Kernel Optimizations and Prefetch with the Spike Executable Optimizer

**Richard Flower, Chi-Keung Luk, Robert Muth, Harish Patil,
John Shakshober, Robert Cohn, and P. Geoffrey Lowney**

Compaq Computer Corporation*

ABSTRACT

Spike is an executable optimizer that uses profile information to place application code for improved fetch efficiency and reduced cache footprint. This placement reduces the number of cache misses and the latencies they add to program execution time. This paper presents extensions of Spike to take advantage of three additional performance opportunities: 1) optimization of the Unix kernel code, 2) prefetching to reduce latencies of long latency loads, and 3) prefetching for loads with predictable strides that are not detected at compile-time.

INTRODUCTION

Knowledge gained from profiles of program execution can be used to improve performance. A programmer can use profiles to improve algorithms; a compiler to generate better code; a linker to produce better procedure placement; or an executable optimizer to rewrite the executable for better fetch and cache miss behavior. Spike is an executable optimizer originally intended for NT applications and later modified for Tru64 Unix applications [Cohn97a, Cohn97b]. Initially Spike improved TPC-C performance by approximately 30% and other applications by lesser amounts. However a number of performance opportunities remained. This paper presents Spike modifications to exploit three of these opportunities. The first opportunity is the time spent in the operating system code. TPC-C Oracle spends roughly 30% of its time in the Unix kernel while SPECweb spends 85% of its time in the kernel. The second opportunity is long latency loads. On a NUMA machine like the Compaq GS320, accesses to non-local memory can take three times longer than local memory references. If a prefetch can be placed earlier in the code path, the stall on the load is reduced or eliminated, especially when the prefetch can be overlapped with another memory access. The third opportunity is the latency of some loads that are executed repeatedly. If the addresses accessed by the load exhibit a regular strided pattern, then the address for the next execution of the load is predictable and a prefetch can reduce its latency. Many of these strided access patterns are recognized and prefetched by the compiler. However some strided access patterns cannot be recognized by the compiler, and Spike can often profitably prefetch these.

The remainder of this paper is organized as follows. The next two sections discuss the profile collection and Spike's code layout algorithms for icache packing. Section 4 discusses optimization of the Unix kernel and section 5 gives some results. Latency based prefetching is presented in section 6 and stride-based prefetching in section 7. The remaining two sections discuss related work and conclusions.

2. PROGRAM PROFILE COLLECTION

The primary profile information used by Spike is a set of execution counts for the basic blocks of a program. The counts can be exact counts collected by instrumenting the program [Smi91] or estimated counts collected with the DCPI statistical profiler [And97]. Instrumentation has the advantage of producing exact counts but the instrumentation increases execution time significantly. Also not every basic block can be instrumented in large complex programs like the Unix kernel where there is some use of self-modifying code. Statistical profiles have the advantages of only minor increases in execution time and coverage of the entire image. However the counts produced are estimates and the program may need to be run longer to get statistically significant counts. For the Unix kernel and the large commercial applications reported in this paper, we have not seen significant performance differences between the optimized code produced from estimates and from exact counts. For programs with a short

* Luk, Muth, Patil, and Lowney are currently with Intel Corporation, Massachusetts Microprocessor Design Center, Shrewsbury, MA.

run-time, such as the training workloads in SPECcpu2000, exact counts sometimes give better results.

3. ICACHE PACKING

The goal of Spike's code layout is improved instruction cache performance. Spike uses the algorithm introduced by Pettis and Hansen [Pet90]. The basic blocks of a procedure are arranged so that the frequent execution path is straight-line. The rarely executed code is split into a separate *cold* procedure. Finally procedures that call each other are placed close together so that they will not inadvertently conflict in the cache.

To arrange a straight-line execution path, Spike first constructs a flow graph for the basic blocks of a procedure. Edge weights are estimated from the basic block execution counts. The blocks are joined into sequential traces based on edge weight and the infrequently executed blocks are split into a separate *cold* procedure. Minor padding is added to improve branch prediction and fetch efficiency.

To place procedures Spike constructs a procedure call graph. The call counts are based on the execution counts for the basic blocks containing the calls. Procedures connected by a frequently exeeeuted call are placed adjacent so that they will not conflict in the cache.

