Improving Software Pipelining with Hardware Support for Self-Spatial Loads

Steve Carr
Philip Sweany
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
{carr,sweany}@mtu.edu

Abstract

Recent work in software pipelining in the presence of uncertain memory latencies has shown that using compiler-generated cache-reuse analysis to determine proper load latencies can improve performance significantly [14, 19, 9]. Even with reuse information, references with a stride-one access pattern in the cache (called self-spatial loads*) have been treated as all cache hits or all cache misses rather than as a single cache miss followed by a few cache hits in the rest of the cache line.*

In this paper, we show how hardware support for loading two consecutive cache lines with one instruction (called a prefetching load*) when directed by the compiler can significantly improve software pipelining for scientific program loops. On set of 79 Fortran loops when using prefetching loads, we observed an average performance improvement of 7% over assuming all self-spatial loads are cache misses (assuming all hits often gives worse performance than assuming all misses [14]). In addition, prefetching loads reduced floating-point register pressure by 31% and integer register pressure by 20%. As a result, we were able to software pipeline 31% more loops within modern register constraints (32 integer/32 floating-point registers) with prefetching loads. These results show that specialized prefetching load instructions have considerable potential to improve software pipelining for array-based scientific codes.*

1 Introduction

In modern processors, main-memory access time is at least an order of magnitude slower than processor speed. A small, fast cache memory is used to alleviate this problem. However, the cache cannot eliminate all accesses to main memory and programs incur a significant penalty in performance when a miss in the cache occurs. To help tolerate cache miss latency, hardware designers have developed *prefetching* caches, *non-blocking* caches and *software prefetching* instructions. Prefetching caches automatically prefetch the next successive cache line on each cache access [11]. Non-blocking caches allow cache accesses to continue when misses occur [11]. Software prefetching involves an additional instruction to bring a memory location into cache in advance of when a load is issued to store the value in a register [8, 17]. All of these techniques are important to instruction-level parallelism (ILP) because they allow the instruction scheduler to overlap more operations with memory accesses, possibly hiding main-memory latency. Thus, a significant increase in ILP can be achieved.

To take full advantage of the parallelism available in ILP computers, advanced instruction scheduling techniques such as software pipelining have been developed [3, 16, 18, 22]. Software pipelining allows iterations of a loop to be overlapped with one another in order to take advantage of the parallelism in a loop body. While software pipelining can yield significant performance gains by overlapping loop iterations, it can also require significant register resources.

The existence of uncertain memory latencies produces a problem for the instruction scheduler. To deal with this situation, instruction schedulers typically assume that all memory accesses are cache hits, assume that they are all cache misses, or use cache-reuse analysis to predict which accesses are misses and which are hits [14, 19]. Assuming all hits reduces the lifetimes of registers and keeps register pressure to a minimum. However, significant penalties are incurred when a cache miss occurs. Assuming all cache misses tolerates the latency of a cache miss better. Unfortunately, assuming that memory latencies are all cache misses to avoid paying cache-miss penalties will likely lead software pipelining to overlap more lifetimes. This can lead to significantly greater register usage since the register lifetimes get stretched needlessly by the assumption that each load is a cache miss. Additionally, if a reference that is a cache hit is scheduled with a cache-miss latency and that reference appears on a recurrence, the recurrence constraint for software pipelining can be needlessly increased, resulting in degraded performance.

Recently, Sánchez and González [19] and Ding *et al.* [14, 9] report on experiments that show that using memory-reuse analysis to determine which loads are hits and which are misses gives better performance than both the all-cache-hit and all-cache-miss memory models. However, neither method handles loads with a stride-one access pattern in the cache (called *self-spatial loads*) adequately. Sánchez and González assume that a self-spatial load is either always a cache miss or always a cache hit depending upon when assuming a miss increases the recurrence constraint too much. In [14], Ding *et al.* assume that a self-spatial load is always a cache hit. In [9], they assume that a self-spatial load is always a cache miss.

Consider the following loop.

```
DO I = 1, N
    ... = A(I)
ENDDO
```

If we assume that the reference to `A(I)` is a cache miss, we do not take advantage of the fact that $l - 1$ (where l is the cache-line size) out of every l references to `A(I)` is a cache hit. Thus, there is potential needless increase in register pressure due to longer overlapped lifetimes. If we assume that `A(I)` is always a cache hit, we keep the register pressure lower, but we pay the cache miss penalty once out of every l references.

