Feedback-Directed Data Cache Optimizations for the x86

Ronald D. Barnes[*] Ronnie Chaiken David M. Gillies
Microsoft Research
One Microsoft Way, Redmond WA, 98052
rdbarnes@crhc.uiuc.edu, {rchaiken, dgillies}@microsoft.com

Abstract

The vast majority of desktop microprocessors in use today belong to a single architectural family, the x86. The success of this architecture has led to a large number of microarchitectures and a growing need to evolve the ISA to meet the changing demands of applications. Unfortunately, most compiled code today targets the 486 or Pentium® ,[¤] thereby missing performance opportunities of newer processors. We believe that specialized compilation within the x86 family can yield large performance gains over generically compiled code. This paper examines the effectiveness of the Pentium III data cache management instructions on desktop applications. We use a memory trace analysis in our optimization to guide the placement of cache prefetch and cache bypass instructions at the binary-level. Our results show that significant performance improvements can be achieved for a wide range of applications.

1 Introduction

Since the advent of the 386, the number of implementations within the x86 family of microprocessors has grown very rapidly. These implementations run the gamut of modern computer architecture: non-pipelined to pipelined, scalar to superscalar, in-order to out-of-order, cacheless to having a highly stratified cache structure. At the same time, the ISA has been evolving to meet the changing demands of applications. ISA enhancements include support for 32 bit addressing, partial instruction predication, multimedia extensions with SIMD support, and an increase and widening of the register set. In addition, the circuitry surrounding the CPU has changed. For example, the sizes of off-chip cache and main memory have increased many-fold. Factoring the contributions of several different manufacturers into the mix has made for a very diverse landscape within the x86 family.

Unfortunately, most code intended for the x86 platform is compiled targeting the 486 or Pentium thereby missing performance opportunities of newer processors. There are several reasons for this choice. In order to deliver software that runs on the full range of x86 platforms one cannot make assumptions about the existence of special new instructions or try to exploit a new microarchitectural feature. Also, due to the ever-tightening product development cycle time, testing multiple executables has generally prevented machine-specific versioning. This limitation has been overcome in a very small number of cases by the conditional execution of specialized code within dynamically linked libraries. These examples are often handwritten in assembly language to maximize the performance of a few critical routines. However, this is far from a complete solution for general programs.

While there are a number of pragmatic reasons for adopting the one-size-fits-all approach to software delivery, we contend that significant potential performance gains offered by variations in x86 platforms are being overlooked. In this paper we demonstrate one area for performance improvement in this space. Specifically, we make use of the Pentium III's data cache management instructions to illustrate the gains that are available in general programs. Our optimization heuristics improve data cache performance by inserting prefetch and bypass instructions directly into the binary, through binary rewriting. The heuristics use information that is gathered from a cache simulation that consumes memory traces on the fly.

The remainder of this paper is divided into five sections. Section 2 reviews previous related work in this area. Next we describe the methodology used in our experiments. Section 4 describes the Pentium III instructions used in this study and the heuristics used to drive the optimizations are described in Section 5. Our results are presented in Section 6 and the final section contains some concluding remarks.

2 Previous Work

Previous work advocating processor-specific optimizations on x86 processors has been done by Merten[13]. Much of this work focused on a framework that enables optimizations, rather than on the optimizations themselves. In addition, the optimizations presented were mostly ad-hoc pattern matching. In this work, we present a trace-directed analysis to guide the placement of data cache management instructions.

---

[*] Ronald D. Barnes is currently with the IMPACT Research Group in the Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, Illinois 61801.

[¤] Intel, Pentium, MMX and Pentium® III Xeon are either registered trademarks or trademarks of Intel Corporation in the United States and/or other countries.

Table 1. Cache specifics of the target Pentium III Xeon machine.

L1 Cache Size	16kB
L1 Associativity	4-way
L1 access latency	1 cycle (assumed)
L2 Cache Size (instruction & data)	512kB
L2 access latency (measured)	~20 cycles
Main memory latency (measured)	~65 cycles
Cache Replacement	LRU (assumed)
Cache block size	32 bytes

The performance limitations caused by the ever-widening gap between processor and main memory speed are well understood, and hierarchical caches have been used to ameliorate this effect. Methods for instruction cache optimizations, like code reordering[12][21] and instruction cache prefetching[11], have been studied and shown to increase performance in some cases. Hardware mechanisms for runtime data cache management[8] have been examined; these mechanisms, however, often require expensive hardware support. Although dynamic scheduling has been shown to increase the tolerance of load latency[19], even machines with out-of-order execution benefit from prefetching[3] and load speculation[18].