4. SPIKE APPLIED TO UNIX KERNEL

Getting Spike and the Tru64 Unix kernel to work together required minor changes in each. Most changes were due to 1) the kernel's unique execution environment especially during boot, 2) the kernel's use of self-modifying code to gain additional performance, or 3) kernel assumptions about the order of (or distance between) basic blocks.

For example the kernel normally executes with a fixed kernel address established in a *global pointer* or *gp* register. This *gp* register is used as a base register for many accesses to the kernel area. Spike makes use of this register in adjusting some procedure calls. However an adjusted procedure call would go astray if executed during boot prior to *gp* establishment. Thus Spike needs kernel specific knowledge of which code executes prior to *gp* establishment.

The kernel makes some use of self-modifying code in a few key performance critical areas. Most such modifications either patch out a procedure call or insert a branch. Prior to Spike the branch displacements could be known at link time. However Spike changes the branch displacements when it changes procedure layout to avoid cache conflicts, and the kernelís self-modifying code must be able to accommodate the code rearrangement. In particular, the kernelís self-modifying code must be prepared to deal with displacements that exceed the 21 bit limit in the Alpha branch instructions.

One example of kernel assumptions about basic block and procedure order involves clearing and reclaiming a memory region. There is a set of kernel procedures that are used only during system boot. Immediately following system boot the memory they occupy is cleared and reclaimed. The reclamation relies upon the fact that this special set of procedures is linked at low addresses and that a symbol on the last basic block of the last procedure indicates the end of the region to be cleared. Spike requires kernel specific knowledge of this boundary symbol to avoid placing normal kernel procedures in the cleared and reclaimed region.

Spike eliminates unreachable code. However for dynamically loaded drivers, the kernel makes use of code that is not reachable by normal means but is reachable through use of the symbol table. Spike retains this code.

Prototype application of Spike to the kernel presented some interesting challenges. Some problems were difficult to find because the kernel crashed too early for establishment of any crash dump or debugging environment. The most difficult to diagnose problems were the result of bugs in the front end tools feeding Spike. For example relocation information in the original kernel executable [Obj01] is used by Spike to update addresses. Errors in the relocation information resulted in optimized kernels that failed to boot. Independent checks are now run on this front end information prior to consumption by Spike.

Minor changes were required in Spike to accommodate the kernel debugging tools. To construct a traceback from a current program counter location, the kernel debugger scans back through the contiguous procedure code to the beginning, expecting to find instructions that allocated the stack frame and saved the return address. When Spike forms a separate ìcoldî procedure from the rarely executed code, it prepends a copy of the instructions that allocate the stack frame and save the return address. These instructions are strictly for use by the debugging tools when constructing a traceback; they are never executed. Since they are associated with a cold procedure, they have negligible effect on instruction cache utilization.

5. RESULTS FOR ICACHE PACKING

Spike has been used to optimize a number of important applications including transaction processing (TPC-C), decision support (TPC-H) and

Benchmark	Description	Machine
TPC-C	Online transaction processing benchmark running inventory control against an Oracle 8.1.7 warehousing database measured in transactions per minute	GS320 GS160
SAP-BW	Business data warehouse application running Oracle 8.1.7 and SAP client application measured in query response time.	GS160
TPC-H	Oracle 8.1.7 data warehouse benchmark running 22 ad hoc queries and 2 update functions measured in queries per hour.	GS320
Oracle App	Oracle 11i application server running clients against an Oracle 8i database measured in number of users.	GS320
SPECweb99	Web server benchmark with dynamic content measured in simultaneous users.	ES45
SPECweb96	Older web server benchmark with only static content measured in operations per second.	ES40

Table 1 Benchmarks

web server (SPECweb). Table 1 lists the benchmarks along with a short description and the machine model used to run them. Table 2 gives the processor and memory characteristics of the benchmark machines. All machines had 21264A (EV67) or 21264C (EV68) processors. These processors have a 64 KB on chip Icache, a 64 KB on chip Dcache, and a unified board-level L2 cache.

