A Comparative Evaluation of Software Techniques to Hide Memory Latency

Lizy Kurian John and Vinod Reddy
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

Paul T. Hulina and Lee D. Coraor
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802

Abstract

Software oriented techniques to hide memory latency in superscalar and superpipelined machines include loop unrolling, software pipelining, and software cache prefetching. Issuing the data fetch request prior to actual need for data allows overlap of accessing with useful computations. Loop unrolling and software pipelining do not necessitate microarchitecture or instruction set architecture changes, whereas software controlled prefetching does. While studies on the benefits of the individual techniques have been done, no study evaluates all of these techniques within a consistent framework. This paper attempts to remedy this by providing a comparative evaluation of the features and benefits of the techniques. Loop unrolling and static scheduling of loads is seen to produce significant improvement in performance at lower latencies. Software pipelining is observed to be better than software controlled prefetching at lower latencies, but at higher latencies, software prefetching outperforms software pipelining. Aggressive prefetching beyond conditional branches can detrimentally affect performance by increasing the memory bandwidth requirements and bus traffic.

Keywords: Compiler Optimization, Data Prefetching, Loop Unrolling, Memory Latency, Software Cache Prefetching, Software Pipelining, Static Scheduling.

1 Introduction

Processor speeds have increased tremendously in the past few years, but memory systems have barely kept pace, widening the speed disparity between processors and memory systems. Several software and architectural techniques have been proposed in the past to hide and/or decrease memory access times by prefetching data and overlapping access delays with useful computations. Loop unrolling and static scheduling within the large loop, software pipelining [15] [5] [7], software controlled prefetching [20] [8] [12] [2] [4] [18] [23], lock-up free caches and non-blocking loads [13], hardware cache prefetching [10] [3], etc. are techniques aimed at overlapping accessing with computations and hiding memory access delays. In this paper, we consider three primarily software techniques to hide latency (i) loop unrolling and static scheduling within the unrolled loop (ii) software pipelining and (iii) software cache prefetching.

Loop unrolling is a basic block enlargement technique in which several copies of the original loop body are concatenated to form a large new loop body. Careful scheduling of load instructions within the new large basic block often increases the time between the data request and data consumption and allows latency of accessing to be overlapped with computations. Melvin and Patt [17] showed that basic block enlargement and scheduling the memory access instructions within the enlarged block can reduce memory access delays in pipelined computers.

Software pipelining is a technique that can be used to overlap loads, computations and stores of different iterations in program loops. Software pipelining has been shown to be very effective for VLIW architectures [15] [5] and architectures such as the IBM RS/6000 [24] and the Cydra [21] which provide hardware support for software pipelining.

Software controlled prefetching is a technique in which programs are analyzed at compile-time and special prefetch instructions that load data into a cache (or prefetch buffer or local memory) are inserted ahead of the actual reference for data. The actual load instructions which follow the prefetch instructions find the data in the cache or prefetch buffer or local memory, which are faster than the main memory. Software cache prefetching requires instruction set architecture (ISA) and microarchitectural changes, but it may still be considered as a primarily software technique. In the past few years, there has been extensive research in software cache prefetching [2] [20] [12] [4]. Porterfield et al. [2] [20] presented the software cache prefetching strategy, and showed that it improves cache performance. They also observed that the overhead of executing the prefetches, the increased data traffic and unnecessary prefetches may nullify the benefits, but noted that further optimizations are possible. Gornish et. al. [8] presented a prefetch algorithm that prefetches data into a fast local memory. Klaiber and Levy [12] illustrated the software prefetching technique for a MIPS [11] style RISC processor. They showed how loop unrolling together with the multi-word cache block can be used to reduce the number of prefetch instructions. Mowry, Lam and Gupta [18] incorporate optimizations to avoid unnecessary prefetches and implement a selective prefetch algorithm and compare indiscriminate prefetching with selective prefetching. They show that a selective prefetch algorithm can reduce the overhead associated with software prefetching. For bus-based multiprocessors, software prefetching may result in increasing the bus traffic and reducing the benefits [23].

