SPAID: Software Prefetching in Pointer- and Call-Intensive Environments

Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger
IBM Corporation
3705 Highway 52 North
Rochester, MN 55901

Abstract

Software prefetching, typically in the context of numeric- or loop-intensive benchmarks, has been proposed as one remedy for the performance bottleneck imposed on computer systems by the cost of servicing cache misses. This paper proposes a new heuristic–SPAID–for utilizing prefetch instructions in pointer- and call-intensive environments. We use trace-driven cache simulation of a number of pointer- and call-intensive benchmarks to evaluate the benefits and implementation trade-offs of SPAID. Our results indicate that a significant proportion of the cost of data cache misses can be eliminated or reduced with SPAID without unduly increasing memory traffic.

1. Introduction

It is well known that processor clock speeds are increasing exponentially over time, while memory speeds are not increasing nearly as rapidly [RD94]. The computing industry has reached the point where system performance is dominated by the cost of servicing cache misses. To address this problem, several instruction set architectures (e.g. PowerPC [IBM93]) include non-blocking prefetch instructions that allow the hardware to overlap cache misses with other useful work. It is incumbent upon the designers of optimizing compilers to find ways to utilize prefetch instructions to reduce both the cost and the frequency of cache misses.

This paper proposes a compile-time heuristic called *SPAID (*speculatively prefetching anticipated interprocedural dereferences) for inserting prefetches into the instruction stream to reduce both the cost and the frequency of a certain class of data cache misses. Specifically, the heuristic considers pointers that are passed as arguments on procedure calls, and inserts prefetches at the call sites for the data referenced by the pointers. The fundamental premise of this heuristic is that pointer arguments passed on procedure calls are highly likely to be dereferenced within the scope of the called procedure.

2. Related work

Since the introduction of cache memories [Smi82, Prz90], researchers have continually sought to improve their performance. Some investigators have concentrated on improving the caches themselves, by such techniques as placing caches on the same chip as the processor [ACH+87], inventing non-blocking caches that can tolerate multiple outstanding misses [Kro81, SD88, SF91], or adding additional hardware features to reduce the probability and latency of cache misses [FP89, Jou90]. Others have considered modifying programs themselves to better utilize existing cache hardware.

Efforts to improve instruction cache behavior of programs have their roots in methods to improve paging behavior of main memory [HG71, Fer74, Har88]. A popular area of research has been repositioning of code sections by the compiler, both at the basic block level and the procedure level [HC89, GC90, PH90, CMH91, Wu92]. Some such methods operate on the executable after link time, allowing intermingling of basic blocks from different procedures [Hei94b, Hei94a], while others take into account the branch prediction architecture of the hardware [CG94]. McFarling [McF91] has investigated the use of cache parameters in selecting procedures to be inlined. Mendlson et al. have shown how to avoid conflict misses of instructions in loops [MPS94].

Improvements to the data cache performance of programs have primarily been limited to scientific code that operates on loops. Many of these investigations focus on analysis of data utilization to guide program transformations, particularly on loops, to improve data locality [ASKL81, GJG88, CK89, FST91, LRW91, WL91, KM93, CMT94]. With the advent of prefetch instructions, other researchers have sought ways for compilers to intelligently insert such instructions to improve data cache performance. Most such techniques do not require additional hardware support [Por89, CKP91, FP91, MG91, MLG92, Mow93, BCF95], while others use a combined hardware/software approach [KL91,CMCH91]. Chen and Baer [CB94] have argued that a pure hardware prefetching approach can outperform software prefetching techniques.

Most of the data prefetching techniques discussed above apply principally to loop-intensive scientific applications (in [CMCH91] Chen et al. propose hardware prefetch buffers to support aggressive prefetching for non-numeric benchmarks, while Abraham et al. study the predictability of load/store latencies in non-numeric codes in [ASW+93]). However, improvement of cache performance is crucial to more general applications running on standard hardware. Various studies have shown [CB93, MDO94] that nonscientific workloads such as operating system code can be heavily dependent upon cache behavior, and that real workloads tend to require better cache performance than industry benchmarks indicate [GHPS93]. The intent of our research is to provide a first step in addressing the problem of data cache latency in general-purpose code that is both pointer- and call-intensive.

