Circular Scheduling: A new Technique to Perform Software Pipelining

Suneel Jain

MIPS Computer Systems, Inc.
928 Arques Avenue
Sunnyvale, CA 94086

email: suneel@mips.com

ABSTRACT

With the advent of deeply pipelined RISC processors, static instruction scheduling by the compiler has become extremely important to obtain high processor performance. This is especially true for floating point code, since in general floating point operations have longer latencies compared to integer operations. This paper suggests using a new algorithm called *circular scheduling*, to perform software pipelining. Software pipelining has previously been investigated mostly for VLIW architectures. The algorithm described in this paper is shown to be quite effective for a scalar architecture. Register renaming, an idea that originates from dynamic instruction scheduling, is used in conjunction with this algorithm to augment its performance. The techniques described here have been implemented as part of a commercial, production quality optimizing compiler for a RISC architecture. The resulting performance improvement has verified the feasibility and practicality of the techniques.

Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
Toronto, Ontario, Canada, June 26-28, 1991.

1. Introduction

Basic block scheduling [GrHe 83] [GiMu 86] has been used very successfully to schedule code for scalar RISC processors. This paper describes a new technique to improve the performance beyond this level by performing software pipelining.

Software Pipelining [HePa 90] is an optimization to reorganize loops such that each iteration of the software pipelined loop contains instructions from different iterations of the original loop. Instructions from a new loop iteration are initiated before the preceding iterations are complete.

During the normal execution of a loop, the processor resources typically are not fully utilized for the entire duration of the loop. For each iteration, there is a ramp up at the start and a ramp down at the end of the loop. This is caused by values being loaded at the start of the loop and stored back at the end of the loop. Software pipelining reduces these periods to once before the loop and once after the loop. These portions of the code represent the prolog and the epilog for the loop created after software pipelining.

In our implementation, software pipelining is accomplished by selectively moving instructions from the top of the loop body to the bottom of the loop body. The instructions can be thought of as having been moved out from the loop body and the same instructions from the next iteration

moved up into the loop body. The loop body can be visualized as a circular list of instructions, instead of a linear list. Instructions can be moved freely along this circular list. This leads to the name *Circular Scheduling* for this algorithm. Any node on the circle can be treated as the start of the loop body and the previous node its end. The modified block of instructions is scheduled using the existing basic-block scheduler. This process of moving instructions is repeated based on certain heuristics, till the best possible schedule is obtained.

In this implementation, circular scheduling is done after register allocation, as part of the normal scheduling process. Register allocation introduces dependencies in the dag that did not exist in the code before, resulting in unnecessary constraints for software pipelining. An algorithm to remove these dependencies is also described in this paper. The algorithm performs register renaming during the process of instruction scheduling. This allows the scheduler to alter decisions made by the register allocator and thus effectively utilize the limited number of floating point registers in the architecture.

2. Related Work

Software pipelining for scalar architectures has not been studied extensively. S.Weiss and J.E. Smith [WeSm 87] compare loop-unrolling and software pipelining as two alternative techniques to improve scalar code for the Cray-1S. Their approach to software pipelining is similar to this implementation. However, their algorithm is too limited. They allow only one class of instructions, either loads, execution instructions, or stores to be moved out of the loop. They also assume the existence of 256 registers and thus do not handle problems caused by limited number of registers. The work described here does software pipelining in a much more general way and allows any instruction to be moved out of the loop based on heuristics. It also works with the 16 floating-point registers available in the MIPS architecture. Better results are also obtained by combining loop unrolling and software pipelining [HePa 90].

Other work in software pipelining ([BoCh 90], [Char 81], [DHB 89], [Lam 88], [RaGl 81], [SDX 86], [Touz 84]) has been done mostly for VLIW architectures. The algorithms used in most cases are very complex and not suitable for doing software pipelining for scalar architectures. In a VLIW architecture, multiple instructions can be issued in each cycle. Touzeau [Touz 84] and Lam [Lam 88] find the minimum initiation interval for which a schedule can be found. These methods are similar to solving a packing problem for the given instructions. The scheduling constraints are divided into resource constraints and dependence constraints, and are specially handled to perform software pipelining. Touzeau and Lam also assume that the target machine has a large number of registers. Software pipelining is done before register allocation. Touzeau handles running out of registers by introducing spill code which makes the schedule strung out and not optimal. Lam uses simpler techniques for scheduling such loops, which serializes the execution of loop iterations.

The circular scheduling algorithm described here basically moves instructions around the loop iterations. It makes use of the existing basic-block scheduler to do the actual scheduling. All the constraints, including dependence constraints between instructions of different iterations are handled by edges in the dag. In contrast to the algorithms for the VLIW architectures, our algorithm does not find *the* optimal schedule. It attempts to improve the scheduling using iterative steps. This makes it adaptable to more complicated scheduling rules.