1.1 Objectives

Our primary objective in this paper is to characterize the features and benefits of loop unrolling, software pipelining and software prefetching techniques in a systematic and consistent manner. Although the different techniques have been individually evaluated in the past, a comparative evaluation portraying the relative merits and demerits of the different techniques in a consistent framework is lacking. These techniques are not in fact mutually exclusive; several of the techniques may be combined in the same system. A quantitative evaluation of the individual and combined techniques in a consistent framework, will enable the architect and the compiler designer to make intelligent decisions during the system design process. We study these techniques as they would apply to simple RISC processors such as the MIPS. The latency sensitivity of the different techniques is studied in detail.

Software controlled prefetching requires changes in the processor microarchitecture and instruction set architecture (ISA) while software pipelining can be implemented without any architectural changes. Klaiber and Levy [12] mention that software controlled prefetching compares favorably with nonblocking LOADs into a large register set, but quantitative results supporting the statement were not presented. We perform a quantitative comparison of software pipelining and software controlled prefetching for a broad range of latencies.

Loop unrolling requires no change in the processor architecture or instruction semantics. In terms of compiler complexity, it is simpler than software pipelining. It would be interesting to see whether loop unrolling and scheduling of loads in the bigger basic block can achieve a performance close to the other techniques. In this paper, we quantitatively analyze the performance of loop unrolling, software pipelining and software prefetching. Since the techniques are not mutually exclusive, software pipelining is combined with loop unrolling, and software prefetching is combined with principles of loop unrolling and software pipelining. A quantitative evaluation of the different techniques individually and in combinations, is presented.

1.2 Overview

The paper is organized into 4 sections. Section 2 describes the architectural assumptions, the benchmarks and the simulator used for the comparative study. In section 3, we present a comprehensive comparison of the three techniques. We compare the execution times with the different techniques and also analyze the hardware and compiler requirements, code size, run-time overhead etc. Section 4 offers concluding remarks.

2 Simulation Methodology

This study is in the context of pipelined RISC processors. We simulate a pipelined uniprocessor architecture with the MIPS instruction set and instruction latencies. The simulated processor supports nonblocking loads and a lockup-free cache [13]. For software controlled prefetching, a *prefetch* instruction as in [12] is added to the instruction set. One of the unused opcodes of the MIPS processor is used to represent this instruction. Trace-driven simulation with a cycle by cycle simulator is used to compare the techniques.

The system bus is 32 bits wide. Hence in the case of double precision data, two memory accesses are required for each data element. In our experiments with double precision data, we found that memory bandwidth was becoming a bottleneck at low latencies itself and the experiments were not yielding any valid results. Since the techniques being studied are latency hiding techniques, in order to see any differences, it is essential that bandwidth does not become a bottleneck. Hence the results presented are obtained from simulations with single precision floating point data, rather than double precision. Single precision simulation on a 32-bit architecture, would apply at least qualitatively to double precision computations on 64-bit architectures, regarding the number of memory access instructions, memory bandwidth requirements, and balance of memory references versus computations.

A four-way sequentially interleaved memory system is assumed for the simulations. Each memory bank is 32 bits wide. All memory bank conflicts during accesses are considered. A 1 Kbyte instruction cache and a 1 KByte data cache are assumed to be present. Our benchmarks are program loops and hence the instruction cache hit-ratio is very high for most cache configurations. The cache sizes are unrealistic for modern microprocessors, but they have been intentionally kept small because our benchmarks are small. The cache block size is 8 bytes because lower block size keeps spurious effects from memory traffic to a minimum. (One cache block can hold two data elements; so the 8 byte block with single precision data would be the equivalent of a 16 byte block in the case of double precision data.) The caches are 4-way set associative, and LRU replacement policy is employed. We did perform experiments with different block sizes, different associativity, etc, and the observations concur to results presented in previous cache studies, and hence we are presenting only the results from one typical cache configuration. The memory access time is varied between 10 processor cycles and 90 processor cycles, so that the impact of latency on the techniques could be studied. The data cache is nonblocking with four Miss Status Holding Registers (MSHRs).

We certainly appreciate the value of being able to incorporate the different techniques in a compiler and perform the experiments automatically as Mowry, Lam and Gupta [18] did and use large benchmarks such as the SPEC. But we faced the same problem as Rogers and Li [22] did, and without a compiler that would apply these optimizations, we were forced to intervene in the code compilation process. Lack of a compiler prevented us from using large benchmarks. We perform simulations with a set of 8 programs that consist of two signal processing algorithms CONV and CORR, the SAXPY routine from Linpacks, the IFLOOP which has a data dependent IF-THEN-ELSE construct inside, three Lawrence Livermore

Loops, an array copy program ACP, etc. All benchmarks involve sequential referencing of arrays and hence spatial locality. The signal processing algorithms, CONV and CORR, involve nested loops and exhibit significant temporal locality in the data reference pattern. The only loop which has an embedded data dependent control dependency is the IFLOOP. This loop has an IF-THEN-ELSE in every iteration, the IF testing a data dependent condition. The different characteristics of these different benchmarks should reveal the different features of the techniques being studied.