Standard compiler techniques[15] for data prefetching are well established and are used in many modern compilers. These algorithms, however, tend to rely on strided loops such as those in scientific applications[17]. Since such loops are rare in integer programs, data prefetching is not often used for general applications. Much of the work[16] relating to prefetching in integer programs has relied upon informing memory operations that expose the cache behavior of instructions to the program. However, such instructions are not yet available in the x86 microarchitectural family. In this work, we demonstrate that significant improvements can be achieved by using Pentium III specific cache management instructions and that this is applicable to a wide range of integer and floating point applications.

3 Trace Methodology

To effectively utilize the instruction set extensions, we rely on memory trace information to locate opportunities for optimizations. To gather this information, every memory reference in a binary is instrumented to generate a call to a runtime routine that generates output to a trace buffer with information about the memory reference. The instrumentation code sends the address of the memory instruction, the address of the accessed data and the corresponding size of the access to a runtime routine on each invocation. Control is occasionally passed to a cache simulator that consumes this trace on the fly. The simulator, which is modeled after the Pentium III Xeon™ L1 data cache, keeps track of the miss statistics for each memory reference. The specifics of the cache hierarchy are shown in Table 1. The latency values are determined experimentally and are used later to determine the optimal prefetch distance.

To enable prefetch and bypass optimizations the simulation keeps track of memory accesses and their reuse patterns. Four types of information are gathered during the simulation that is used later in the optimization phase: 1. a record of the miss frequency of all loads and stores is kept; 2. the strides between successive dynamic occurrences of each memory instruction are recorded; 3. the result of analysis performed to determine if the cache lines brought in by each memory instruction tend to be reused before being replaced from the cache is stored; 4. lines that are displaced by store instructions are recorded to determine if those lines are later brought back into the cache on a miss. This last mechanism detects situations where the miss could be avoided if the earlier store instruction had not write-allocated.

During the analysis phase, the executables and dynamically linked libraries of the benchmarks are instrumented. The C runtime libraries however are not instrumented, nor are the Microsoft®[*] Windows NT® system libraries. While the simulation thus sees only a partial list of all data memory accesses, the number of memory references and cache misses found during the simulation was a close approximation when measured against the Pentium III performance counters. This validation check provides some level of confidence in our simulation.

Both the instrumentation and optimization of the benchmarks is performed at the binary level using a post-link binary rewriting technology called Vulcan [20]. This technology is similar to ATOM[4], EEL[10], or the IMPACT Binary Reoptimization System[13]. Information from the simulation is fed to a post-link optimizer tool to produce an optimized binary using the heuristics detailed in Section 5. The optimized binaries are then rerun using the same input data as the training set to measure the effectiveness of the transformation. All of our measurements are made using a Pentium III Xeon 500 MHz machine running Microsoft Windows NT 4.0. Each program is run repeatedly on an unloaded machine and the actual execution time is measured. The additional

[*] Microsoft and Windows NT are registered trademarks of Microsoft Corporation in the United States and/or other countries.

Table 2 – Data cache control instructions

prefetcht0	Prefetch into 0th level cache (Into both L1 and L2 in Pentium III)
prefetcht1	Prefetch into 1st level cache (Into just L2 in Pentium III)
prefetcht2	Prefetch into 2nd level cache (Into just L2 in Pentium III)
prefetchnta	Prefetch non-temporal data (Into just L1 in Pentium III)

prefetch instructions

movntps	128 bit streaming store (From Streaming SIMD Extensions register set)
movntq	64 bit streaming store (From MMX register set)
maskmov	Masked variable length streaming store (From MMX register set)

streaming store instructions

performance metrics detailed in Section 6 are also measured during these runs using the Pentium III performance counters.

Although all the described optimizations are performed on existing binaries, the techniques described can be used at compile time to generate a machine-specific executable. In addition, since the source code is not used, these optimizations possibly could be done on the host machine. This might be facilitated by hardware profiling mechanisms[4][14].

4 Using the Pentium III Instructions

The Intel® Pentium III microarchitecture features several ISA extensions that can be used to manage placement of data in the memory hierarchy. The extensions include both data cache prefetching and cache-bypassing types of instructions. These instructions can dramatically improve the performance of the memory hierarchy, and thus can also substantially improve application performance.