One possible solution to this problem is to use software prefetching instructions to bring in the cache lines for self-spatial loads and schedule them as cache hits [8, 17]. Unfortunately, software prefetching has been shown to sometimes hurt performance due to increased instruction overhead, increased cache interference [13, 19], and increased register pressure due to longer lifetimes for address registers [13]. In this paper, we propose a combination hardware and software solution to the problem of self-spatial load performance that allows us to get the low register pressure similar to assuming self-spatial loads are cache hits and the better performance available by assuming they are all cache misses.

In the remainder of this paper, we begin with an overview of software pipelining in Section 2 with special attention paid to register requirements and pipelining with uncertain memory latencies. Then, we give a discussion of our proposed solution to this problem in Section 3. Section 4 details the experimental evaluation of our proposed technique and Section 5 presents our conclusions.

2 Software Pipelining

While local and global instruction scheduling can, together, exploit a significant amount of parallelism for non-loop code, to best exploit instruction-level parallelism within loops requires software pipelining. Software pipelining can generate efficient schedules for loops by overlapping execution of operations from different iterations of the loop. This overlapping of operations is analogous to hardware pipelines where speed-up is achieved by overlapping execution of different operations.

Allan et al. [3] provide a good summary of current software pipelining methods, dividing software pipelining techniques into two general categories called *kernel recognition* methods and *modulo scheduling* methods. In the kernel recognition technique, a loop is unrolled an "appropriate" number of times, yielding a representation for N loop bodies which is then scheduled. After scheduling the N copies of the loop, some pattern recognition technique is used to identify a repeating kernel within the schedule. Examples of kernel recognition methods are Aiken and Nicolau's perfect pipelining method [2] and Allan's petri-net pipelining technique [4].

In contrast to kernel recognition methods, modulo scheduling does not schedule multiple iterations of a loop and then look for a pattern. Instead, modulo scheduling selects a schedule for one iteration of the loop such that, when that schedule is repeated, no resource or dependence constraints are violated. Lam's *hierarchical reduction* is a modulo scheduling method as is Warter's [21] *enhanced modulo scheduling* which uses *IF-conversion* to produce a single super-block to represent a loop. Rau [18] provides a detailed discussion of an implementation of modulo scheduling.

2.1 An Example

As an example of what software pipelining can do, consider a generalized loop, L1, with four operations we will call *A*, *B*, *C*, and *D*. For purposes of illustration let us assume that dependences require a sequential ordering of these operations within a single loop iteration. Thus, even if our target architecture allows 4 operations to be issued at once, a schedule for a single loop iteration, requiring 4 instructions would be

```
do N times
  A
  B
  C
  D
```

due to dependences among the operations. A software pipelined version of this loop might well be able to issue all 4 operations in one instruction by overlapping execution from different loop iterations. This might lead to a single-instruction loop body of $A^{i+3}B^{i+2}C^{i+1}D^{i}$ where X^j denotes operation *X* from iteration *j* of the loop[1]. Of course, if the loop body is concurrently executing operations from multiple loop iterations in a pipelined fashion, we need a prelude to set up the software pipeline and a postlude to empty it. The entire idealized loop then becomes:

(Prelude)			
A^1			
B^1	A^2		
C^1	B^2	A^3	
(Loop Body)			
do $N-3$ times (with index i)			
A^{i+3}	B^{i+2}	C^{i+1}	D^i
(Postlude)			
D^{N-2}	C^{N-1}	B^N	
D^{N-1}	C^N		
D^N			

2.2 Modulo Scheduling

Our software pipelining implementation is based upon Iterative Modulo Scheduling and follows the method presented by Rau [18]. First a single data dependence graph (DDG) is constructed for the loop body. Once the DDG is constructed, modulo scheduling attempts to identify the smallest number of instructions which might separate different loop iterations. This minimum *initiation interval* (II_{min}) represents the shortest time interval between the initiation of consecutive loop iterations. II_{min} depends upon two characteristics of the DDG for the loop (and the parallelism available in the target architecture):

- The *recurrence* constraint, *RecII*, represents a limit on II_{min} due to dependence arcs within the DDG. The recurrence constraint is due to loop-carried dependences which create cycles in the DDG.

- The *resource* constraint, *ResII*, represents a limit based upon how much “work” the loop requires (how many integer operations, floating-point operations, memory operations, etc.) and how much parallelism is provided by the architecture (how many floats, ints, etc. can be issued in an instruction.) In this context, a resource can be thought of as a functional unit needed to complete some operation.