Total program execution time is the ultimate measure of performance and we use execution time and a speed up factor based on execution time as the performance metrics. Mowry, Lam and Gupta [18] also used execution time as the metric.

Traces are generated from the assembly output from the DEC station compilers. The benchmark sources are compiled with the highest level of optimization (-O4), and assembly code obtained. We incorporated the different techniques with manual intervention at this stage. Appendix II illustrates the code sequences for various techniques for the SAXPY routine with equal increments. Despite taking a lot of space, and despite the difference between some of the sequences being very minor, several code sequences are illustrated in Appendix II, so that readers could examine them and verify that the study used comparable quality implementations of the different techniques.

The baseline code is obtained from the MIPS compiler output by considering just one body of the loop within each iteration (in the optimized code). The baseline code (illustrated in part (a) of Appendix II) is thus state-of-the-art code, optimized in every respect except unrolling. In our experiments, we limit unrolling to degree one. Naive unrolled code as obtained from the MIPS compilers is illustrated in part (b) of Appendix II. It may be noted that the MIPS compiler allocates different sets of registers for different iterations. Then we rearrange the loads and stores in the code sequence to hide latency. Basically all the loads are moved to the beginning of the loop and all the stores to the end of the loop. The resulting code is illustrated in part (c) of Appendix II.

The code for the software pipelining (illustrated in part (d) of Appendix II) is obtained by rearranging the baseline code so that loads of iteration $i+1$ and computations of iteration i are grouped together. A prologue and epilogue are also added. For this sequence, the basic block size is the same as the baseline code. Principles of loop unrolling and software pipelining are then combined to yield code sequences in part (e) of Appendix II. Two other methods for software pipelining are illustrated in [14]. It may also be noted that we pipeline only the loads. Stores could be buffered at the memory and the latency hidden/alleviated.

The code sequence for software cache prefetching is illustrated in part (f) of Appendix II. Then all optimizations as discussed in [12] and [18] are applied, and the code sequence illustrated in part (g) of Appendix II is obtained. Loop unrolling is applied, and the overhead of prefetching is reduced by avoiding unnecessary prefetches (prefetches that would be cache hits). Principles of software pipelining are applied, and data for next iteration are prefetched during the current iteration.

The example in Appendix II does not have any conditional branches within the loop. When conditional branches are embedded as in IFLOOP, issues regarding anticipatory fetching beyond the conditional branch should be addressed. We generate 3 sets of code with no anticipatory fetches, anticipatory fetches along the IF-path, and anticipatory fetches along both paths, to study the effect of lifting loads above conditional branches.

3 Performance Comparison

In this section, we present a comprehensive comparison of the various techniques, based on our simulation studies. In section 3.1, the performance of the different techniques is compared to baseline performance using execution time and speedup metrics. The latency sensitivity of the execution time is discussed in section 3.2, and the sensitivity of the speedups is presented in section 3.3. Appendix I and Fig 3 in section 3.3 summarize the comparison. Section 3.4 discusses the effect of aggressive prefetching beyond conditional branches. In section 3.5, we compare the performance of software pipelining in an architecture with hardware register renaming to that in simple RISC architectures such as the MIPS. Miscellaneous hardware and software issues are discussed in section 3.6.

3.1 Comparison of Execution Time and Speedup

We performed experiments with loop unrolling, software pipelining (with and without unrolling), and software prefetching (with and without unrolling), at latencies of 10, 20, 30, 60, and 90 processor cycles. The execution time for each technique is presented in Fig 1, for memory latencies of 10, 30, and 90 processor cycles. Naive unrolling as in code sequence (b) of Appendix slightly reduces execution time due to reduction in number of branches and loop incrementing instructions. More significant reduction in performance is obtained by rearranging the loads and increasing the distance between data loads and data use as in sequence (c). At lower latencies, unrolling and rescheduling produce benefits comparable to software pipelining (sequence d) and software prefetching (sequence f). Loop unrolling and further optimizations as in code sequences (e) and (g) produce more significant benefits from software pipelining and software controlled prefetching. The IFLOOP is the only benchmark where the issue of aggressive prefetching beyond conditional branches arises. The results presented in Fig 1 are without aggressive speculative loading (lifting loads above data dependent conditional branches). Some results with speculative loading are presented later in Fig 4.