Model	Processors Nr.	Type	MHz	L2 Cache	Memory
GS320	32	21264A	731	4 MB	128 GB
GS160	16	21264A	731	4 MB	64 GB
ES45	4	21264C	1000	8 MB	32 GB
ES40	4	21264A	667	8 MB	2 GB

Table 2. Machines

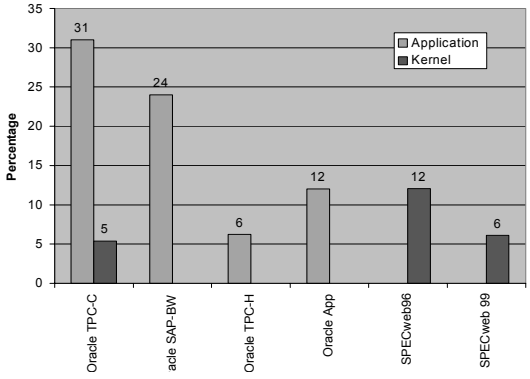

Figure 1. Spike Performance Gains

Some applications such as TPC-C were retuned after being optimized with Spike, to take advantage of the cycles freed up by the optimizations. All but TPC-H and SPECweb96 were submitted to the relevant benchmark organization [Tpc01, Spec01]. The benchmarking organizations seek to have benchmark run rules that result in little run to run variation in performance. In practice we saw variations of up to two percent.

The performance gains on these applications for the icache packing optimization range from 5 to 35 percent as shown in Figure 1. These performance gains are typical of the gains seen with Spike for large call-intensive programs.

The goal of the icache packing is to improve fetch efficiency and reduce the cache footprint. These improvements are reflected in lower numbers for Icache (L1) misses and unified board-level cache (L2) misses. The relative improvements in Icache and L2 misses are shown in Figure 2 for SPECweb96 and TPC-C. For SPECweb96 the original pre-Spiked kernel had an Icache miss rate of 16.3 misses per thousand instructions while the Spiked kernel gave a reduced miss rate of 8.4 Icache misses per thousand instructions. The relative improvement in Icache miss rate for SPECweb96 was nearly 50 percent. Similarly the L2 miss rate was reduced from 1.74 to 1.15 misses per thousand instructions for a relative improvement of 34 percent for SPECweb96.

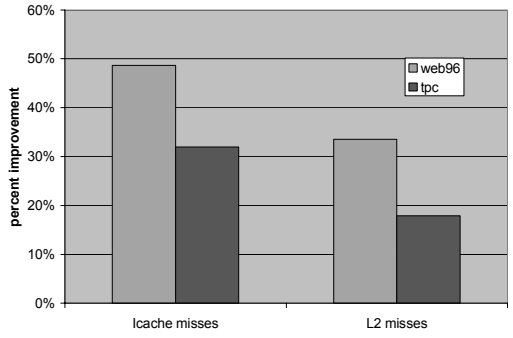

Figure 2. Improvement in Miss Rate

The cache miss data were collected from the 21264A hardware performance counters using DCPI with ProfileMe [Dea97]. In some cases the events recorded using ProfileMe are not exactly the events that we would like to monitor. In particular the event

used to indicate icache misses is more precisely a *not-yet-prefetched* event that requires some explanation. When the 21264A misses in the icache, it initiates a fetch of the necessary cache line and also prefetches of the following three cache lines. The *not-yet-prefetched* event occurs when a cache fill must be initiated and the fetch unit stalls for the full cache fill latency. The event does not include those cases where a prefetch is in progress and only a partial cache fill latency is seen. Thus the *not-yet-prefetched* event undercounts misses in the sense that it does not include the case where a prefetch is already in progress.

The miss rate improvements from Spike can produce a measurable improvement in benchmark performance if overall miss latency accounts for a measurable portion of the original benchmark execution time. Conversely a benchmark that spends little of its time in Icache misses and L2 misses for code access will show little improvement with Spike. SPECsfs (formerly LADDIS) is an NFS fileserver benchmark that seems to fall into this category.

The portion of overall execution time attributable to Icache and L2 misses can be calculated using estimates of the miss latencies along with DCPI event counts for Icache misses (*not-yet-prefetched*) and L2 misses. The miss latencies on the Alpha 21264A are approximately 100+ cycles for L2 misses to memory, 50+ cycles for ITB misses, and 20 cycles for an Icache miss that hits in the L2.