4.1 ISA Extensions

The Pentium III's cache control instructions are shown in Table 2. Four prefetch instructions are provided to allow prefetching into different levels of the cache hierarchy. Since only two levels of cache are visible on the Pentium III, the effect of the `prefetcht1` and `prefetcht2` instructions is the same. Three streaming store instructions provide support for stores of various sizes that bypass the cache hierarchy. Each of these instructions performs a non-allocating store from either the 64 or 128 bit registers. These stores bypass the cache hierarchy if their cache line is not already present in the cache, otherwise they act as normal store instructions. The `maskmov` instruction performs a streaming store of selected bytes from an MMX register that are determined by a mask in an additional MMX register. For more information about these instructions see [6] and [7]. Further information on how these instructions are used in this work, and the difficulties and costs for employing them can be found in [1].

4.2 Data Prefetching

A significant body of previous research demonstrates that software controlled data prefetching can significantly improve cache effectiveness and system performance (see Section 2). In order to effectively use any type of data prefetching, however, it is necessary to determine how far in advance of a load to place a corresponding prefetch. To determine this optimal distance, experimental measurements are made. The results of using prefetch instructions in this test program executing on a Pentium III are presented in Figure 1. In at test program, the number of cycles between the prefetch and a load are varied from 1 to 100. The program consisted of long sequences of dependent instructions to minimize the effect of the out-of-order execution of the Pentium III. The test program also ensures that the load targets would reside in the L2 cache, but would not reside in the L1 cache. Each run of the test program is identical, except for the number of dependant instructions between the load and store. Figure 1 shows that the optimal number of cycles needed for prefetching an L1 cache miss in our test program is around 18 cycles. Not surprisingly, this corresponds approximately to the latency of a L1 cache miss as given in Section 3. It is important to note from Figure 1 that as long as the prefetch is issued a few cycles before the corresponding load, some performance gain is achieved, and this benefit increases relatively linearly up to the optimal distance. However, if a prefetch occurs too far in advance of the load, there is an increased chance that the desired cache line might be displaced before it is needed or the fetched line might displace some data that is needed first. When the load must go to main memory, maximum benefit will come from prefetching much farther in advance. Thus, maximum benefit comes from knowing which level of the memory hierarchy the data resides and scheduling the prefetch appropriately.

4.3 Important issues in using cache control instructions

The use of the machine-specific optimizations considered in this work is not without cost. In the case of data prefetching, the inserted prefetches can significantly increase the dynamic instruction count. If the memory hierarchy is swamped with prefetches it can cause trailing loads to stall. Also, extraneous prefetches can displace useful cache lines. When streaming stores are performed, an increase in memory transactions can occur if the cache line of the stored value is loaded soon after the store. In addition, since

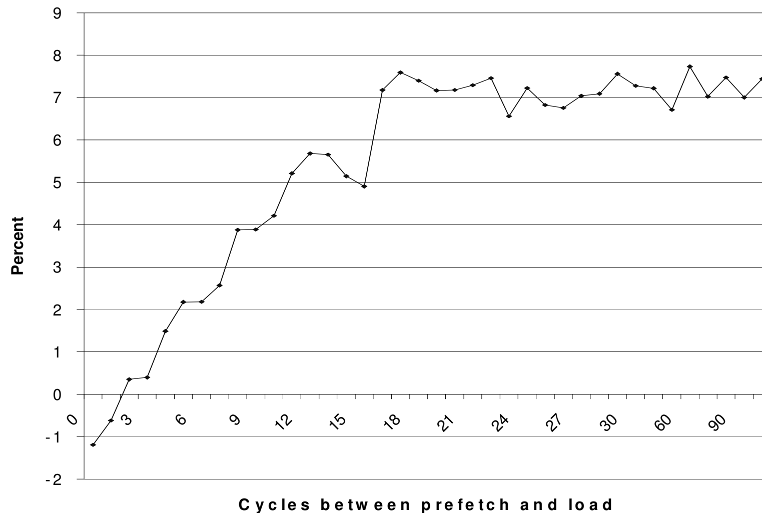

Figure 1. Speedup vs. prefetch distance in an experimental program.

there are no streaming stores from the 32-bit register set, general purpose stores cannot be made to be streaming without performing potentially costly moves of the data to the MMX™ register set. In addition, to do a 32-bit store requires setting up an additional mask register for the `maskmov` instruction.