In our implementation, software pipelining is done after register allocation. Register renaming is performed to reduce dependencies introduced by this. Earlier work [Ferr 87], [DHB 89] has also indicated that potential parallelism in a code segment is generally increased if renaming is used to eliminate multiple definitions of a variable. At the end of our software pipelining algorithm, all register references are fully resolved. There is no need for the scheduler to introduce spill code. In the event of running out of regis-

ters, the scheduler compromises by doing less aggressive code movement, thus cutting down on the number of registers used.

3. Background

The scheduling techniques described in this paper have been developed as part of the common backend for all the MIPS optimizing compilers [CHKW 86]. The instruction scheduler for the MIPS compilers is implemented at the assembler level. The scheduler is invoked after register allocation has been done. An important objective was to leverage the work performed by the existing scheduler. The existing scheduler already computes global dataflow information for the general and floating-point registers. This information can also be used for register renaming and software pipelining.

The MIPS architecture [Kane 87] provides 16 floating-point registers (FPR). The FPRs can hold either single precision or double precision values. The 16 registers become a constraining factor when performing aggressive optimizations like loop-unrolling and software pipelining and need to be factored into the algorithms.

There are several implementations of the MIPS floating point architecture. The R3010 and the R6010 are the current implementations in CMOS and ECL respectively.

Operation	Latency in clock cycles			
	R3010		R6010	
	SP	DP	SP	DP
LOAD	2	n.a.	2-3	2-3
STORE	1	n.a.	1-2	2
ADD	2	2	3	4
SUB	2	2	3	4
MULT	4	5	5	6
DIV	12	19	17	26

Table 1: Latency of floating-point operations

The latencies for the most common floating-point operations, both single precision and double precision, are given above in Table 1.

Both the floating-point chips permit loads and stores to execute concurrently with floating point operations, provided they do not modify or use the result registers of the executing operation. Fixed-point operations may also execute concurrently with floating-point operations. The floating-point operations can be overlapped with each other in certain circumstances. The exact rules are quite complex. The conditions for overlap differ for the two floating-point chips.

4. Circular Scheduling

In this section, we give the motivation for circular scheduling and the intuitive reasoning behind it. We then give a more detailed description of our software pipelining algorithm.

4.1. Motivation

A loop body is a block of code that gets repeatedly executed. The first instruction in the loop executes after the last instruction in the loop body for all iterations except the first one. The first instruction can thus be viewed as a successor to the last instruction. This leads to the idea of treating the instructions in a loop body as a circular list of instructions, rather than as a linear list.

Instructions in a loop body that are the roots of its dag are logically the instructions at the top of the loop. These instructions can be moved from the top of the loop to the bottom of the loop. An instruction moved in this manner has circled once around the loop, and we call it a *circled* instruction. Circled instructions in a loop body correspond to the next iteration. A copy of each circled instruction constitutes the prolog for the software pipelined loop. The remaining instructions are copied to form the loop epilog. If the original loop had N iterations, the software pipelined loop is executed N - 1 times and the prolog and epilog combined form the remaining iteration.

If a circled instruction is also a root in the dag for the loop, it can be moved again. Such an instruction is said to have been circled twice. It becomes an instruction for the next to next iteration (two iterations after the current iteration of

the loop). This process of circling can be continued to cause as many different iterations executing at the same time in the loop body as needed. For a loop that has been circled **k** times, there are **k** prologs and **k** epilogs. The loop body is executed **N - k** times. Having too many iterations executing simultaneously causes code expansion in the prolog and epilog and thus should be done only when justified by the resulting performance gains.

How does moving an instruction from the top to the bottom of the loop help? It helps if the scheduler is able to generate a better schedule for the modified block of code. The basic premise of software pipelining is that this is indeed the case. The benefits are twofold:

1. Instructions close to the top of the loop (the successors of the roots of the dag) now become roots of the dag of the modified loop. They can thus be scheduled earlier, eliminating stalls at the start of the loop body.
2. The moved instruction can be scheduled before some of the instructions at the end of the loop body if there are no dependencies between them. This effectively utilizes stalled cycles that are usually present towards the end of the loop.

As more instructions are moved, the number of stalls will decrease to a minimum level. At a certain point, moving more instructions will cause the schedule to get worse. After all, if all instructions in the loop are moved, we get back the original loop body and thus the original schedule.

Keeping track of inter-iteration dependencies is fairly simple with this method of doing software pipelining. The iteration number of each instruction is the number of times it has been circled. Whether an edge needs to be inserted in the dag for any pair of instructions from different iterations can be determined by analyzing the code. Sometimes data dependency information is needed, and the algorithm relies on such information being provided by an earlier pass of the compiler.