Figure 3 shows the estimated percentage of execution time attributable to misses for SPECweb96. For the original base kernel approximately 20 percent of the execution time was due to Icache misses and another 10 percent due to L2 misses. These numbers were cut to 13% and 8% for the Spiked kernel.

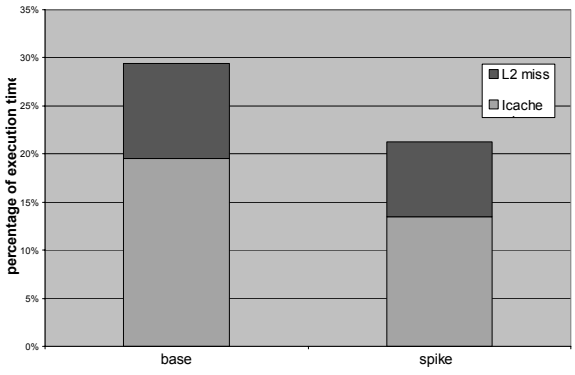

Figure 3. Normalized Execution Time Due to Misses in SPECweb 96 Benchmark

These estimates assume that little of the miss latency is overlapped with execution. This is almost certainly the case for the Icache misses and those L2 misses which load instructions. It is less certain for those L2 misses which load data. This is because the stall occurs on the use of the load rather than occurring on the load itself. Execution time between the load and use covers some of the memory latency. The prefetching optimizations discussed in the following sections seek to cover more of the latency by prefetching the data into the L1 Dcache ahead of the load.

Icache packing typically also reduces the memory footprint of a program and reduces the number of TB entries needed to map that footprint. Figure 4 gives an indication of the success of the icache packing algorithm in reducing memory usage.

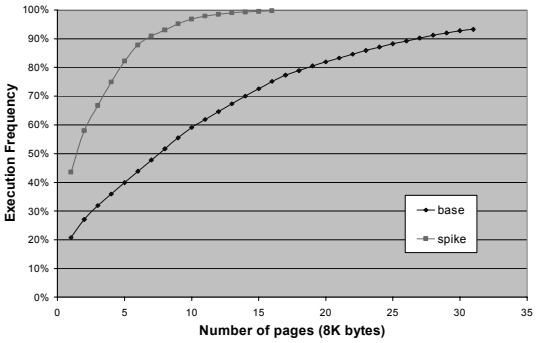

Figure 4. Effects of Icache Packing

The plot shows cumulative execution frequency for the hottest 8K-byte pages in the original pre-spiked kernel and in the spiked kernel executing the SPECweb96 benchmark. In the spiked kernel the most frequently executed page accounted for 40 percent of the execution frequency while in the original base case the most frequently executed page accounted for only 20 percent and five pages were needed to account for 40 percent of the execution frequency. Spike managed to place instructions accounting for 93% of the execution frequency in 8 8K byte pages.

6. LATENCY-BASED DATA PREFETCHING

For TPC-C a significant amount of time is spent on memory stalls for data accesses [Bar98]. Spike is able to reduce memory stalls by selectively inserting prefetches ahead of those loads that have been measured to have high memory latency. Once load latencies have been measured Spike evaluates each load to determine whether it is a candidate likely to benefit from prefetching. For likely candidates Spike

identifies possible prefetch sites, estimates the benefits of each site based on prefetch distance and execution frequency, and inserts a prefetch at the best site if benefits outweigh costs. Details of this general algorithm along with performance results are given in the following subsections.

Selecting candidate loads: Two techniques based on DCPI are used to estimate load latency. The technique of value profiling [Bur00] periodically interprets a sequence of instructions and uses the cycle counter to measure the latency of any loads. A second technique uses ProfileMe data to estimate instruction latency as the accumulated retire delay for an instruction divided by the number of retires. Since the delay occurs on the use of a load rather than the load itself, Spike looks for loads whose consumers have long latencies. The value profiles and/or the ProfileMe data needed for estimating load latency are held in an enhanced basic block database [Alb99].

Prefetches are more likely to produce performance improvements if the corresponding loads are frequently executed and if the load misses and must go to memory. Thus we consider only loads that are part of the 75% most frequently executed instructions and which have average load latencies greater than 50 cycles. The remaining loads are eliminated from further consideration. Further consideration would increase Spikeís run time without improving the optimized executableís performance.