3. Benchmarks

To evaluate the efficacy of the *SPAID* heuristic, we selected a set of pointer- and call-intensive benchmarks written in C and C++, several of which are well-known and have been used in previous studies. The benchmarks and their input sets are described in Table 1.

TABLE 1. Benchmark Set

Benchmark	Description (Language)	Input Set
xlisp	Lisp Interpreter (C)	Six queens lisp program
gcc	Phase cc1 of Gnu C Compiler (C)	insn-recog.c
groff	Text Formatter (C++)	groff man page (40K)
idl	OMG IDL parser (C++)	somcls.idl definition file
gperf	Hash Generator (C++)	scrabdict.200 (200 words)
sched	Instruction scheduler (C++)	eightq C program object
spaid	Cache Model (C/C++)	short trace of gcc (17K)

The first two benchmarks, *xlisp* and *gcc*, are part of the SPEC integer suite [spe89], and have been studied extensively in the past. The next two, *groff* and *idl*, are C++ benchmarks that have been also been included in previous studies. The fifth benchmark, *gperf*, is Gnu's perfect hash function generator implemented in C++. The sixth benchmark, *sched*, is a post-pass speculative instruction scheduler for the PowerPC architecture written in C++ [DLS93]. The final benchmark, *spaid*, is the trace-driven cache model used to collect the data presented in this paper. It is written primarily in C++, although pieces of the low-level cache directory code are written in C with C++ object wrappers.

TABLE 2. Benchmark Characteristics

Bench mark	Run Length	Call Sites	Proc. Length	Touch Sites (% calls)
xlisp	52.10M	1.36M	38.33	1.15M (85%)
gcc	146.14M	1.94M	75.23	1.36M (70%)
groff	118.90M	5.55M	21.42	4.25M (77%)
gperf	7.81M	222K	35.22	105K (47%)
idl	10.84M	356K	30.45	331K (93%)
sched	78.21M	4.38M	17.87	3.61M (83%)
spaid	99.43M	5.03M	19.78	5.01M (100%)
Total	**513.43M**	**18.84M**	**27.26**	**15.82M (84%)**

The relevant characteristics of these benchmarks are shown in Table 2. The four columns show run length in instructions, a dynamic count of procedure calls, average procedure length, and the number and fraction of those calls that passed at least one pointer argument. The data indicate that all seven benchmarks are both pointer- and call-intensive, making them less amenable to previously reported approaches to data cache optimizations and/or software prefetching.

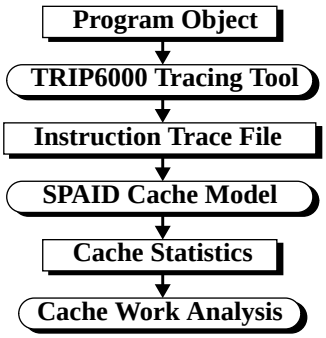

FIGURE 1. Experimentation Framework

4. Experimentation framework

Our experimentation framework, summarized in Figure 1, consists of three main phases: trace collection, cache simulation, and cache work analysis. All benchmarks were run to completion and were compiled with the IBM XL family of compilers (xlC for C++ and xlc for C) at full optimization under AIX version 3.2. However, inlining was explicitly disabled to enable us to detect all call sites during program execution.

4.1. Trace collection

We used the TRIP6000 instruction tracing tool to collect complete instruction and address traces of all the programs in our benchmark set. TRIP6000 is an early version of a software tool developed for the IBM RS/6000 that captures all instruction and address references made by the CPU while in user state. Supervisor state references between the initiating system call and the corresponding return from system call are lost.

The traces were post-processed to insert data cache prefetches at call sites that pass one or more pointer arguments. For one set of measurements, the post-processor was restricted to placing a single prefetch at each call site. For a second set, each call site was allowed up to two prefetches.