See Rau [18] for details on how to compute both the *RecII* and *ResII*.

Given both *ResII* and *RecII*, the actual minimum initiation interval (II_{min}) is the maximum of *ResII* and *RecII*. Having computed II_{min}, modulo scheduling next attempts to schedule the DDG in II_{min} instructions. Again following Rau, we use a modified conventional acyclic list scheduling method. If *ResII* is significantly greater than *RecII*, the nodes are scheduled using traditional list scheduling. If, however, II_{min} is close to *RecII*, heuristics are used to give priority to nodes that are in the longest dependence cycle. If a schedule of II_{min} instructions can be found that does not violate any resource or dependence constraints, modulo scheduling has achieved a minimum schedule. If not, scheduling is attempted with $II_{min}+1$ instructions, and then $II_{min}+2$, ..., continuing up to the worst case which is the number of instructions required for local scheduling. The first value of II to produce a “legal” schedule of the DDG becomes the actual initiation interval.

[1]This notation is borrowed from Allan et al. [3]

2.3 Increased Register Requirements

As shown in the example in Section 2.1, software pipelining can, by exploiting inter-iteration concurrency, dramatically reduce the execution time required for a loop. Such overlapping of loop iterations also leads to additional register requirements, however. For illustrative purposes let us reconsider our 4-operation loop, L1. Let us assume that operation *A* computes a value, *v*, in a register and that operation *D* uses *v*. In the initial sequential version of a loop body one register is sufficient to store *v*’s value, since the value computed by the next iteration’s *A* is not available until after *D* has used the value computed by the current iteration. Notice, however, that, in the software pipelined version, we need to maintain several different copies of *v* because we have different loop iterations in execution simultaneously. Specifically we need to have as many registers “assigned” to *v* as we have different iterations of L1 in execution concurrently, namely 4 in our example.

To handle overlapping lifetimes within the kernel, most architectures require software support to produce correct schedules. The most popular technique is Lam’s *modulo variable expansion* (MVE) [16]. MVE overcomes inter-interval dependences by copying the loop body, or kernel, M times, where M is the number of different loop iterations included in the longest lifetime for any variable in the loop. Each register within the (II-length) loop body is then “expanded” to become a group of registers, one per copy of the original loop body, thereby removing conflicts produced by register reuse dependences.

As shown above, overlapping too many lifetimes dramatically increases register pressure. One problem with high register pressure is that most implementations of software pipelining will pipeline a loop only if it requires no more registers than the target machine provides. Therefore, we would like to schedule references that are known to be cache hits with a small latency, thereby limiting register pressure, and references that are known to be cache misses with a longer latency, thereby hiding miss latency when beneficial.

2.4 Recurrence Initiation Interval

In addition to increased register pressure, assuming that a reference is a cache miss may unnecessarily lengthen the *RecII* of a loop. If a memory access appears on a recurrence, assuming a miss for that memory access may unnecessarily lengthen the cycle on which it appears. For cases in which reuse information is able to determine that the access is a cache hit, assuming a miss will result in a pipeline with an unnecessarily long initiation interval; thus, increasing execution time. By using memory-reuse analysis we can eliminate the unnecessary recurrence lengthening and improve loop performance.

3 Method

To improve the cache performance for self-spatial loads, hardware prefetching can be effective. Chen and Baer [11] show that hardware prefetching is an effective technique for hiding memory latency. The problem with hardware prefetching is that the next cache line is prefetched whether or not it will be used. Thus, there is potential for cache pollution. Instead, we propose to augment an architecture with a non-blocking cache with an instruction to fetch two cache lines, called a *prefetching* load, when directed by the compiler. If we use the compiler to determine which loads should use prefetching, we get the performance benefit of prefetching without the cache pollution. In addition, software pipelining with prefetching loads utilizes fewer registers with the same or better performance than assuming that self-spatial loads are all cache misses.

In array-based applications, the compiler can use reuse analysis to determine which references will be self-spatial loads. Thus, only those references that are self-spatial loads will utilize a prefetching load. It is important to note that by bringing in two cache lines with a load, we can, assuming no conflicts, eliminate all cache misses except the first reference in the first loop iteration for self-spatial loads. Loading two cache lines removes the need for a prefetching load to be on a cache-line boundary to eliminate cache misses to the next $l-1$ loads, where l is the cache-line size. Additionally, there is no need to bring in more than two cache lines, thus, limiting cache pollution.