Insert Prefetch: For each load we only consider prefetch insertion points that dominate the load and where the address can be easily calculated. An important concept is the notion of the "most distant (intraprocedural) prefetch point", *mdpp*. Intuitively, this is the earliest dominating point where the load address can be computed by adding a constant to the contents of some register. To determine *mdpp* Spike walks up the use/def chain tracking the load address across register moves, addition of a constant, and register spill/restores. Possible insertion points for prefetches for a load are all dominating points on the path from the loadís *mdpp* to the load. Figure 5 shows an example where we are trying to prefetch for the load**,** *e=*d*, at the bottom of the control flow graph. The load's *mdpp* can be found at the merge point of the two instructions writing register *c*. All possible prefetch insertion points are marked with black bullets.

We are currently investigating another mechanism that will lead to even earlier prefetch points. We can exclude little or never executed parts of the control flow graph from the dataflow analysis for the register defining the load address. Unlike regular dataflow analysis such as liveness, this does not cause correctness problems, because prefetching does not require accurate dataflow analysis. In our example above only the shaded nodes are executed. Excluding the two non-shaded blocks and the corresponding edges will result in a smaller control flow graph. The new *mdpp* is marked as *mdpp'*. The additional prefetch insertion points are marked with white bullets.

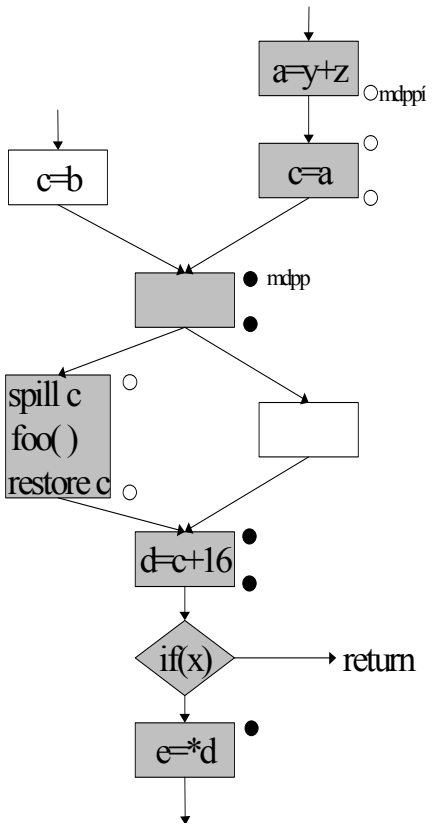

Figure 5. Possible Prefetch Insertion Points

Cost Benefit Analysis: The cost benefit analysis takes the list of possible prefetch points computed in the previous step and computes a benefit metric for each of them utilizing additional profile information. In our simplified model two properties affect the benefit of a prefetch: 1) its distance ahead of the load and 2) the likelihood that the prefetch is being utilized.

The earlier we can prefetch the more likely it is that we can hide the memory latency and the higher the benefit. Determining the exact distance between the prefetch and the load is difficult and we approximate it using the number of instructions on the shortest path between them. We currently require that the prefetch be at least 8 instructions earlier than the load.

Not all prefetches are useful. We require that a prefetch dominate its associated load, but this does not guarantee that the load will always be executed. For example in Figure 5 we could prefetch at *mdpp*, evaluate *x* as true, and return from the procedure without ever reaching the load at *e=*d*. A prefetch that was executed much more often than the load would frequently incur costs yet rarely return benefits. Using path profiles it would be possible to accurately determine how often a prefetch is followed by execution of the load. Since we do not have path profiles we approximate this by the ratio of prefetch execution frequency to load execution frequency. We currently require that the prefetch be executed no more than twice as often as the load. We favor points where the execution frequencies are equal.

For Oracle TPC-C the above algorithms result in the insertion of approximately 70 prefetches into the Oracle image by Spike. (While tuning, the number of inserted prefetches varied from 40 to 100.) The prefetches result in performance improvements of approximately 10 percent. For the particular TPC-C machine configuration used for earlier results, Spike without prefetching gave a 31 percent improvement over the base while Spike with prefetching gave a 39 percent performance improvement.