5 Code Transformations and Heuristics

To effectively use the ISA extensions during optimization, several heuristics are used for determining where and when to apply transformations that use these instructions. The heuristics are tuned to minimize the total number of transformations made to the code by optimizing only those memory accesses that have a high probability of causing a cache miss. This might leave some opportunities unexplored but the goal of this work is to demonstrate the feasibility of such machine specific optimizations. This section details some of the code transformations and optimization heuristics that are used in this work. For further details see [1].

5.1 Important issues in using cache control instructions

There are several kinds of code transformations that are useful for making the optimizations considered in this work more effective and less expensive. This usually requires transforming the code sequence to make it more amenable to the optimization.

Often randomly strided memory accesses occur inside loops for integer applications. In these cases, it is often not possible to calculate the target address of loads in future iterations. One technique that might help in this situation is to speculatively prefetch across the back edge of a loop. In such cases, loop rotation[15] may allow more distance to be placed between the prefetch and its matching load. As shown in Figure 2, this technique allows the target address calculation for a load in the next iteration to be speculatively performed in the current iteration. In comparison to the original code sequence, this type of speculation causes more instructions to be executed, as it performs the address calculation for one more iteration than the original code. Therefore, it is important to ensure that the loop iteration count is relatively large so that the additional calculation becomes relatively insignificant.

Frequently, the address calculation for a candidate load instruction is dependent on another load instruction. In this case, the control flow often limits the distance that the two loads can be moved apart; thereby limiting the distance between the prefetch and the load. However, using conditional move instructions, the load can be moved above a branch without breaking the semantics of the program, as the execution of the load is still predicated on the branch condition. Now the corresponding prefetch can move above the branch, and the number of cycles between the issue of the prefetch and the load can be increased, improving the effectiveness of the optimization. This type of speculative load transformation is shown in Figure 3.

The x86 architecture features string instructions that can load and store to memory using a single instruction. Often these instructions are designated with a `rep` prefix so that they are repeated without fetching and decoding additional instructions. In this way, a single instruction can operate on a large amount of memory. These instructions are found in many general programs and are amenable to special optimizations. Since these instructions access a large portion of contiguous memory, the 64-bit registers can be used to operate on the majority of this data, reducing the dynamic number of memory operations. The single instruction is replaced with an optimized loop that can utilize both prefetches and streaming stores from the MMX register set. The string instructions use a loop counter to specify how much

data is to be accessed. This counter can be properly adjusted and the majority of the data can be accessed through 64-bit MMX registers, facilitating the use of streaming stores of this data. For string instructions with poor cache behavior, performance is greatly improved by using prefetches to bring data needed for future iterations of the loop into the cache and by using streaming stores to store data with poor temporal locality. Although these optimizations increase the number of instructions that execute, they improve execution performance.

5.2 Prefetch Heuristics

Much of the prior work related to data prefetching has looked at optimizing array accesses in inner loops of numerical applications. Unfortunately, these accesses occur infrequently in general integer applications. To determine if a load behaves in a strided fashion, our simulation uses a form of value profiling[2][9] in the simulation on the target addresses of the load instructions. For every static load instruction in an executable the simulation maintains a last address-accessed datum. The difference between the current memory location and the last memory location is a stride that gets recorded into a stride list for a particular load instruction. For a load that accounts for more than five percent of all the misses, we use the most commonly encountered strides to determine the prefetch optimization.

When a candidate has a single dominant stride (>70%) we found that placing it in the cache as early as possible gives the best results. Intel defines the number of iterations to prefetch ahead as the *prefetch distance* and provides a formula for its calculation[7]. A similar, but slightly simpler approach is taken in this work. The length of a loop iteration in dynamic instructions is used to determine the distance to prefetch ahead. Since the loop may contain some control flow, profiling information can be used to determine the frequency of each control flow edge, and the average length of a loop iteration is used. The prefetch is inserted to attempt to prefetch 25 dynamic instructions before the load. Thus the number of iterations to prefetch ahead is equal to 25 divided by the average length of a loop in instructions. The distance of 25 dynamic instructions was chosen experimentally, and corresponds to an IPC of approximately 1.4 if the distance in cycles is desired to be 18.