In the rest of this section, we detail the memory-reuse analysis, developed by Wolf and Lam [23], that we use in this paper. We then show, based upon the results of the reuse analysis, how to identify which loads can benefit from prefetching.

3.1 Data Reuse

The two sources of data reuse are *temporal* reuse, multiple accesses to the same memory location, and *spatial* reuse, accesses to nearby memory locations that share a cache line or a block of memory at some level of the memory hierarchy. Temporal and spatial reuse may result from *self-reuse* from a single array reference or *group-reuse* from multiple references [23]. Without loss of generality, we assume Fortran's column-major storage.

In Wolf and Lam's model, a loop nest of depth n corresponds to a finite convex polyhedron Z^n, called an iteration space, bounded by the loop bounds. Each iteration in the loop corresponds to a node in the polyhedron, and is identified by its index vector $\vec{x} = (x_1, x_2, \ldots, x_n)$, where x_i is the loop index of the i^{th} loop in the nest, counting from the outermost to the innermost. The iterations that can exploit reuse are called the localized iteration space, L. The localized iteration space can be characterized as a localized vector space if we abstract away the loop bounds.

For example, in the following piece of code, if the localized vector space, L, is *span*$\{(1,1)\}$, then data reuse for both `A(I)` and `A(J)` are exploited.

```
DO I= 1, N
  DO J = 1, N
    A(I) = A(J) + 2
  ENDDO
ENDDO
```

In Wolf and Lam's model, data reuse can only exist in *uniformly generated* references as defined below [15].

Definition 1 *Let n be the depth of a loop nest, and d be the dimensions of an array* A*. Two references* $\mathsf{A}(f(\vec{x}))$ *and* $\mathsf{A}(g(\vec{x}))$*, where f and g are indexing functions $Z^n \to Z^d$, are* uniformly generated *if*

$$f(\vec{x}) = H\vec{x} + \vec{c}_f \text{ and } g(\vec{x}) = H\vec{x} + \vec{c}_g$$

where H is a linear transformation and $\vec{c}_f$ and $\vec{c}_g$ are constant vectors.

For example, in the following loop,

```
DO I= 1, N
  DO J = 1, N
    A(I,J) + A(I,J+1) + A(I,J+2)
  ENDDO
ENDDO
```

the references can be written as $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, and $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix} + \begin{bmatrix} 0 \\ 2 \end{bmatrix}$. These references are all uniformly generated and we can partition them into different sets, each of which operates on the same array and has the same H. These sets are called *uniformly generated sets* (UGSs).

A reference is said to have *self-temporal* reuse if $\exists \vec{r} \in L$ such that $H\vec{r} = \vec{0}$. The solution(s) to this equation is called the self-temporal reuse vector space or R_{ST}. A reference has *self-spatial* reuse if $\exists \vec{r} \in L$ such that $H_S\vec{r} = \vec{0}$, where H_S is H with the first row set to $\vec{0}$. The solution(s) to this equation is called the self-spatial reuse vector space or R_{SS}. Two distinct references in a UGS, $\mathsf{A}(H\vec{x} + \vec{c}_1)$ and $\mathsf{A}(H\vec{x} + \vec{c}_2)$ have *group-temporal* reuse if $\exists \vec{r} \in L$ such that $H\vec{r} = \vec{c}_1 - \vec{c}_2$. And finally, the two references have *group-spatial* reuse if $\exists \vec{r} \in L$ such that $H_S\vec{r} = \vec{c}_{1,S} - \vec{c}_{2,S}$.

Using the above equations we can partition the UGSs into sets whose members exhibit group-temporal reuse (GTSs) and group-spatial reuse (GSSs). The *leader* of one of these sets is the first reference in the set to access a particular data element or cache line.

3.2 Identifying Self-Spatial Loads

Not all references that have self-spatial reuse need prefetching. For each group-spatial set that has self-spatial reuse, only one reference needs a prefetching load since all of the references access the same set of cache lines. Therefore, we choose the first load in each group-spatial set having self-spatial reuse to utilize a prefetching load.[2]

As an example, consider the following loop.

```
DO J = 1, N
   DO I = 1, N
      A(I,J) = B(I,J) + B(I-2,J) +
               C(I,J)
   ENDDO
ENDDO
```

In this loop, both `B(I,J)` and `C(I,J)` would be loaded using a prefetching load.

Once self-spatial references needing a prefetching load have been identified, we schedule them using the cache-hit latency. If, there is not enough computation to cover the miss latency in l iterations of the loop, we can increase the latency of the prefetching load to ensure the removal of potential idle cycles.

