to pre-execute in order to be effective. Although instruction cache misses that are detected during runahead are desirable in the sense that they become instruction stream prefetches, it may be possible to obtain additional performance by continuing to pre-execute instructions after an instruction cache miss is detected, or by initiating runahead on an instruction cache miss. We leave this for future work.

5. References

[1] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," In the Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.

[2] T.C. Mowry, M.S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," In the Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992.

[3] A.C. Klaiber and H.M. Levy, "An Architecture for Software-Controlled Data Prefetching," In the Proceedings of the 18th Annual International Symposium on Computer Architecture, 1991.

[4] A.J. Smith, "Cache Memories," ACM Computing Surveys, vol. 18, num. 3, September 1982.

[5] N.P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," In the Proceedings of the 17th Annual International Symposium on Computer Architecture, May 1990.

[6] J.L. Baer and T.F. Chen, "An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty," In the Proceedings of Supercomputing, 1991.

[7] J.W.C. Fu and J.H.Patel, "Stride directed prefetching in scalar processors," In the Proceedings of the 25th International Symposium on Microarchitecture, December 1992.

[8] R. Bianchini and T.J. LeBlanc, "A Preliminary Evaluation of Cache-Miss-Initiated Prefetching Techniques in Scalable Multiprocessors," University of Rochester Computer Science Department Technical Report 515, May 1994.

[9] T.F. Chen, "An Effective Programmable Prefetch Engine for On-Chip Caches," In the Proceedings of the 28th International Symposium on Microarchitecture, 1995.

[10] J. Pierce and T. Mudge, "Wrong-Path Instruction Prefetching," In the Proceedings of the 29th International Symposium on Microarchitecture, 1996.

[11] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger, "SPAID: Software Prefetching in Pointer- and Call-Intensive Environments," In the Proceedings of the 28th International Symposium on Microarchitecture, 1995.

[12] C.K. Luk and T.C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," In the Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.

[13] A. Eustace and A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools," Digital Equipment Corporation Western Research Laboratory Technical Note TN-44, July 1994.

[14] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," In the Proceedings of the 8th International Symposium on Computer Architecture, May 1981.

[15] J. Dundas and T. Mudge, "Using Stall Cycles to Improve Microprocessor Performance," University of Michigan Department of Electrical Engineering and Computer Science Technical Report CSE-TR-301-96, September 1996.

Benchmark (Spec95)	Data Set	Load/Store Data References		Description
		First 100M Instr.	First 500M Instr.	
Vortex	reference	23.4M/17.7M	121.3M/85.9M	Database program
Go	play = 10 board = 13	30.1M/6.4M	159.8M/36.6M	Artificial intelligence program that plays Go
Ijpeg	reference	20.0M/10.7M	91.4M/36.4M	Image compression and decompression
Perl	reference	24.2M/12.6M	120.7M/62.6M	Manipulates strings and prime numbers using Perl
Compress	reference	24.8M/10.9M	124.0M/54.9M	Compresses and decompresses a file in memory
Tomcatv	reference	22.9M/10.9M	Not simulated	Mesh generation (scientific benchmark)

Table 1 Benchmark Descriptions

Policy	L1 Data Cache	L2 Data Cache	Main Memory
Placement	direct-mapped	direct-mapped	N.A.
Store-hit	write-through (up to 8 outstanding)	write-back	N.A.
Store-miss	write-allocate	write-allocate	N.A.
Line / transfer size	32 B	32 B	32 B
Size	4 KB	1 MB	N.A.
Blocking	Non-blocking (up to 8 outstanding)	Hit-under-miss only	N.A.
Maximum throughput	1 accesses/cycle	0.2 accesses/cycle	0.01 accesses/cycle
Latencies (cycles)	Hit: 0 (covered by pipeline model) Miss: 24 + time in DMAQ	Hit: 0 (covered by L1 miss) Miss: additional 100 (200 if Dirty)	0 cycles (covered by L2 miss and dirty WB)