Often in integer programs, memory accesses fail to follow any strided pattern. Only the current iteration of a loop is prefetched in these cases. It is generally difficult to prefetch effectively within one loop iteration. However, several of the optimizations described earlier in this section can be used to increase the distance between the availability of the load address and the execution of the load instruction. A candidate falling into this category is not prefetched if the prefetch cannot be placed at least three instructions before the load, since as shown in Figure 1, inserting prefetches immediately before corresponding loads hurts performance.

5.3 Streaming Store Heuristics

Using the streaming store instructions in general programs is slightly more complicated than using the prefetching instructions. There is the additional cost described in of moving data to the MMX register set before performing the store that must be accounted for in the cost analysis.

Candidates for streaming store optimization are chosen via the same cache simulation used for the prefetch analysis. We found that the best candidates for optimization are those stores that miss in the cache more than 90% of the time. Due to the cost of this optimization, it is detrimental to use a streaming store on a location that is already resident in the cache. Cache hits due to cache lines brought in by previous executions of the same instruction are not considered hits in this determination. This helps to eliminate the cache hits that would not occur if it a non-allocating write had been executed. Of these candidates, only the store instructions with data that are written back to L2 before being read in more than 90% of its dynamic occurrences are considered. Analysis is done to determine if bringing a cache line in for each store miss pollutes the cache and causes additional misses. The number of additional misses caused by an instruction is multiplied by the latency of a L1 cache miss. This value is compared to the number of times the instruction executed times 6 to account for the overhead of inserting the code transformation. If the effect of the additional cache misses is larger than the assumed transformation cost, the benefit of using the streaming store instruction likely outweighs the large cost in the general case.

The cost of using the streaming store optimization in general programs makes it unlikely to be of much benefit. However, several special cases do occur in general programs that make effective use of the streaming store optimization. For example, if two instructions store to consecutive memory locations and these operations are both cache polluting then the two stores can be combined into a single larger store using the 8-byte registers. This has the advantage of possibly reducing bus traffic and reducing the number of instructions executed.

Another example of stores that can often be effectively transformed is the string instructions. As described earlier in this section, using the streaming stores for this case is inexpensive since the code transformation has already placed the data in an MMX register. If the string instruction is likely to cause a large number of cache misses and poor temporal locality is identified, then expanding the instruction into a small loop using both prefetching and streaming stores tends to improve the performance.

```
LOOP:  MOV  EDX, DWORD PTR [ECX]
       ADD  EDX, ESI
       XOR  ECX, EDX
       ADD  EDI, DWORD PTR [ECX]
                 .
                 .
                 .
       CMP  DWORD PTR [ECX], EBX
       JNE  LOOP
```

```
       MOV          EDX, DWORD PTR [ECX]
       ADD          EDX, ESI
       XOR          EAX, EDX
LOOP:  MOV          ECX, EAX
       MOV          EDX, DWORD PTR [ECX]
       ADD          EDI, EDX
       ADD          EDX, ESI
       XOR          EAX, EDX
       PREFETCHT0   BYTE PTR [EAX]
                      .
                      .
                      .
       CMP          DWORD PTR[ECX], EBX
       JNE          LOOP
```

Figure 2. Using loop rotation to facilitate prefetching across a back edge

```
              .
              .
              .
CMP   EAX, 0
JE    NULL
              .
              .
              .
MOV   EBX, DWORK PTR [EAX]
              .
              .
              .
MOV   EDX, DWORD PTR [EBX]
```

```
MOV          EBX, 0
CMP          EAX, 0
CMOVNE       EBX, DWORD PTR [EAX]
PREFETCHT0   BYTE PTR [EBX]
               .
               .
               .
JE           NULL
               .
               .
               .
MOV          EDX, DWORD PTR [EBX]
```

Figure 3. Using speculative load to increase distance between prefetch and load

6 Experimental Evaluation

By improving the cache behavior of application programs, significant overall performance increases can be achieved. To demonstrate that machine specific optimizations on the Pentium III can yield this improvement, seven test cases are run on six different applications. Each is probed and simulated to determine locations for optimization, and then optimized using a binary rewriting tool.