Appendix I illustrates a comparison of the speed up from the different techniques for latencies 10, 30 and 90 cycles. The speed up is calculated as the ratio of the execution time with the baseline code to the execution time

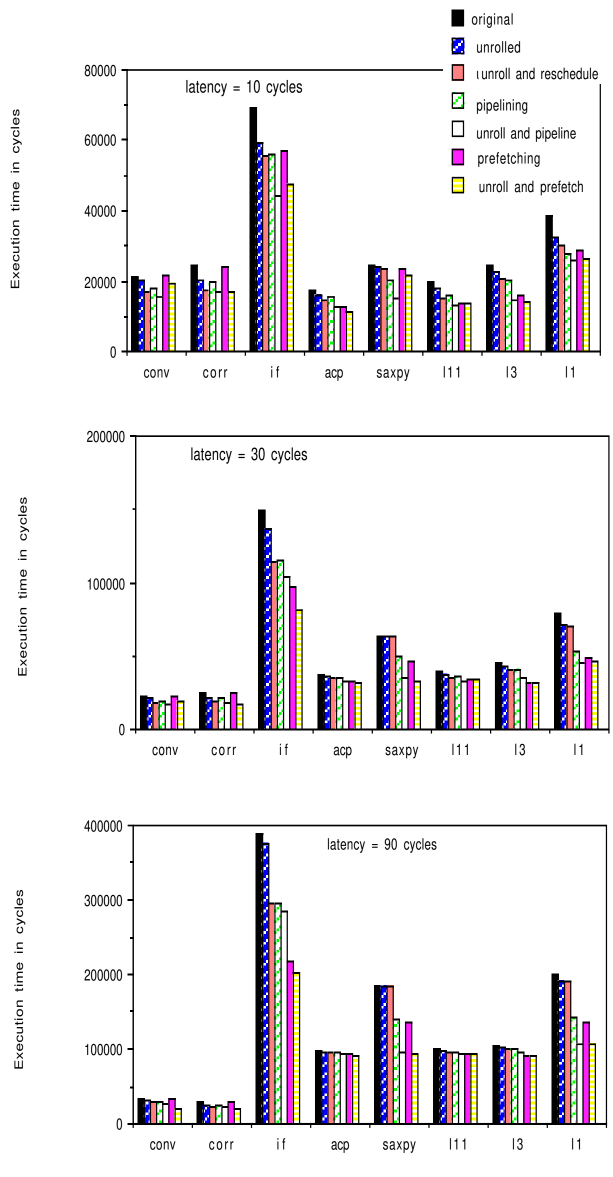

Figure 1: Comparison of Execution Time

with the corresponding technique. The mean speedup presented is the geometric mean of the speedup of the 8 programs.

At $t = 10$ cycles, naive unrolling results in a performance improvement of roughly 11%. Careful static scheduling within the unrolled loop increases the improvement to 23%. At this latency, the performance of naive software pipelining and software prefetching are not higher than that of static scheduling with loop unrolling. At lower latency, software pipelining combined with unrolling is the best technique, and software prefetching with unrolling is the next best technique. At a latency of 10 cycles, software pipelining (unrolled) exhibits better performance than software prefetching (unrolled), in 6 out of the 8 programs. In three of the benchmarks, ACP, L11 and L3, software prefetching is better than software pipelining even at lower latencies. At a latency of 30 cycles, naive software pipelining and software prefetching is better than unrolling and static scheduling in most of the benchmarks. Software prefetching (unrolled) is better than software pipelining (unrolled) in 5 of the 8 programs. Software prefetching (unrolled) is the best technique at a latency of 30 cycles and software pipelining (unrolled) is the next. At latency of 90 cycles, improvement by loop unrolling to degree one becomes less significant than pipelining or prefetching of degree one. Of course it is possible to increase the degree of unrolling and improve the performance. But higher degrees of unrolling increases the number of registers consumed and it may not be practical to incorporate several degrees of unrolling except in small loops. So we do not perform higher degrees of unrolling. At latency of 90 cycles, unrolling and scheduling yields 11% improvement, software pipelining yields 33% and software prefetching gives 48% improvement.