4 Experiment

We have implemented the above analysis in Memoria, a source-to-source Fortran transformer based upon the D System infrastructure [1]. Memoria first performs scalar replacement [10] for array references and then annotates Fortran code with the reuse information. The annotated Fortran code is then converted to annotated Iloc intermediate statements. The Iloc is optimized using the optimization passes provided by the Massively Scalar Compiler Project at Rice University [5]. We use constant propagation, global value numbering [7], partial redundancy elimination [6], operator strength reduction [12] and peephole optimization. Our retargetable compiler, Rocket [20], then reads in the annotated Iloc and schedules it using iterative modulo scheduling [18].

[2]Note that this reference may or may not be the leader.

Our target architecture for this experiment is a superscalar machine that can issue two integer and two floating-point operations per cycle. The architecture requires 3 cycles for both integer and floating-point operations. There is an 8K direct-mapped cache with a 25 cycle miss penalty. The cache is two-way interleaved to allow up to two accesses per cycle. There is a miss buffer with 10 entries. For prefetching loads, both cache modules are used since consecutive cache lines are in opposite modules. So, a prefetching load acts like two loads: one to the specified cache line and one to the next successive cache line. The architecture also assumes that all branches will be taken. Finally, there are 128 integer and 128 floating-point registers.

We tested the use of prefetching loads on 66 Fortran loops from the SPEC benchmarks swm256 and hydro2d with 13 additional loops from standard kernels such as matrix multiply.

4.1 Results

Table 1 summarizes the execution time and software pipelining results for our suite of loops using both memory-reuse analysis that assumes that self-spatial loads are cache misses (Reuse) and memory reuse analysis that uses prefetching loads to make them all cache hits (Prefetch). We do not show results for assuming self-spatial loads are always cache hits because the prefetching load has the same latency as assuming a cache hit and gives better cache performance. Table 1 gives the mean, median and standard deviation for execution cycles, II, *RecII*, integer register pressure and floating-point register pressure.

For the 79 loops we observed an averaged of a 7% increase in performance using prefetching loads over just using memory-reuse analysis. Forty-five loops had an execution-time improvement ranging from 1% to 57%, 29 loops showed no change in performance and 5 loops degraded in performance ranging from 1% to 12%. Five of the loops that showed execution-time improvements, observed around a factor of 2 increase in performance. Each of these loops had a large reduction of *RecII* and II (on the order of 70%). For the majority of the loops, the run-time improvements were smaller and were due to fewer idle cycles as a result of denser schedules with the much shorter memory latencies. The 5 loops that degraded in performance had additional loop overhead associated with software pipelining. In each case, the amount of unrolling for the loops scheduled without prefetching loads was greater than with

Data	Stat	Reuse	Prefetch	Percent
Cycles	Mean	1022985	876281	93%
	Median	121148	114805	97%
	StDev	2764083	2448140	
II	Mean	13.6	9.3	93%
	Median	7.0	7.0	100%
	StDev	21.0	5.8	
RecII	Mean	11.5	6.3	94%
	Median	5.0	5.0	100%
	StDev	21.4	3.2	
Int Reg	Mean	23.7	16.4	80%
	Median	18	12	77%
	StDev	18.2	11.2	
FP Reg	Mean	14.2	7.8	69%
	Median	11.0	6.0	53%
	StDev	13.3	7.1	

Table 1. Experimental Results

prefetching loads. This reduced the number of branch instructions executed and improved execution time. In addition, the schedules without prefetching loads were dense and hid memory latency well. In two of the loops that degraded with prefetching loads, the register pressure without prefetching loads was too high for an architecture with the standard 32 integer registers. Therefore, using prefetching loads was still beneficial.

Sánchez and González [19] give a method for dealing with the effects of loop overhead that would work in our case too. They evaluate multiple software pipeline schedules: one schedule that uses reuse analysis and one schedule that assumes that all references are cache misses, except where the miss harms the recurrence constraint and it is reasonable to assume a cache hit. Then they select the schedule with a lowest cost considering software pipelining overhead. This allows them to pick a schedule with low overhead when the reuse models do not yield substantially different performance.

In this experiment, there was an average of a 31% drop in floating-point register pressure with a median drop of 47%. There were 44 loops whose floating-point register pressure was lowered when prefetching loads were used, some by as much as 80%. Four loops had an increase in floating-point register pressure due to tighter schedules, although no increase caused a successful pipeline for a 32-register machine to fail. Thirty-one loops showed no difference in floating-point register pressure. Finally, there were 10 loops that were able to be pipelined with 32 floating-point registers using prefetching loads that were not able to be pipelined without prefetching loads.