4.2. Cache simulation

The traces collected in the first step were then used to drive a diverse set of data cache models. The results presented are limited to cache sizes between 4K and 32K, primarily because the working sets of the benchmarks used do not sufficiently exercise caches larger than 32K. Other parameters that were varied were line size (between 16 and 256 bytes) and associativity (both direct-mapped and 4-way set-associative) [Smi82, Prz90].

4.3. Cache work analysis

To quantify the performance gains due to the *SPAID* heuristic, we compute a measure called *CW* (cache work*)* that approximates processor cycles spent executing each benchmark program. *CW* is defined as follows:

$$CW = I \times CPI_{\infty} + I_{miss} \times I_{latency} + D_{miss} \times D_{latency} + \sum_{i=0}^{D_{latency}} M_i \times (D_{latency} - i)$$

In the above equation, the first term is the product of *I (*the number of instructions executed) and CPI_{∞} (the perfect-cache average instruction latency in cycles per instruction). Since we do not model processor pipeline characteristics in our experiments, we assume a conservative CPI_{∞} value of one, which will tend to understate the *cache work* improvement provided by *SPAID*. The second term accounts for the contribution of instruction cache misses and is the product of I_{miss} (the number of instruction cache misses) and $I_{latency}$ (the latency in cycles per miss). The third term accounts for the contribution of data cache misses, excluding those preceded by a prefetch, and is the product of D_{miss} (the number of data cache misses) and $D_{latency}$ (the latency in cycles per miss). The final summation term accounts for the cycles that the processor must stall to complete data cache references that were preceded by a prefetch of the same cache line; for each distance *i* from zero to $D_{latency}$, the product of M_i (the number of data cache references that were preceded by a prefetch miss at a distance of *i* cycles) and the remaining latency ($D_{latency}$ - *i*) is added to the total *cache work*.

5. Results

We chose to report three different types of results from our measurements: the effect of *SPAID* on data cache misses, the effect of *SPAID* on cache work as computed above, and the effect of *SPAID* on memory traffic.

5.1. Cache misses

In Figure 2, we see the worst, average, and best case effects of *SPAID* on our benchmark set, given data cache sizes of 4K to 32K with 64-byte lines. The misses are presented according to the Three-Cs model of cache performance [Hil87], where the components of the stacked bar chart show compulsory, capacity, and conflict misses (the compulsory misses are not visible on the chart because they are insignificant relative to the capacity and conflict misses). The breakdown was approximated by counting misses to a very large

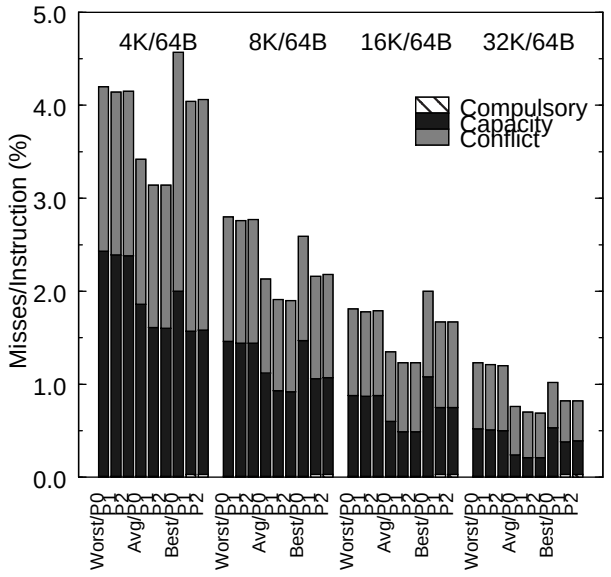

FIGURE 2. Cache Misses vs. Cache Size

(32MB) cache as compulsory misses, additional misses to a 4-way associative cache as capacity misses, and additional misses to a direct-mapped cache as conflict misses.