6.1 Benchmarks

The benchmarks in the study are chosen to represent a wide range of applications. *Compress* and *tomcatv* are taken from the SPEC95 integer and floating-point benchmark suites respectively. For these benchmarks, the SPEC reference inputs are used for performance measurements. *Ghostscript* is taken from the Aladdin Ghostscript 5.5 public release from the University of Wisconsin. *Ghostscript* is a postscript interpreter, and as the test case, a postscript file containing a large document is rastered and displayed. *Microsoft FoxPro* is a database application, and a scenario performing numerous transactions on a database is used as its test case. *Microsoft Word* and *Microsoft Excel* are large desktop applications, taken from the Microsoft Office 2000 suite. As a test case for *word*, a large word document is run through the find-and-replace operation. For *excel*, two different test cases are used. The first test case stresses the recalculation engine of *excel* by making numerous changes to a large spreadsheet. The second test case stresses the column editing operation over the same spreadsheet.

6.2 Performance

As shown in Figure 4, significant gain can be achieved on a wide range of applications. An average speedup of 6.8% with speedups as high as 27% is measured using the prefetch and bypass optimizations. Ignoring the high and low outlier results, an average of 4.3% speedup is obtained.

While the speedups for each benchmark vary considerably, only *compress* shows insignificant performance improvement. The performance statistics in Figure 4 is broken into three components: the combined effect, the prefetching optimization alone and the effect of the streaming store optimization alone*. Where the streaming store optimization is applied alone, a small decrease in performance is observed in all cases except *ghostscript*. There are a number of reasons for this decrease in performance. For *compress*, the insertion of several consecutive streaming store instructions has a serializing effect on instruction issue, since each of these instructions require the single complex instruction decoder available on the Pentium III. For the other applications, the weight of the enabling instructions for the streaming store optimization proves costly. The prefetching optimization alone has a positive

* *foxpro* and *tomcatv*, however, did not have streaming store optimization candidates.

effect on all applications. For *word* we have the situation where the combined optimizations yield a superior result to prefetching alone, even though the streaming store optimization alone has a negative effect. In the combined optimization, the use of prefetches, by placing greater pressure on the L1 cache, has made the use of streaming stores more effective.

Word is a clear outlier with the most improved performance. The majority of this improvement comes from the string optimization described in Section 5. Since a large number of changes are made to the *word* document in the test case, a routine for moving memory is called frequently. With the improved performance of this routine, the performance of w*ord* improves dramatically.

Another way of evaluating the effectiveness of the optimizations is the utilization of the data cache as seen from Figure 5. This chart shows the weighted number of cycles that the processor might have to wait on the data cache to service an outstanding miss due to a load. This is calculated in the following way: during each cycle a count of the number of outstanding load misses is added to a running sum which is then normalized by the total number of cycles to execute the original program. For *tomcatv*, *compress*, *foxpro* and the e*xcel* insertion scenario, this number is originally very close to one. Thus, on average there is one outstanding cache miss throughout the run of the test. This is further evidence that the out-of-order execution of the Pentium III is quite successful in tolerating load misses, since for this average to be close to one, often the processor must have several outstanding loads at a time. However, when optimized, there is a dramatic difference in this average. This difference is caused by the data prefetches that are started earlier than the load. These measurements are made using the performance monitoring counters of the Pentium III. Unfortunately this metric does not count any cycles for a load whose address is prefetched, even if the prefetching is not done in time. Thus the decrease in relative weighted number of cycles is the load latency that prefetching attempts to tolerate, not the latency that is actually tolerated.

6.3 Memory Bandwidth

An undesirable side effect of data prefetching is the increase in bus traffic as shown in Figure 6. This can happen if the prefetch displaces data from the cache that is used before the prefetched data, or if speculative prefetches are made for memory locations that are not in fact loaded. One notable result is the relatively dramatic increase in the memory bus transactions in the *ghostscript* benchmark. As the number of bus transactions almost double, this likely hampers some of the gain from using the cache optimization. It should also be noted however that while there is a large percentage increase in this bus traffic, the total number of memory bus transactions in *ghostscript* is actually rather small.