Table 2 Data Memory Hierarchy Parameters

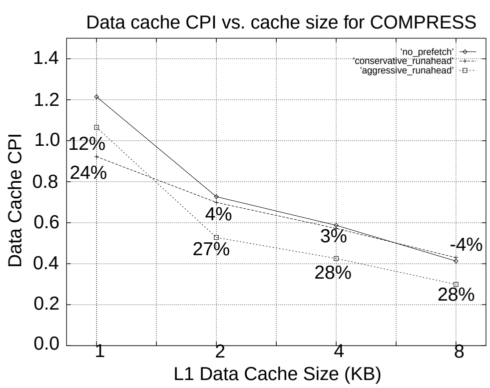

Figure 1

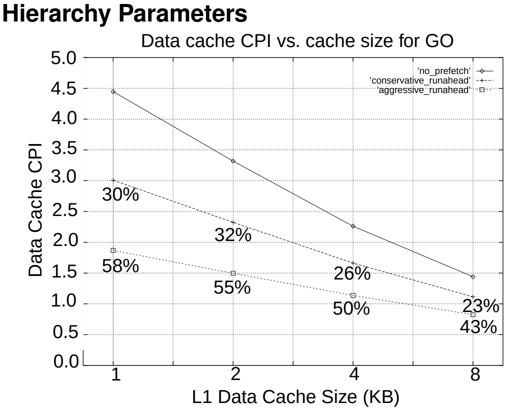

Figure 2

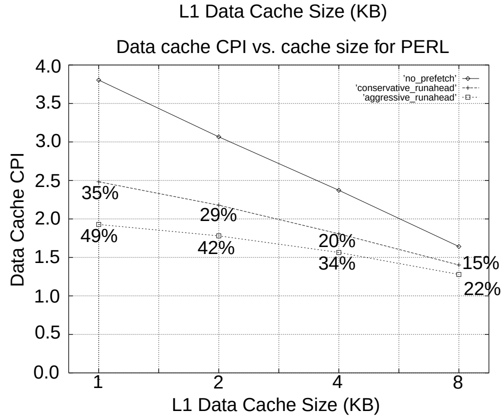

Figure 3

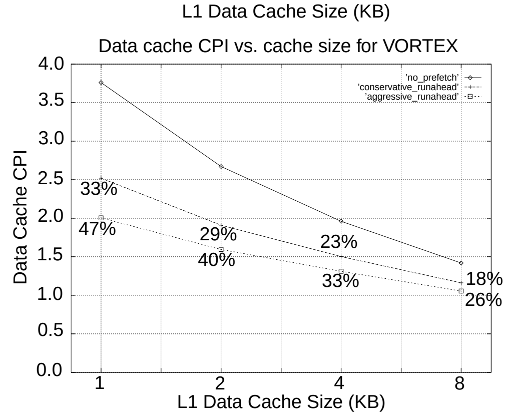

Figure 4

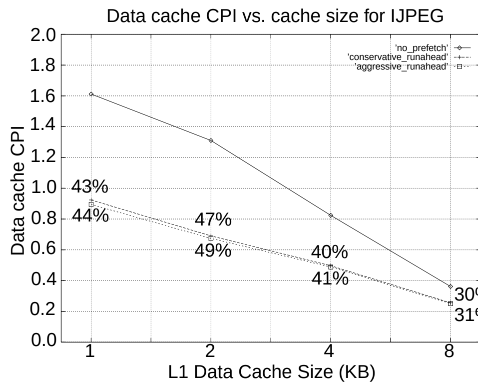

Figure 5

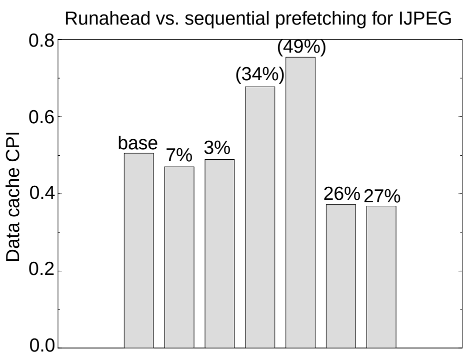