We applied all optimizations as discussed in [18] [12]. In spite of it, at lower latencies, software pipelining appears better than software prefetching. But as the latency reaches 90 cycles, 6 out of the 8 benchmarks show better performance for software prefetching. Thus the simulation results demonstrate that software pipelining is more fruitful at lower latencies and software prefetching is better suited for higher latencies.

3.2 Latency Sensitivity of Execution time

The techniques we are studying are meant to hide latency and hence the performance after incorporating the technique should be less sensitive to latency than original code. Fig 2 illustrates execution time versus latency for three selected benchmarks. (All the 8 benchmarks are not presented due to lack of space and similarity in behavior between the programs. Interested readers can find them in [14].) In most of the benchmarks, software pipelining starts out better, but as latency increases, software prefetching catches up or even outperforms. To be fair in comparison, software pipelining without (with) unrolling should be compared with software prefetching without (with) unrolling. One may observe that in CONV, software prefetching with unrolling exhibits perfect latency

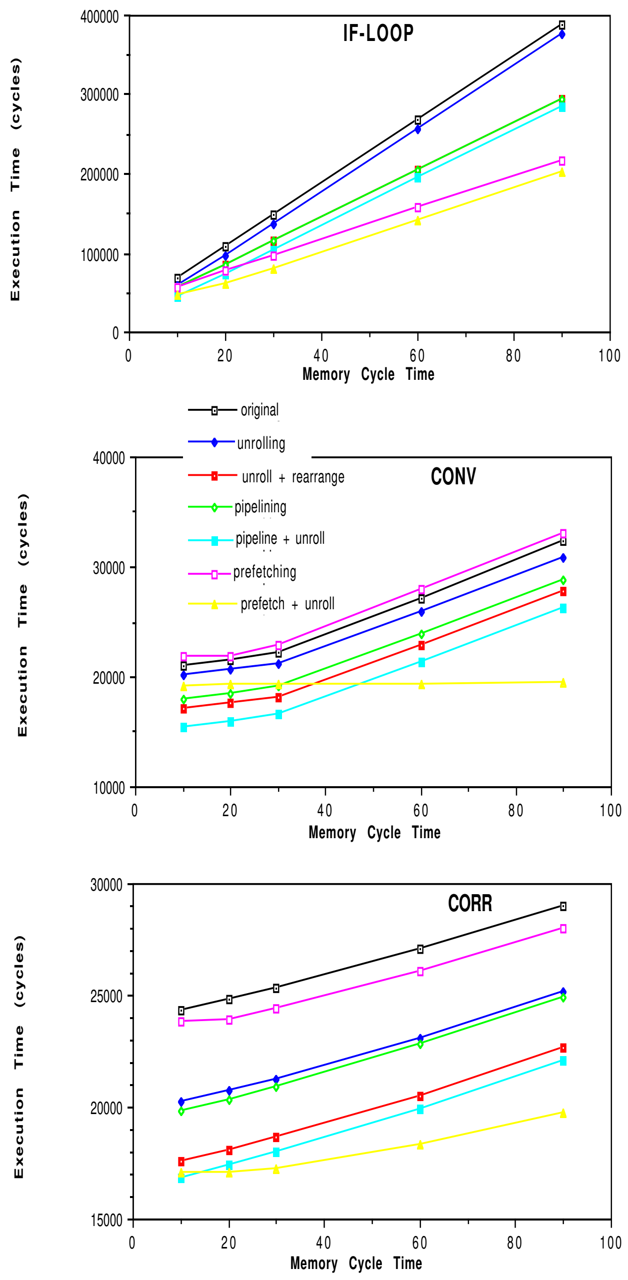

Figure 2: Execution Time vs Latency

insensitivity. Among all the techniques studied, software prefetching (unrolled) exhibits lowest sensitivity to latency in most of the benchmarks.