4.2. Example

The concepts described in the previous section are best illustrated with a simple example. The example is a simple loop to add a constant to all the elements of an array. The source code is given below:

```
for (i = 0; i < N; i = i + 1)
    x[i] = x[i] + c;
```

The object code generated with the normal scheduling algorithm, for the R6000/R6010, is shown below in Figure 1. There is a 2 cycle interlock caused by the 4 cycle latency of a double precision add for the R6010. Thus each iteration of the loop takes 7 cycles.

```
LOOP:
        ldc1    $f4,0(r3)
        addiu   r3,r3,8
        add.d   $f6,$f4,$f12
        bne     r3,r2,LOOP
        < 2 cycle interlock >
        sdc1    $f6,-8(r3)
```

Figure 1: Object code with normal scheduling

The instructions immediately following the branch instructions (beq, bne) are the delay slot of the branch and logically execute before the branch. The instruction scheduler can do the following change:

```
addiu  t0,t0,i      =>      lw     t1,i+ j(t0)
lw     t1,j(t0)             addiu  t0,t0,i
```

This allows loads and stores to be freely moved above or below the increment of the base register. Without this capability, the benefits of software pipelining would be very limited.

The object code for the same example, when compiled with software pipelining is shown in Figure 2. Two instructions, *addiu* and *ldc1* are moved from the top to the bottom of the loop (i.e *circled* once). They are also copied into the prolog. The loop body now executes N - 1 times. There are no interlocks left in the loop body. Each iteration thus takes 5 cycles.

The epilog of the software pipelined loop contains those instructions that have not been

copied over to the prolog. There is a 3 cycle interlock which takes effect only once per loop, in the last iteration.

```
LOOP:
        addiu   r3,r3,8
        beq     r3,r2,LEND          prolog
        ldc1    $f18,-8(r3)
LBEG:
        add.d   $f16,$f18,$f12
        ldc1    $f18,0(r3)
        addiu   r3,r3,8             loop body
        bne     r3,r2,LBEG
        sdc1    $f16,-16(r3)
LEND:
        add.d   $f16,$f18,$f12
        < 3 cycle interlock >       epilog
        sdc1    $f16,-8(r3)
```

Figure 2: Object code after software pipelining

Loop unrolling was disabled while creating this example. This was done to get short, concise object code. In normal operation, the loop will be unrolled in most cases before it is software pipelined.

4.3. Algorithm

The software pipelining algorithm described below relies on the existing basic-block scheduler to do most of the work. Loop unrolling is done by the global optimizer in an earlier phase of the compiler. This ensures that the scheduler has enough instructions to rearrange during software pipelining. The scheduler does not attempt to perform loop unrolling. The current version of the algorithm only moves instructions once across the loop. This is adequate because the input is unrolled loops. Software pipelining is currently done only for single basic block loops with no procedure calls. Only loops with a loop index that is incremented once inside the loop are considered. This allows construction of a prolog such that the software pipelined loop body is executed exactly N - 1 times.

A key issue in the algorithm is how to select instructions to move out of the loop. Only instructions that can be scheduled first are eligible to be moved out. This is the same as the list of candidate instructions available for scheduling (the roots of the dag). Selecting an instruction out of this list is based on the following heuristics:

a) Move instructions that are on the longest path in the dag, since they are most likely to cause stalls towards the end of the loop.

b) Identify resources that cause bottlenecks at some places in the loop. Move instructions that use those resources. For example, if most of the stalls are due to instructions waiting for the floating-point multiply unit to be free, give higher priority to moving multiply instructions.

The algorithm to perform circular scheduling is summarized below:

1. Apply the basic-block scheduling algorithm and check if there are any stalls. If there are no stalls, use the schedule.
2. If there are any stalls, check if the basic block is a loop that is software pipelinable. The constraints to be satisfied are listed earlier in this section.
3. Choose some instructions that are suitable candidates to move from the top of the loop. Move the instructions to the bottom of the loop.
4. Rebuild the dag for the modified loop and apply the scheduling algorithm as before. Count the number of stalls in this schedule.
5. If there are no stalls, use the current schedule. If there are still some stalls, repeat steps 3 and 4.
6. Terminate this process if there are no more instructions that can be moved or the generated schedule is a *lot worse* than earlier schedules. The determination of *lot worse* is done using some heuristics. If one of the earlier schedules was better than the current one, it is selected for use.
7. Create the prolog and epilog basic blocks and schedule them. Alter the main loop body so that it is executed for one less itera-

tion than before. The prolog and epilog combined, execute one full iteration.

4.4. Extensions

The current algorithm does not use data dependency information that is being computed by an earlier phase of the compiler. The use of this information is being implemented and will reduce the number of inter-iteration dependencies.

The software pipelining algorithm described above can easily be extended to handle loops with multiple basic blocks also. As instructions are moved out of the loop, all the basic blocks in the loop will need to be rescheduled.