Each cache size is shown with three sets of three bars, one set for the worst case benchmark (*gcc*), one set for the average of all the benchmarks, and one set for the best case benchmark (*idl*). From left to right, each set has three bars; one for the miss rates without the *SPAID* heuristic (P0), one with the *SPAID* heuristic limited to one prefetch per call site (P1), and one with the *SPAID* heuristic limited to two prefetches per call site (P2).

TABLE 3. Prefetch Results: 8K, direct-mapped, 64B line

Bench mark	Prefetch 1 Pointer Per Call Site			Prefetch 2 Pointers Per Call Site		
	Pref. Count	Miss Rate	Use-ful	Pref. Count	Miss Rate	Use-ful
xlisp	1.15M	8.8%	75.3%	1.89M	7.4%	60.0%
gcc	1.36M	9.9%	61.1%	2.27M	5.5%	41.9%
groff	4.25M	10.4%	73.5%	7.00M	8.6%	41.0%
gperf	105K	31.1%	83.1%	162K	22.4%	77.7%
idl	331K	20.5%	52.8%	633K	13.3%	39.0%
sched	3.61M	4.3%	63.9%	4.60M	4.9%	51.6%
spaid	5.01M	7.0%	47.5%	7.73M	5.8%	46.7%
Σ/GM	**15.82M**	**10.8%**	**64.2%**	**24.29M**	**8.4%**	**49.7%**

On average, from 5-7% (depending on cache size) of miss penalties are eliminated or reduced with just a single prefetch per call site. For the best case benchmark (*idl*), 12-20% are affected. In the worst case (*gcc*), only 1.5% of misses are affected. We can attribute this worst-case behavior to a combination of three factors that differentiate *gcc* from the other benchmarks. First of all, *gcc* has fewer call sites due to longer procedures (its procedures are 75 instructions long on average, against an average of 27 for the benchmark set) and hence provides fewer opportunities for inserting prefetches. Second, it has a lower percentage of call sites that pass pointers as arguments (70% vs. an average of 84%), which again reduces the number of opportunities for inserting prefetches. Third, it has a relatively low prefetch miss rate of 9.9% (i.e. the items that are prefetched tend to already be resident in cache). In contrast, *idl* has an average procedure length of 30 instructions, 93% of call sites pass one or more pointers as arguments, and its prefetch miss rate is 20.5%.

In general, we observe that inserting multiple prefetches per call site does not provide significant additional performance benefits beyond inserting just a single prefetch. Table 3 summarizes the behavior of *SPAID* for our benchmark set with a direct-mapped 8K data cache with 64 byte lines, and illustrates two significant trends that occur as *SPAID* attempts to insert multiple prefetches per call site. First of all, while the absolute number of prefetches increases (in some cases quite significantly), the prefetch miss rates tend to decrease. Second, the percentage of those prefetch misses that are useful (i.e. actually referenced later in the trace) also decreases. These two factors combine to reduce the relative efficacy of the heuristic and to increase both cache conflicts and memory traffic, leading in most cases to performance that is equivalent to or slightly degraded from that of the single-prefetch version.

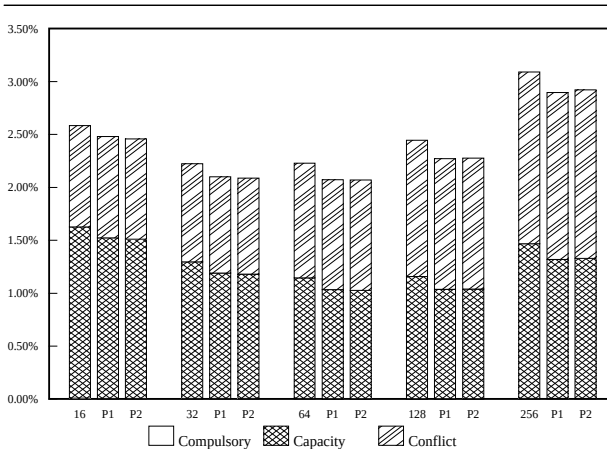

FIGURE 3. Effect of Line Size

These trends indicate a need for further research in the area of selecting which pointers to prefetch at call sites. Of the benchmarks in our set, we had better success with those written in C++ simply because we selected as the prefetch target the first argument passed on each call. In the xlC implementation of C++, the first argument is always the *this pointer*, which, intuitively, has a very high probability of being dereferenced in the ensuing method call. Clearly, since *SPAID* is speculative in nature, better heuristics are needed to control and select prefetch targets in environments where the choice is not as obvious as it is in C++, or where multiple prefetches are both supported by hardware and potentially useful.