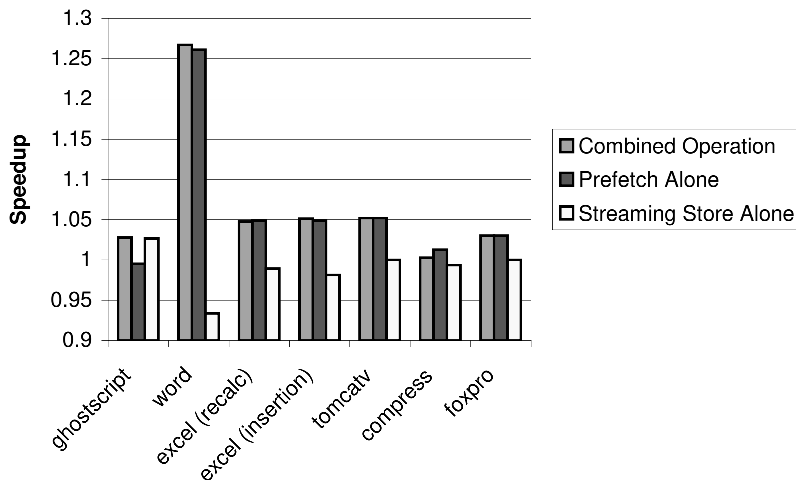

Figure 4. Breakdown of performance improvement

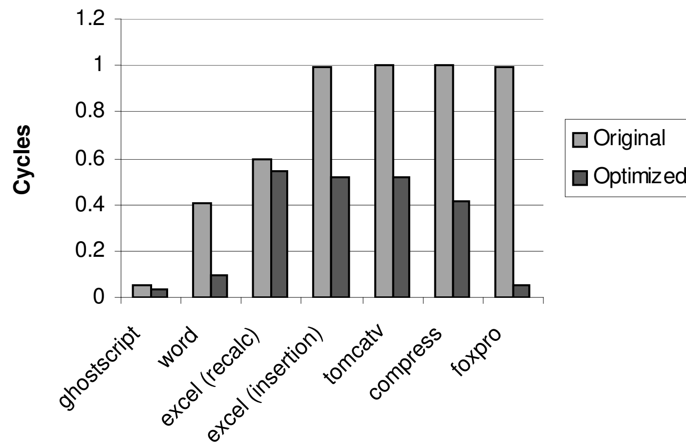

Figure 5. Weighted numbers of cycles with an outstanding cache miss

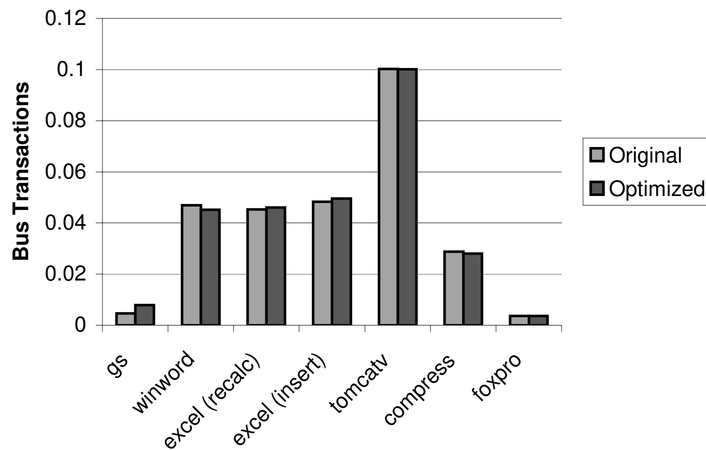

Figure 6. Bus transactions normalized by the original number of memory references

Since a streaming store is not write-allocating, it prevents the need for an initial read to bring the line into the cache hierarchy. This can serve to decrease the number of memory bus requests, canceling out some of the effect of increased pressure caused by data prefetching. In addition, the reduction in cache pollution and corresponding reduction in cache misses can reduce the traffic to memory as well. This positive effect is shown most prominently in Word. In addition, optimizations such as the string optimizations detailed in Section 4, use a 64-bit register where 32-bit registers were used previously. Since the memory bus of the Pentium III is 64 bits wide, this can also serve to reduce the pressure on the memory bus.

7 Conclusions

This paper provides some experimental support for specialized compilation within the x86 processor family. To do so, this paper studies the effectiveness of the Pentium III prefetch and streaming store data cache-controlling instructions. These instructions are highly effective in optimizing the cache behavior under ideal circumstances. To show that the instructions can be useful to general programs, several applications are studied including non-loop intensive, integer desktop applications. Memory traces are collected with the aid of program articulation at the binary level and are concurrently processed by a cache simulator. With information from the cache simulation, heuristic optimization techniques are applied to identify and optimize, again at the binary level, a small number of instructions with poor cache behavior. The performance results show that the Pentium III cache-controlling instructions can be effectively utilized to achieve performance improvements in the range of 3-27%, with an average of 6.8%. With this and other specialized compilation techniques, the performance of applications within the x86

family can be greatly improved over generically compiled code.