7. PROFILE-GUIDED STRIDE PREFETCHING

If a load instruction accesses memory in a regular strided pattern, then the next address to be accessed is easily predictable. Spike is able to insert prefetches and reduce memory stalls for these strided memory accesses [Luk01]. Using Spike to insert profile-guided prefetchs takes three steps: 1) instrumentation of the application, 2) collection of a stride profile, and 3) insertion of prefetch instructions.

Instrumentation: Load instructions are instrumented with an enhanced control-flow profiling tool. We use heuristics to avoid instrumenting all loads. There is no need to instrument loads that have been prefetched by the compiler. These are typically loads in a loop with a constant stride known at compile-time. Also loads of scalars and small structures are not candidates for stride-based prefetching, and they do not need to be instrumented. These loads are recognized by their use of the stack pointer or the global pointer as the base register. Conceptually each of the remaining loads is instrumented. In practice where several loads use different fixed displacements from the same base register, only the single base register value needs to be profiled. Instrumenting all loads will slow an application down by a factor of 10. By using heuristics to reduce the amount of instrumentation we can reduce profiling time by a factor of three without compromising our stride-based optimization.

Stride Profile: The instrumented application is run to collect a stride profile. For each instrumentation point, the stride profile gives stride values, frequencies for the stride value, and run lengths for the stride value. The run time of the instrumented application can be reduced through sampling without giving up accuracy.

Prefetch Insertion: Spike uses the stride profile to guide insertion of prefetches. For an inserted prefetch one needs to determine both a stride value and the prefetch distance (i.e. how many iterations ahead to prefetch). For each inserted prefetch Spike uses a fixed stride value. An alternative would be to dynamically calculate a stride value at run time. However, profiles have shown that one stride value tends to dominate. The added instruction and register costs of dynamically

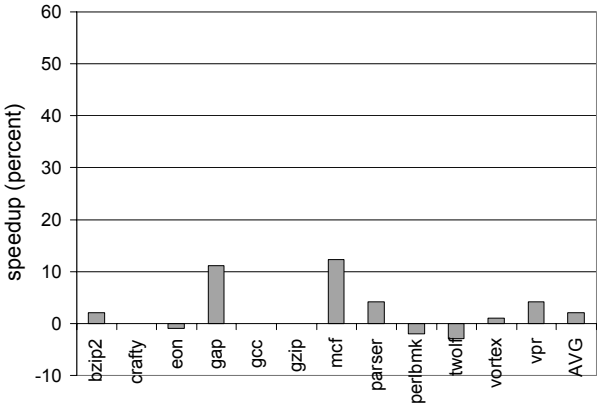

Figure 6a Integer Benchmark Improvements

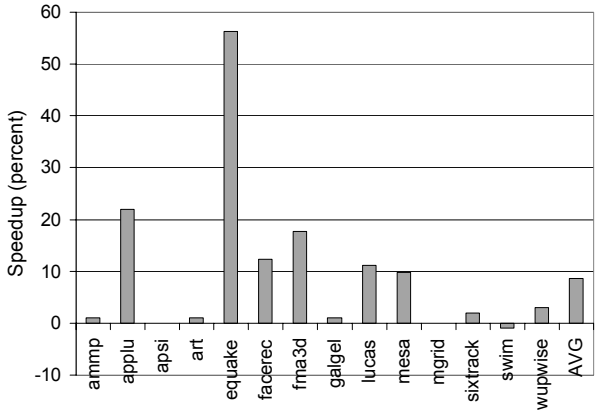

Figure 6b Floating Point Benchmark Improvements

calculating a stride at run time outweigh any associated benefits. Ideally one would chose a prefetch distance such that the data arrives from memory just as it is needed. We compute the prefetch distance as the memory latency (estimated as 100 cycles) divided by the time in cycles to execute the loop body (estimated as the number of instructions in the loop multiplied by 1.4 cycles for execution of an average instruction). If this number of iterations turns out to approach (or exceed) the average run length for the stride value in this loop, the prefetch distance is reduced. If there are multiple prefetches to the same cache line, then some of the prefetches can be omitted.