In Figure 3 we see the effects of line size on our benchmark set. Again, we used an 8K cache, since this size illustrates the prevailing trends effectively, and varied the line size from 16 bytes to 256 bytes. The three bars shown for each line size indicate miss rates without the *SPAID* heuristic, with a single prefetch per call site (*P1*) and with two prefetches per call site (*P2*). For this set of benchmarks, a 64-byte line appears to be the most effective, both in terms of absolute miss rates as well as *SPAID*'s efficacy. However, we observe that *SPAID* is fairly robust in that it reduces misses to approximately the same degree regardless of line size (4-7% for the line sizes shown). Robust behavior with a variety of line sizes is desirable, since previous studies have shown that line sizes longer than 64 bytes can be beneficial in certain important environments (e.g. transaction processing [MDO94]).

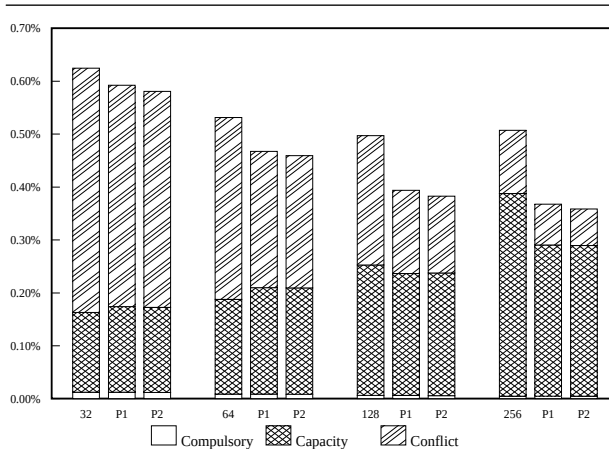

FIGURE 4. Effect of Line Size (*spaid*)

One of our benchmarks (*spaid*) displayed anomalous behavior with respect to the line size trends shown in Figure 3. This behavior is best demonstrated in Figure 4, where miss rates are shown for *spaid* in a 32K cache with line sizes varying from 32 to 256 bytes. Both total misses and the effectiveness of SPAID improve with larger cache lines, all the way up to 256 bytes. This behavior is explained by the large objects (cache directories) referenced in the *spaid* cache model code. Both the implicit hardware prefetching brought about by longer lines and the explicit software prefetching of longer lines are more effective with this benchmark, since larger pieces of the modeled cache directories are prefetched. In the 16K and smaller caches, however, the miss rate minimum occurs at 128-byte lines, due to the excessive conflict misses caused by the larger 256-byte lines.

TABLE 4. Cache Work 8K/64B/ 4-way Data Cache

		Prefetch 1 Pointer Per Call Site			Prefetch 2 Pointers Per Call Site		
Bench mark	**Base WPI**	**WPI**	**IMP**	**OPP**	**WPI**	**IMP**	**OPP**
xlisp	**1.29**	1.27	98.5%	18.8%	1.27	98.3%	22.0%
gcc	**1.72**	1.71	99.9%	0.8%	1.71	99.9%	0.8%
groff	**2.12**	2.12	99.6%	3.6%	2.11	99.5%	4.8%
gperf	**1.28**	1.22	95.3%	27.0%	1.22	95.2%	27.2%
idl	**1.48**	1.43	97.2%	15.7%	1.43	96.8%	18.1%
sched	**1.41**	1.41	99.7%	5.5%	1.41	99.4%	10.3%
spaid	**1.45**	1.43	98.6%	9.1%	1.43	98.1%	12.2%
Mean	**1.66**	**1.65**	**98.3%**	**7.3%**	**1.65**	**98.1%**	**9.1%**