This paper also demonstrates that, for general programs, it is relatively easy to utilize the Pentium III prefetch instructions but that there can be significant overhead associated with the use of the streaming store instructions. To enable streaming store optimization on a wide range of programs it is highly recommended that the hardware implement a version of the streaming store instruction using a general-purpose register.

Acknowledgements

The authors would like to thank all the members of the Programmer Productivity Research Center at Microsoft Research and the IMPACT compiler team at the University of Illinois for their support, comments, and suggestions. We would also like to thank the anonymous referees for their constructive comments.

References

[1] Barnes, Ronald, Ronnie Chaiken, and David Gillies. “Feedback-Directed Data Cache Optimizations for the x86”, Technical Report, Microsoft Research, Sept. 1999.

[2] Calder, Brad and Peter Feller, and Alan Eustace. “Value Profiling and Optimization”, *Journal of Instruction Level Parallelism*, March 1999.

[3] Chen, William Y., Scott A. Mahlke, Pohua P. Chang, and Wen-mei W. Hwu. “Data Access Microarchitectures For Superscalar Processors with Compiler-Assisted Data Prefetching”, *Proceedings of the 24th International Symposium on Microarchitecture*, Nov. 1991.

[4] Dean, J., J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, “Profileme: Hardware support for instruction level profile-driven compilation using the profile buffer,” in Proceedings of the 29th International Symposium on Microarchitechure, Dec. 1996.

[5] Eustace, Alan and Amitabh Srivastava. “ATOM: A Flexible Interface for Building High Performance Program Analysis Tools”, USENIX Winter Conference, 1995. Also available as WRL Technical Report TN-44.

[6] *Intel Architecture Software Developer’s Manual, Volume 2: Instruction Set Reference*. Intel Corporation, 1999.

[7] *Intel Architecture Software Optimization Reference Manual*. Intel Corporation, 1999.

[8] Johnson, Teresa L., Matthew C. Merten, and Wen-mei W. Hwu. “Run-time Spatial Locality Detection and Optimization”, *Proceedings of the 30th International Symposium on Microarchitechure*, Dec. 1997.

[9] Kalamatianos, John, Ronnie Chaiken, and David Kaeli. “Parameter Value Locality of Windows NT-based applications,” Workshop on PC-Performance and Analysis held in conjunction with Micro-31, November 1998.

[10] Larus, James R. and Eric Schnarr. “EEL: Machine-Independent Executable Editing”, *Proceedings of the Conference on Programming Language Design and Implementation*, June 1995.

[11] Luk, Chi-Keung and Todd C. Mowry. “Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors”, *Proceedings of the 31st International Symposium on Microarchitechure*, Nov. 1998.

[12] McFarling, Scott. “Program Optimization for Instruction Caches,” *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[13] Merten, Mathew C. “A Framework for Profile-Driven Optimization in the IMPACT Binary Reoptimization System”, Masters Thesis, University of Illinois, 1999.

[14] Merten, Matthen C., Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen-mei W. Hwu. “A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization”, *Proceedings of the 26th International Symposium on Computer Architecture*, May, 1999.

[15] Muchnick, Steven S. *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, Inc., San Francisco, CA 1997.

[16] Mowry, Todd C. and Chi-Keung Luk. “Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling”, *Proceedings of the 30th International Symposium on Microarchitechure*, Dec. 1997.

[17] Mowry, Todd C., Monica S. Lam, and Anoop Gupta. “Design and Evaluation of a Compiler Algorithm for Prefetching”, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[18] Reinman, Glenn and Brad Calder. “Predictive Techniques for Aggressive Load Speculation”, *Proceedings of the 31st International Symposium on Microarchitechure*, Nov. 1998.

[19] Srinivasan, Srikanth T., and Alvin R. Lebeck. “Load Latency Tolerance In Dynamically Scheduled Processors”, *Proceedings of the 31st International Symposium on Microarchitechure*, Nov. 1998.

[20] Srivastava, Amitabh, et. al. “Vulcan”, Technical Report, Microsoft Research TR-99-76, Sept. 1999.

[21] Torrellas, J. and C. Xia, and R. Daigle. "Optimizing Instruction Cache Performance for Operating System Intensive Workloads", *IEEE Transactions on Computers*, IEEE, Inc., Vol. 47, Number 12, December 1998.