Stride Prefetch Results: Spike with profile-guided prefetching improves SPECint2000 and SPECfp2000 by an average of 2 percent and 9 percent respectively. The results for the individual benchmarks are given in Figures 6a and 6b. Stride profiling is especially effective for equake and applu where improvements are 56% and 22% respectively. These benchmarks contain loops with accesses to regular structures that are allocated at runtime with calls to dynamic memory allocation procedures. The compiler can not insert prefetches because the existence of a regular reference pattern depends upon the runtime behavior of the dynamic memory allocation schemes. However the regular reference pattern can be exposed with profiling, and prefetches inserted with Spike. [Luk01] gives a more complete presentation of ideas and results for stride prefetching.

8. RELATED WORK

OM [Sri94] was the original binary optimizer for Unix applications on Alpha. Alto [Mut01] is also a binary optimizer for Unix applications on Alpha. Etch [Rom97] is an optimizer for Windows applications on the Intel IA32 architecture. Vulcan [Sri01] is an optimizer for Windows applications on both Intel IA32 and Itanium processors.

Ramirez et al. [Ram01] present a detailed analysis of the optimization of a transaction processing workload with Spike.

Operating system restructuring has been studied by Schmidt et al. [Sch98] for the AS400 operating system and Speer et al. [Spe94] for HP-UX. Our work differs in that Spike enables customer optimization of the UNIX kernel specifically for their workload.

Software-controlled data prefetching has been studied extensively as a means to tolerate memory latency. Most work has been focused on using compilers to insert prefetches at the *source* level, with different schemes targeting different types of data access patterns. Mowry *et al.* [Mow92] proposed the first general algorithm for prefetching *array-based* codes. Variations of their algorithm have been implemented in industrial compilers [San97,Ber95, Dos01]. On the other hand, Luk and Mowry [Luk96] proposed three schemes (greedy, jump-pointer, data-linearization) for prefetching *pointer-based* codes. There are two major differences between these schemes and our two prefetching techniques. First, they solely rely on compiler analysis, while our schemes are largely based on profiling feedback. Hence, our schemes are potentially more accurate at the expense of an extra profiling pass. Second, our schemes insert prefetches into the binary directly and hence do not require any source code. This property is particularly attractive when either the source code is not available or re-compilation is infeasible.

Other researchers have also investigated using memory profiles to assist prefetching. Abraham *et al.* [Abr93] demonstrated that selectively prefetching the small number of loads with high cache miss rates as identified by cache simulation can reduce memory stall significantly while only incurring a small amount of prefetching overhead. Mowry and Luk [Mow97] showed that the overhead can be further reduced without sacrificing much prefetching benefits by correlating cache miss rates with execution contexts. More recently, Barnes *et al.* [Bar99] used cache simulations to guide the insertion of stride prefetches into x86 binaries. Comparing these works against ours, we do not use cache simulations to select which loads need to be prefetched. Instead, we use DCPI [And97] (for latency-based data prefetching) or simple heuristics (for stride prefetching) to make that selection. As a result, we achieve significantly lower profiling overhead than the cases where cache simulations are used. This reduced overhead is necessary for building a practical product.

9. CONCLUSIONS

Spike reduces the cache footprint of a program. This improves fetch efficiency and reduces cache misses for instructions. The original Spike work reduced latencies in the application instruction stream. We have extended this work in two directions. First we have applied these techniques to the kernel instruction stream. Second we have applied latency reducing techniques to the data stream. Adding kernel and prefetching optimizations to the original Spike work has raised the overall performance improvements to 40% for TPC-C

ACKNOWLEDGMENTS

We would like to thank Ernie Petrides and Jim Woodward for their work and support for the kernel

project and its performance goals; Brian Allain, Tom Tracy, and Carl Metzger for detailed kernel benchmark data; Bill Gray for support of DCPI; Joe Mario, Lucy Hamnett, Larry Gensch, and John Williams for their work turning Spike and its support tools into shipping products; and Gene Albert for his work on the basic block database.

REFERENCES

[Abr93] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of Load/Store Instruction Latencies. Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture, pages 139-152, December 1993.

[Alb99] G. Albert. A Transparent Method for Correlating Profiles with Source Programs. 2nd Workshop on Feedback Directed Optimization, Haifa, Israel, Nov. 1999.

[And97] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandervoorde, C. Waldspurger, W. Weihl. Continuous Profiling: Where Have All the Cycles Gone?. ACM Trans. on Computer Systems Vol. 15, Nr. 4, pp. 357-390, 1997.