In medium or large loops, one or two iterations are sufficient to hide lower latencies and hence software pipelining with no runtime overhead behaves better than software prefetching at lower latencies. Software controlled prefetching is a very interesting technique, but there is run-time overhead associated with it. The issuing of each prefetch instruction consumes an extra cycle. If the registers with the addresses cannot be preserved till the actual LOAD, the address calculation may have to be duplicated. (During our trace generation, we always preserved the addresses and avoided duplicate address generation, and hence the effect of this overhead is not evident in our results.) In a naive implementation, software prefetching could easily yield a performance deterioration. Porterfield [20] had obtained increase in execution time with software prefetching. Among the results presented by Mowry et al. [18], in the case of indiscriminate prefetching, 3 out of their 13 benchmark programs exhibited increase in execution time. Among our 8 programs, for CONV, software prefetching without unrolling deteriorates the performance, but when unrolling and other optimizations are applied, the performance improves.

At low latencies, the overhead of executing extra prefetch instructions can deteriorate the performance. But as latency increases, the extra prefetch instructions get executed for free and the overhead associated with prefetching gets nullified. In software prefetching, it is easy to increase the prefetch distance. Software pipelining is more restricted, since a register has to be associated with the preloaded data. Software prefetching does not increase register lifetimes as software pipelining does. Software prefetching has the capability to mask higher latencies than software pipelining.

Loop unrolling has no risks or overheads associated with it that may nullify the benefits of the technique. If sufficient unused registers to unroll the loop are available, compilers may incorporate the technique safely, since loop unrolling is a relatively risk-free approach. Loop unrolling often reduces loop overhead such as branches and indexing, and improves performance. If possible optimizations are not properly incorporated, the speed up in software pipelining and software prefetching may become less than unity, but loop unrolling will never result in less than unity speed up. Unrolling alone without code rearrangement also results in benefits due to reduction in branch instructions and loop overhead.

3.3 Latency Sensitivity of Speed Up

Fig 3 illustrates the comparison for the geometric mean of the speed up for the 8 benchmarks. Several important observations can be made from this figure. The performance improvement for loop unrolling becomes less significant as latency increases. At lower latencies, the performance of unrolling and static scheduling is comparable to naive software pipelining and software prefetching. At low laten-

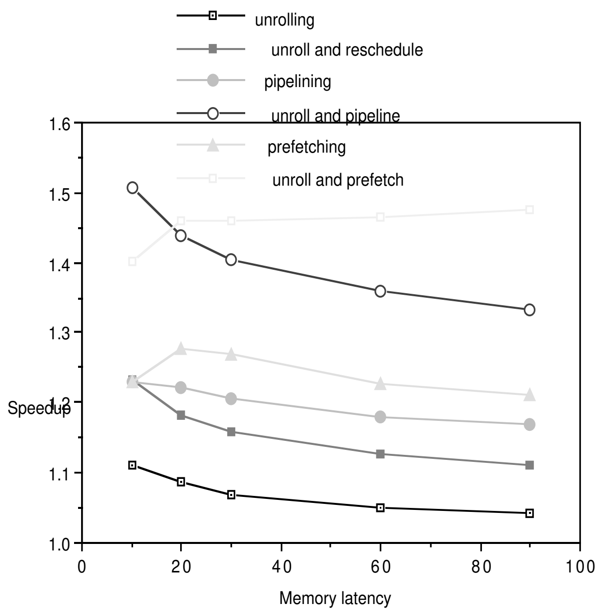

Figure 3: Mean Speed up vs Latency

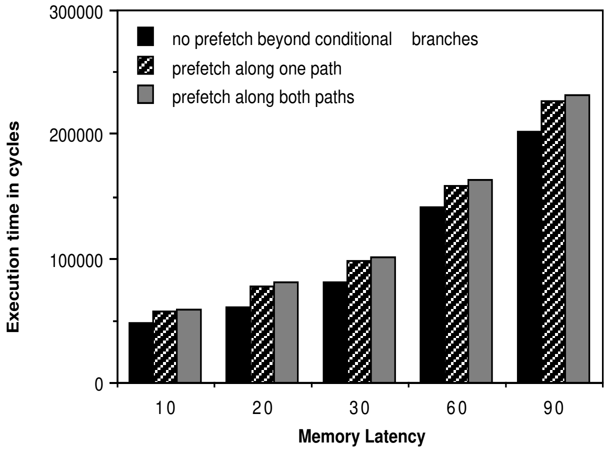

Figure 4: Danger of lifting loads above conditional branches

cies, software pipelining is better than software prefetching. The difference between the two techniques narrows down as latency increases and software prefetching outperforms software pipelining. Loop unrolling and other optimizations can significantly improve the performance of software pipelining and software prefetching. This figure reiterates our conclusion that software pipelining is suitable at lower latencies and software prefetching is more effective at higher latencies. One may refer to Appendix I for exact values of speedup at latencies of 10, 30 and 90 cycles. As latency increases from 10 to 90 cycles, the improvement contributed by static scheduling with unrolling reduces from 23% to 11%, that of software pipelining (with unrolling) reduces from 51% to 33% and that of software prefetching (with unrolling) increases from 40% to 48%.