5.2. Cache work

In Table 4 we show the computed cache work results for each of the benchmarks, given an 8K, 2-way set associative instruction cache with 64-byte lines; an 8K, 4-way set associative data cache with 64-byte lines; and an average cache miss latency of 18 processor cycles[1]. The *WPI* columns report work per instruction, the *IMP* column reports improvement relative to the baseline case, and the *OPP* column reports the percentage of total improvement opportunity that *SPAID* successfully exploits (*OPP* is defined as the improvement in the *WPI* divided by share of *WPI* that is caused by data cache misses, i.e. the share of *WPI* that *SPAID* potentially could exploit.). We report weighted arithmetic means for *WPI* and geometric means for *IMP* and *OPP.*

The *OPP* column in Table 4 indicates that *SPAID* is quite effective at reducing data cache stall cycles, eliminating 27% of them in the best case (*gperf*), and 8-9% on average. While the geometric mean of the improvement in total work is relatively insignificant (less than two percent), we observe that for two of the C++ benchmarks (gperf and idl), significant improvements (3-5%) are achievable. Once again, we find that inserting multiple prefetches per call site pays only marginal dividends.

Figure 5 displays how *SPAID*'s effect on cache work varies with cache size. Once again, the miss latency is set at 18 processor cycles for a 64-byte cache line, while the size of the 4-way set-associative cache is varied from 4K to 32K. Results are displayed only for the best-case *gperf* benchmark. The performance improvement (*IMP*) provided by *SPAID* deteriorates gracefully from 6% down to 3% for the 16K cache, but then drops off to near 0% as cache size increases to 32K. The opportunity (*OPP*) exploited by *SPAID* continues to increase up to 31% for the 16K cache, but then drops off abruptly for the 32K cache, which is large enough to contain the working set of the benchmark (the miss rate is only 0.1% per instruction).

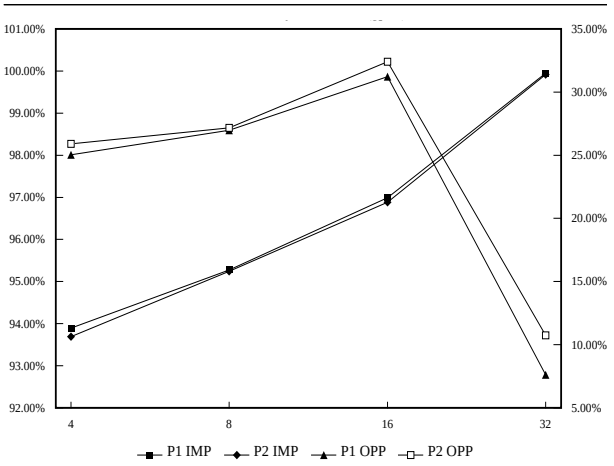

FIGURE 5. Cache Work vs. Size (*gperf*)

In addition to the results shown, we computed cache work data for a broad range of miss latencies for the various cache configurations, but found that the interesting figures of merit (*IMP* and *OPP*) were not very sensitive to memory latency. This leads us to conclude that the prefetches which end up being most useful tend to occur early enough relative to the first use of the prefetched data that latency variations of up to five times (6 to 36 processor cycles) do not change the results noticeably.

5.3. Memory traffic

Figure 6 summarizes the increase in memory traffic for each of the benchmarks measured. The weighted arithmetic means for increases in memory traffic are 3.5% for a single prefetch per call site (*P1*) and 6.6% for two prefetches per call site (*P2*). For the single-prefetch case, even the worst-case increase remains less than ten percent, which is significant only on systems that are already memory bus bandwidth-limited. In addition, *gperf*, the benchmark which benefited most from *SPAID*, experienced only a 1.2% and 2.4% growth for the two cases.