[Bar98] L. A. Barroso, K. Gharachorloo, E. Bugnion,. Memory System Characterization of Commercial Workloads. 25th Intl. Symposium on Computer Architecture, pp. 3-14, June 1998

[Bar99] R. Barnes, R. Chaiken, and D. M. Gillies. Feedback-Directed Data Cache Optimizations for the x86. 2nd ACM Workshop on Feedback-Directed Optimizations, November 1999.

[Ber95] D. Bernstein, D. Cohen, A. Freund, and D. E. Maydan. Compiler Techniques for Data Prefetching on the PowerPC. Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques, pages 19-26, June 1995.

[Bur00] M. Burrows, U. Erlingson, S-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, W. Weihl. Efficient and Flexible Value Sampling. 9th Intl. Conference of Architectural Support of Programming Languages and Operating Systems, pp. 160-167, November 2000.

[Cohn97a] R. Cohn, D. Goodwin, P. G. Lowney. Optimizing Alpha Executables on Windows NT with Spike. Digital Technical Journal, vol. 9, no. 4, pp. 3-20, 1997

[Cohn97b] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: An Optimizer for Alpha/NT Executables. The USENIX Windows NT Workshop, Seattle, Wash., pp. 17ñ24, August 1997

[Dea97] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-level Profiling on Out-of-order Processors. Proc. 30th Annual Intl. Symp. on Microarchitecture, Dec. 1997.

[Dos01] G. Doshi, R. Krishnaiyer, and K. Muthukumar. Optimizing Software Data Prefetches with Rotating Registers. Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, September 2001.

[Luk96] C. K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 222-233, October 1996.

[Luk01] C.K. Luk, et al.. Profile-Guided Post-Link Stride Prefetching. in preparation for submission

[Mow92] T. C. Mowry, M. S. Lam, and A. 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, pages 62-73, October 1992.

[Mow97] T. C. Mowry and C. K. Luk.. Predicting Data Cache Misses in Non-Numeric Applications through Correlation Profiling. Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, pages 314-320, December 1997.

[Mut01] R. Muth, S. Debray, S. Watterson, K. De Bosschere. Alto: A Link-time Optimizer for the Compaq Alpha. Software-Practice and Experience, 31(1), pp 67-101, 2001

[Obj01] Object File and Symbol Table Format Specification. in the programming documentation for Tru64 Unix Version 5.1 available online at http://tru64unix.compaq.com

[Pet90] K. Pettis and R. C. Hansen. Profile Guided Code Positioning. Proc. ACM SIGPLAN Conf. on

Programming Language Design and Implementation, pp 16-27, June 1990.

[Ram01] A. Ramirez, L. A. Barroso, K. A. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, M. Valero. Code Layout Optimizations for Transaction Processing Workloads. Proceedings of the 28th Intl. Symposium on Computer Architecture, pp. 155-164, June 2001

[Rom97] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen. Instrumentation and Optimization of Win32/Intel Executables. 1997 Usenix Windows NT Workshop, pp 1-8, August 1997

[San97] V. Santhanam, E, Gornish, and W.-C. Hsu. Data Prefetching on the HP PA8000. Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 264-273, June 1997.

[Sch98] W. Schmidt, R. Roediger, C. Mestad, B. Mendelson, I. Shavit-Lottem, V. Bortnikov-Sitnitsky. Profile-directed Restructuring of Operating System Code. IBM Systems Journal, 37(2), 1998

[Smi91] M. Smith. Tracing with Pixie. Tech. Rpt. CSL-TR-91-497, Stanford University, Nov. 1991

[Spe94] S. E. Speer, R. Kumar, and C. Partridge. Improving UNIX Kernel Performance Using Profile Based Optimization. 1994 Winter USENIX, pp. 181-188, Jan. 1994

[Spec01] Standard Performance Evaluation Corp, http://www.spec.org

[Sri94] A. Srivastava and D. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, June 1994

[Sri01] A. Srivastava, A. Edwards. Vulcan: Binary Transformation in a Distributed Environment. Microsoft Research Tech. Rpt. MSR-TR-2001-50, April 2001

[Tpc01] Transaction Processing Council, http://www.tpc.org