Figure 6

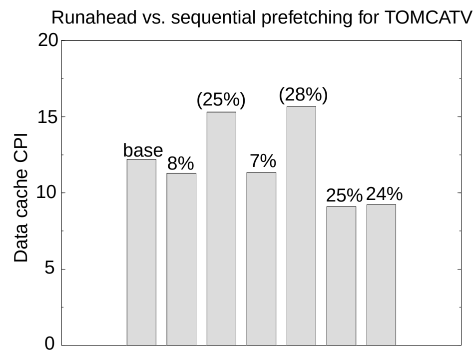

Figure 7

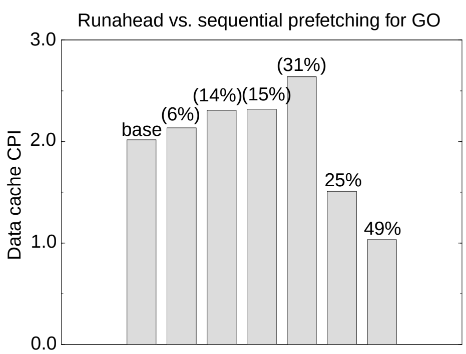

Figure 8

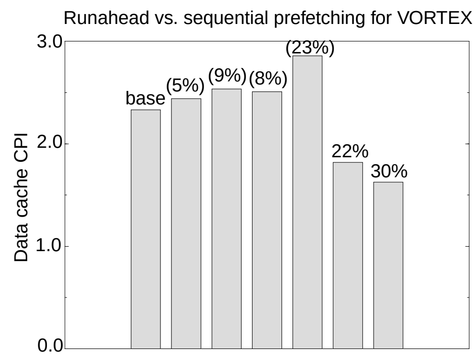

Figure 9

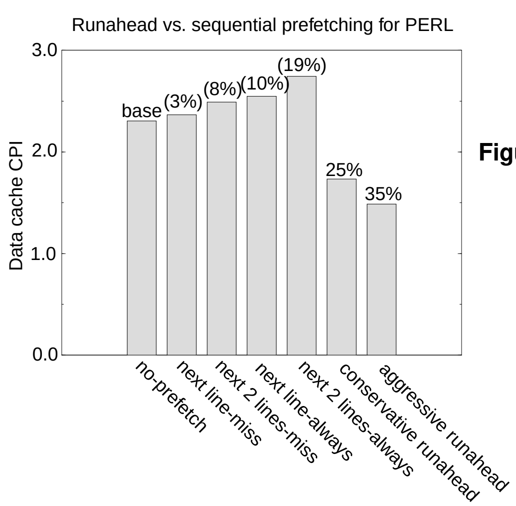

Figure 10

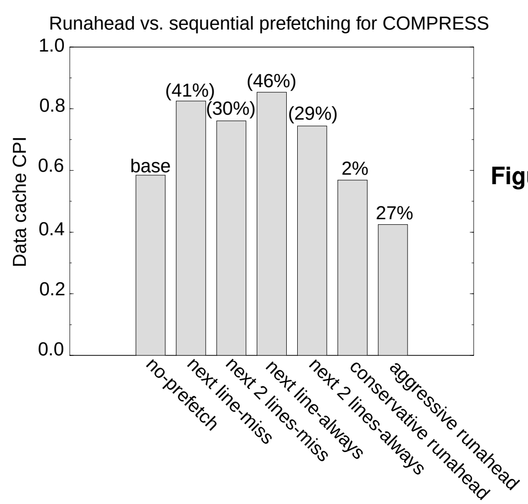

Figure 11

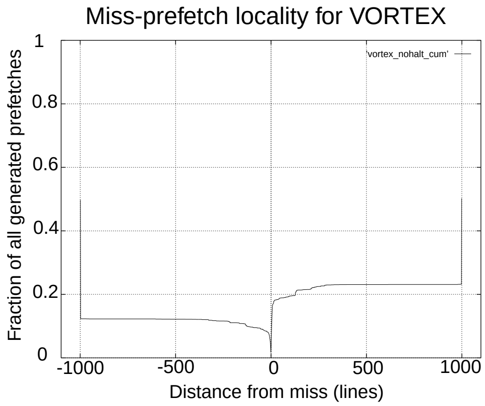

Figure 12

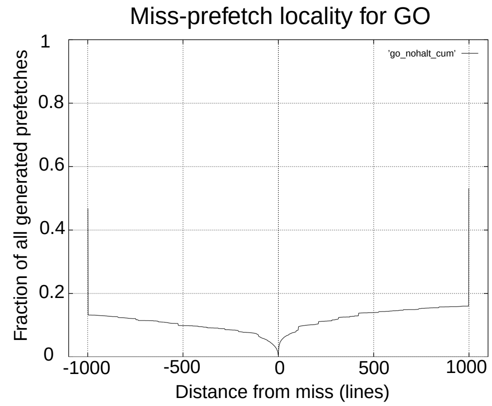