Once again, this data reinforces the need for better heuristics for selecting which pointers to prefetch at call sites. For

1. We assumed two setup and four transfer cycles on a 16-byte memory bus running at one-third of the processor clock speed.

all of the benchmarks except *spaid*, memory traffic increases significantly as we attempt to insert multiple prefetches per call site, with little or no improvement in performance. For *spaid*, on the other hand, the additional prefetches don't cause additional memory traffic since they rarely miss the cache (see Table 3). These observations, in addition to the *Useful* column in Table 3, lead us to conclude that we are, in many cases, prefetching the wrong pointer arguments.

Furthermore, two of the three benchmarks that experienced significant growth in memory traffic (*sched* and *spaid*) frequently reference C++ class objects that are larger than typical cache lines (e.g. 64 bytes in Figure 2). In such cases, given a priori knowledge of which members of a class are likely to be referenced, it might be helpful to prefetch at some non-zero offset from the pointer argument instead of or in addition to prefetching at an offset of zero.

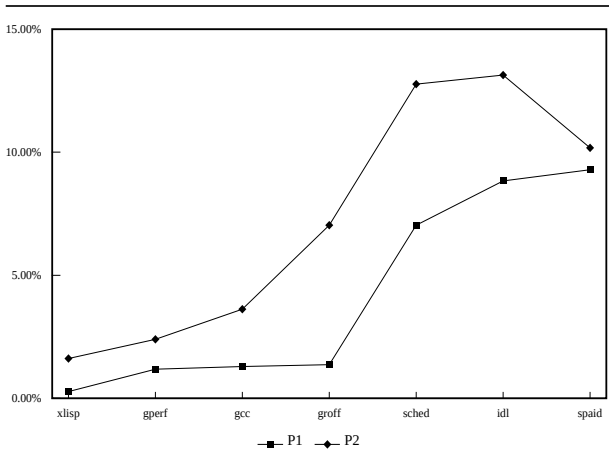

FIGURE 6. Memory Traffic

6. Conclusions and future work

We are encouraged by the early results presented in this paper. Measurable performance gains are possible even with the simple heuristic described here. However, it is clear that significant research remains to be done to fully exploit the idea of prefetching pointer arguments at procedure call sites. We envision future work proceeding in three different areas. First, we must identify and examine benchmarks that exercise larger data caches. These might include multiprogrammed workloads, operating system code, and/or transaction processing workloads. Second, we must validate the early results presented in this paper with more exact models and measurements. This could include implementing the *SPAID* heuristic within an existing compiler for a system that supports non-blocking prefetches and measuring the performance of that system. Alternatively, we could extend our trace-driven approach and replace our statistical *cache work* model with a true cycle-by-cycle simulation model. Third, we must explore heuristics for selecting prefetch targets at call sites that pass multiple pointer arguments. These heuristics might be driven by run-time profile information and/or interprocedural analysis.

Acknowledgments

We want to thank Arturo Martin-de-Nicolas for making the TRIP6000 tracing tool available for our use. We also want to thank Richard Eickemeyer for letting us reuse his cache model code for our simulations. In addition, we acknowledge the contributions of Bilha Mendelson and Mark Funk to the idea of prefetching pointer arguments at call sites.

IBM, PowerPC, PowerPC 601, AIX, and RS/6000 are all registered trademarks of the IBM Corporation.

References

[ACH+87] Anant Agarwal, Paul Chow, Mark Horowitz, John Acken, Arturo Salz, and John Hennessy. On-chip instruction caches for high performance processors. In Paul Losleben, editor, *Proceedings of the 1987 Stanford Conference on Advanced Research in VLSI*, pages 1–24, Cambridge, MA, 1987. MIT Press.

[ASKL81] Walid Abu-Sufah, David J. Kuck, and Duncan H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C–30(5):341–356, May 1981.

[ASW+93] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1993.