For integer registers, there was an average of a 20% drop in register pressure with a median of 23%. Forty-two loops saw a drop in integer register pressure by as much as 68%, 33 stayed the same and 4 increased in register pressure. The 4 loops that increased did so due to tighter schedules. Finally, 23 loops were able to be scheduled within 32 integer registers when using prefetching loads that were not able to be scheduled when prefetching loads are not used. Considering both integer and floating-point register pressure 25 loops (31%) were able to pipelined using modern register pressure limits with prefetching loads that were not able to be pipelined within the limits without prefetching loads. Using prefetching loads caused no loop to be not pipelinable within modern architecture register constraints.

5 Conclusion

In this paper, we have shown how assuming self-spatial loads are either all cache hits or all cache misses is not adequate when software pipelining. Assuming all cache hits leads to a degradation in performance and assuming all cache misses leads to higher register pressure and unnecessarily long initiation intervals. To alleviate this problem, we have suggested the use of a hardware prefetching load instruction that loads two consecutive cache lines rather than one. This allows self-spatial loads to be scheduled as cache hits without a reduction in performance and without the register pressure and initiation interval problems of assuming cache misses.

Our experiment with prefetching loads showed an average of a 7% increase in performance over assuming all cache misses with a corresponding 7% reduction in average initiation interval. Just as importantly, we observed a 31% average reduction in floating-point register pressure and a 20% reduction in integer register pressure. This allowed 31% of the loops in our suite to be pipelined within modern machine register constraints with prefetching loads that were not able to be pipelined within those constraints without prefetching loads.

Given that memory latencies are increasing and that aggressive scheduling techniques such as software pipelining are necessary to get good performance on modern architectures, we need better methods to reduce the negative effects of long latencies on software pipelining. The use of prefetching loads proposed in this paper is a promising step in alleviating the effects of long latencies for scientific program loops.

Acknowledgments

This research was partially supported by NSF grant CCR-9409341.

References

[1] V. Adve, J.-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

[2] A. Aiken and A. Nicolau. Optimal loop parallelization. *SIGPLAN Notices*, 23(7):308–317, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[3] V. Allan, R. Jones, R. Lee, and S. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3), September 1995.

[4] V. Allan, M. Rajagopalan, and R. Lee. Software Pipelining: Petri Net Pacemaker. In *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, FL, January 20-22 1993.

[5] P. Briggs. The massively scalar compiler project. Technical report, Rice Univeristy, July 1994. Preliminary version available via anonymous ftp.

[6] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[7] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. Technical Report CRPC-TR95517-S, Center for Research on Parallel Computation, Rice Univeristy, November 1994. To appear in *Software – Practice and Experience*.

[8] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, California, 1991.

[9] S. Carr, C. Ding, and P. Sweany. Cache-sensitive software pipelining on the digital alpha 21164. To appear in *The Digital Technical Journal*.

[10] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software Practice and Experience*, 24(1):51–77, Jan. 1994.

[11] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, Massachusetts, 1992.

[12] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. Technical Report CRPC-TR95635-S, Center for Research on Parallel Computation, Rice Univeristy, October 1995.

[13] R. Crowell. An experimental evaluation of compiler-based cache management techniques. Master's thesis, Michigan Technological University, Mar. 1998.

[14] C. Ding, S. Carr, and P. Sweany. Modulo scheduling with cache reuse information. In *Proceedings of EuroPar '97*, Passau, Germany, August 1997.

[15] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, 1987.

[16] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[17] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–75, Boston, Massachusetts, 1992.

[18] B. Rau. Iterative modulo scheduling. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, San Jose, CA, December 1994.

[19] F. Sánchez and A. González. Cache-sensitive modulo scheduling. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park, NC, December 1997.

[20] P. H. Sweany and S. J. Beaty. Overview of the Rocket retargetable C compiler. Technical Report CS-94-01, Department of Computer Science, Michigan Technological University, Houghton, January 1994.

[21] N. Warter, G. Haab, and J. Bockhaus. Enhanced Modulo Scheduling for Loops with Conditional Branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 170–179, Portland, OR, December 1-4 1992.

[22] N. J. Warter, S. A. Mahlke, W. mei W. Hwu, and B. R. Rau. Reverse if-conversion. *SIGPLAN Notices*, 28(6):290–299, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[23] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.