Figure 13

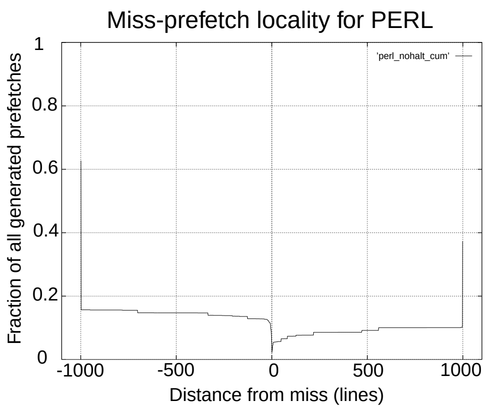

Figure 14

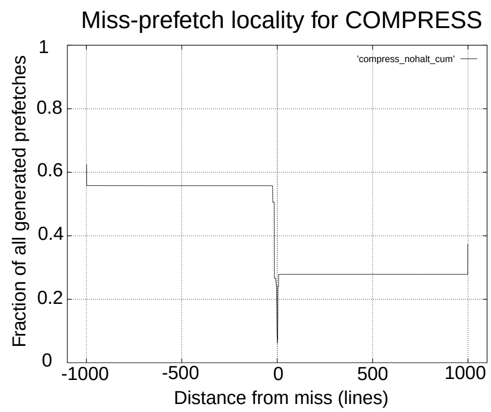

Figure 15

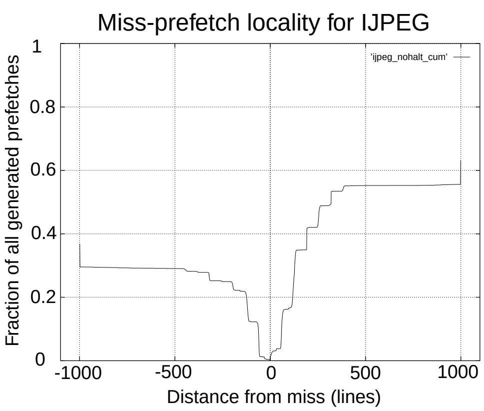

Figure 16

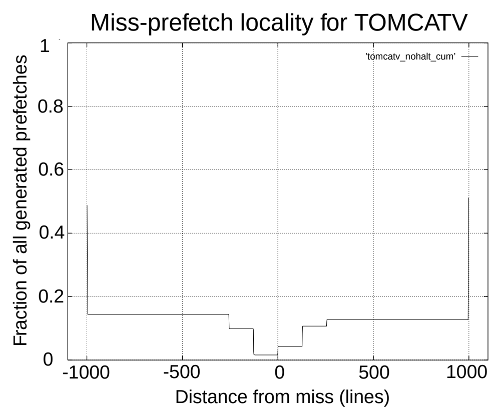

Figure 17

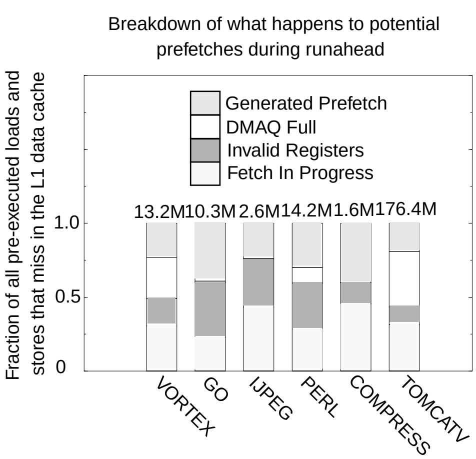

Figure 18

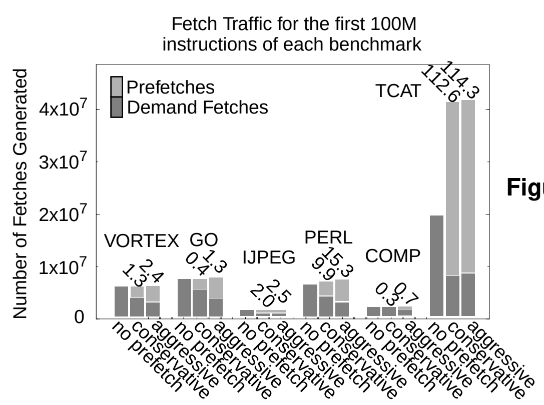

Figure 19