[BCF95] David Bernstein, Doron Cohen, and Ari Freund. Compiler techniques for data prefetching on PowerPC. Submitted, 1995.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 120–133, Asheville, NC, December 1993.

[CB94] Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *21st Annual International Symposium on Computer Architecture*, pages 223–232, 1994.

[CG94] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251, San Jose, October 1994.

[CK89] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, December 1989.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, April 1991.

[CMCH91] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-M. Hwu. Data access microarchitecture for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th International Symposium on Microarchitecture*, 1991.

[CMH91] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. Using profile information to assist classic code optimizations. *Software—Practice and Experience*, 21(12):1301–1321, December 1991.

[CMT94] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, October 1994.

[DLS93] Trung A. Diep, Mikko H. Lipasti, and John P. Shen. Architecture-compatible code boosting for performance enhancement of the IBM RS/6000. In *Proceedings of the IEEE International Conference on Computer Design*, pages 86–93, 1993.

[Fer74] Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, November 1974.

[FP89] Matthew K. Farrens and Andrew R. Pleszkun. Improving performance of small on-chip instruction caches. In *16th Annual International Symposium on Computer Architecture*, pages 234–241. IEEE Computer Society Press, May 1989.

[FP91] John W. C. Fu and Janak H. Patel. Data prefetching in

multiprocessor vector cache memories. In *18th Annual International Symposium on Computer Architecture*, pages 54–63, 1991.

[FST91] Jeanne Ferrante, Vivek Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, Santa Clara, August 1991. Springer-Verlag.

[GC90] Rajiv Gupta and Chi-Hung Chi. Improving instruction cache behavior by reducing cache pollution. In *Proceedings of Supercomputing '90*, pages 82–91, New York, November 1990.

[GHPS93] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, pages 17–27, August 1993.

[GJG88] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[Har88] Stephen J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. *IEEE Transactions on Software Engineering*, 14(11):1640–1644, November 1988.

[HC89] Wen-Mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual International Symposium on Computer Architecture*, pages 242–251, Jerusalem, May–June 1989.

[Hei94a] Randall R. Heisch. FDPR for AIX executables. *AIXpert*, pages 16–20, August 1994.

[Hei94b] Randall R. Heisch. Trace-directed program restructuring for AIX executables. *IBM Journal of Research and Development*, 38(5):595–603, September 1994.

[HG71] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 3:168–192, 1971.

[Hil87] M. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, The University of California at Berkeley, 1987.

[IBM93] IBM Microelectronics Division, Essex Junction, VT. *PowerPC 601 RISC Microprocessor User's Manual*, 1993.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, May 1990.

[KL91] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *18th Annual International Symposium on Computer Architecture*, pages 43–53, 1991.

[KM93] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Sixth International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, August 1993. Published as Lecture Notes in Computer Science, vol. 768.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium on Computer Architecture*, pages 81–87. IEEE Computer Society Press, 1981.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, April 1991.

[McF91] Scott McFarling. Procedure merging with instruction caches. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 71–79, Toronto, June 1991.

[MDO94] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, San Jose, October 1994.

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

[Mow93] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, November 1993.

[MPS94] Abraham Mendlson, Shlomit S. Pinter, and Ruth Shtokhamer. Compile time instruction cache optimizations. *ACM Computer Architecture News*, 22(1):44–51, March 1994.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, June 1990.

[Por89] Allan Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, 1989. Technical Report Rice COMP TR89–93.

[Prz90] Steven A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[RD94] K. Roland and A. Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, 14(4):59–67, 1994.

[SD88] C. Scheurich and M. Dubois. Concurrent miss resolution in multiprocessor caches. In *Proceedings of the International Conference on Parallel Processing*, pages 118–125, 1988.

[SF91] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Santa Clara, CA, April 1991.

[Smi82] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, 1982.

[spe89] Systems performance evaluation cooperative. *SPEC Newsletter*, 1(1), 1989.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, June 1991.

[Wu92] Youfeng Wu. Ordering functions for improving memory reference locality in a shared memory multiprocessor system. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 218–221, Portland, December 1992.