3.4 Effect of lifting loads above conditional branches

Our results in the previous sections employ no aggressive anticipatory loading (or in other words, we do not lift loads above data dependent conditional branches). In Fig 4, we present results from software controlled prefetching with anticipatory or speculative loads for the IFLOOP program, which has conditional branches embedded in the loop. The execution time with speculative loading of only the load in the THEN case, and speculative loading of the loads in both the THEN and ELSE cases are compared to the case with no speculative loads. In all of the cases that we studied, speculative loading was seen to deteriorate performance from the no speculative load case. This suggests that speculative loading and extra memory traffic from superfluous loads are often detrimental. Rogers and Li [22] explained algorithms to aid in lifting loads above conditional branches, but our studies warn about the overhead of superfluous fetches and extra memory bandwidth requirement that it may create. Our results support the approach of Gornish et al. [8] who incorporated an algorithm in their compiler, to suppress superfluous fetches. We assumed equal probability for the IF and ELSE paths. The results may have not been so pessimistic if there was a higher probability for one of the paths, and prefetches only along that path were performed. But even then, there would be some wasted traffic, and increased bandwidth requirements. Since memory bandwidth is a bottleneck in most high performance architectures, we would not recommend such an approach. Aggressive prefetching will be fruitful only in architectures with extra bandwidth. The effect of superfluous fetches on the performance of prefetching schemes was discussed by Mowry [19] also.

3.5 Compiler Issues

All the techniques studied in this paper require assistance from the compiler. In loop unrolling, the compiler has to unroll loops, manipulate indices accordingly, and rearrange the instructions within the loop to increase the load distances. Modern optimizing compilers perform loop optimizations very efficiently. The compiler assistance required in software pipelining is more complex than that for loop unrolling, since loads have to be moved across iterations. Software controlled prefetching requires still more sophisticated compiler intervention. The compiler has to insert prefetch instructions in such a way to guarantee sufficient load distance to hide latency. At the same time, the data should not be loaded very much earlier than required which may result in replacement of the data from the cache before it is actually used in computations. The compiler should minimize the number of prefetch instructions by unrolling the loops and making use of cache locality, and generating prefetches for only potential cache misses. The compiler has to minimize or suppress anticipatory (superfluous) prefetches.

4 Summary and Concluding Remarks

Loop unrolling, software pipelining and software prefetching techniques were quantitatively compared in this paper. One major conclusion from the study is that at low memory latencies, software pipelining without any hardware support outperforms software cache prefetching which requires ISA and microarchitectural changes. Software prefetching has the run-time overhead of issuing extra prefetch instructions and this overhead may cancel the benefits of prefetching at low latencies. Software prefetching can cause detrimental effects if the compiler is not efficient in applying various optimizations. At higher latencies, software prefetching can lead to better benefits than software pipelining, provided bandwidth has not yet become a bottleneck. Loop unrolling is a very powerful technique, and it is essential to combine loop unrolling with software pipelining and software prefetching in order to obtain true benefits from those techniques. Loop unrolling with static scheduling of loads in the unrolled loop produces a speed up of 1.23 at latency of 10 cycles and 1.11 at latency of 90 cycles. The improvement is smaller than best implementations of software pipelining or software prefetching, but loop unrolling never yields a performance deterioration. As latency increases from 10 cycles to 90 cycles, the speedup from software pipelining changes from 1.51 to 1.33, and that of software prefetching increases from 1.40 to 1.48.

Loop unrolling and software pipelining can be successfully performed in systems with no cache also (assuming the memory is pipelined or non-blocking), whereas software prefetching requires a cache (or a fast local memory). Implicit prefetching and overlapping as in loop unrolling with load hoisting and software pipelining, are possible only in load/store architectures. Explicit prefetching, such as software controlled prefetching can be done even in non load/store architectures. All the techniques are particularly suited for iterative programs.

Another conclusion is that aggressive speculative loading beyond conditional branches is often detrimental to performance. In the IFLOOP benchmark, software controlled prefetching with speculative loading was seen to increase the execution time. Unless carefully designed, any explicit prefetching scheme that lifts loads above conditional branches, can nullify the benefits of prefetching by the increase in superfluous fetches and extra memory traffic. The superfluous fetches increase the memory bandwidth requirement. Memory bandwidth is already a bottleneck in most high performance architectures, and any scheme that aggravates the bottleneck problem cannot be favored. Our studies support a conservative approach regarding speculative loads and prefetches. Perhaps the thrust of future research should be on techniques such as blocking [16] [18] which reduce bandwidth requirements rather than just smoothen requirements as in the techniques studied.

We expect our results to be true for larger programs, but detailed experiments with benchmarks such as the SPEC are required to investigate this.

References

[1] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines", ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, pp. 241-255, June 1991.

[2] D. Callahan, K. Kennedy and A. Porterfield, "Software Prefetching", Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems", April 1991, pp. 40-52.

[3] T-F Chen and J-L Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches", Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems", October 1992, pp. 51–61.

[4] W. Y. Chen, S. A. Mahlke, P. P. Chang, and H. W. Hwu, "Data access microarchitectures for superscalar processors with compiler assisted data prefetching", Proceedings of MICRO-24, 1991.

[5] P. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler", Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems", pages 180-192, October 1987.

[6] DECchip 21064-AA Microprocessor Hardware Reference Manual, Digital Equipment Corporation, 1992.

[7] K. Ebcioglu, "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps", IEEE Micro-20, December 1987.

[8] E. Gornish, E. Granston and A. Veidenbaum, "Compiler directed Data Prefetching in Multiprocessors with Memory Hierarchies", 1990 International Conference on Supercomputing, pp. 354-368.

[9] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry and W-D Weber, "Comparative Evaluation of Latency Reducing and Tolerating Techniques", Proc. of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May 1991, pp.254–263

[10] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully associative cache and buffers", 17th International Symposium on Computer Architecture, 1990, pp. 364–373.

[11] G. Kane, "MIPS RISC Architecture", Prentice–Hall, Englewood Cliffs, N.J., 1988.

[12] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching", 18th Intl.

Symp. on Computer Architecture, May 1991, pp. 43–53.

[13] D. Kroft, "Lockup-free Instruction Fetch/Prefetch Cache Organization", Proc. of the 8th Annual Intl. Symp. on Computer Architecture, pp. 81-87, June 1981.

[14] L. Kurian and V. Reddy, "A Comparative Evaluation of Software Techniques to Hide Memory Latency", University of South Florida, Dept. of Computer Science and Engineering, Technical Report, 1994-02.

[15] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines", ACM SIGPLAN '88 conference on programming Language Design and Implementation, pp. 318 – 328, 1988.

[16] M. S. Lam, E. E. Rothberg and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems", 1991, pp. 63 – 74.

[17] S. Melvin and Y. Patt, "Exploiting Fine–Grained Parallelism Through a Combination of Hardware and Software Techniques", Proc. of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May 1991, pp.287-296

[18] T. C. Mowry, M. S. Lam and Anoop Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems", October 1992, pp. 62 – 73.

[19] T. Mowry, "Tolerating Latency Through Software-Controlled Data Prefetching", Stanford University Technical Report, CSL-TR-94-628, June 1994.

[20] A. K. Porterfield, "Software Methods for Improvement of Cache Performance on Supercomputer Applications", Ph. D. dissertation, RICE COMP TR 89-93, May 1989.

[21] B.R. Rau et al. "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and tradeoffs", IEEE Computer, vol. 22, January 1989, pp. 12-35.

[22] A. Rogers and K. Li, "Software Support for Speculative Loads", Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems", October 1992, pp. 38–50.

[23] D. M. Tullsen and S. J. Eggers, "Limitations of Cache Prefetching on a Bus-Based Multiprocessor", Proc. of the International Symposium on Computer Architecture, May 1993, pp. 278-288.

[24] Warren Jr. H., "Instruction Scheduling for the IBM RISC System/6000 processor", IBM Journal of Research and Development 34(1), Jan 1990, pp. 85 – 91.