The algorithm can be extended to allow more than 2 iterations of the loop to be executed within the software pipelined loop. This is done by moving instructions that are already circled. This will lead to the creation of 2 prologs and 2 epilogs. Since the input to the scheduler is unrolled loops, it has not been necessary to implement this. In the presence of operations with extremely long latencies, this would be more important.

The algorithm described above can be used with a VLIW architecture also. In that case it will be necessary to move instructions from more than one iteration out of the basic block. It would be interesting to compare the performance of our algorithm with other algorithms previously implemented.

5. Register Renaming

The success of the circular scheduling algorithm described above is tied to minimizing the dependencies in the loop. This allows greater freedom in moving instructions around. Since the scheduling is done after register allocation, some dependencies are introduced that did not exist in the code before. To minimize these dependencies, register renaming is done in the scheduler.

The term register renaming comes from *dynamic scheduling* [Toma 67], whereby the hardware rearranges the instruction execution to reduce stalls. To reduce data dependences caused by the definition and use of the same register, the hardware can logically rename the register for a subsequent definition. This allows the instructions defining the same register to proceed concurrently. Figure 3. illustrates how renaming the registers eliminates some of the edges in the dag, thus reducing dependencies.

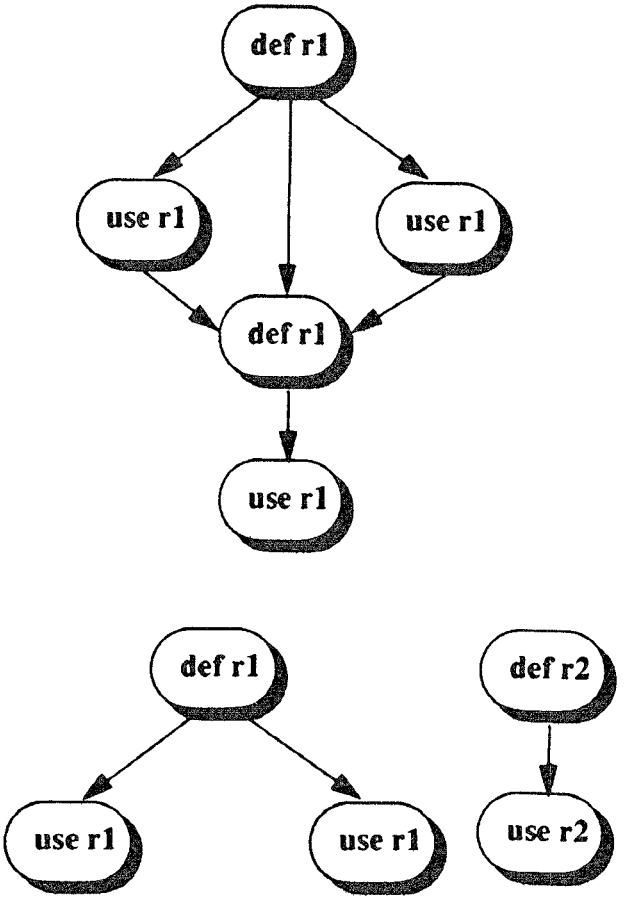

Figure 3: Effect of register renaming on the dag

Other implementations have done software pipelining before register allocation. They avoid the problem of dependencies introduced by register allocation, but introduce the problem of not being able to allocate registers fully. This leads to spilling of some registers which in turn can adversely affect the schedule. For architectures with few registers, this has a big impact. Register renaming allows us to regain most of the benefits of doing instruction scheduling before register

allocation. It is performed at the same time as instruction scheduling.

5.1. Algorithm

The algorithm to do register renaming is described below:

1. Do global dataflow analysis to compute the set of registers live at the end of each basic block.
2. Identify registers that are not live at the beginning and at the end of the basic block. This includes registers that are used as temporaries within the basic block as well as registers that are unused. This forms the pool of registers available for register renaming.
3. Identify the live ranges for the temporary registers within the basic block. This is done while building the dag for the basic block.
4. While building the dag, ignore the dependencies between different live ranges. In other words, do not add edges from the uses of a live range to the definition of the next live range. Also remove the edge from the definition of a live range to the definition of the next live range.
5. Pick an instruction to schedule based on certain heuristics.
6. If the instruction uses a temporary register, that register is replaced by the new register allocated for its live range. If the use is the last use in the live range, the new register is put back in the pool of available registers.
7. If the instruction being scheduled defines a temporary register, a new register is chosen for it from the pool of available registers. This new register is now assigned for the live range containing the definition. If there are no more registers available for renaming, the scheduling algorithm is aborted. We use an earlier schedule that was the best so far.
8. Repeat steps 5, 6 and 7 till all instructions in the basic block are scheduled.

To reduce the likelihood of running out of registers to do renaming, register usage is used as one of the heuristics while picking the next instruction to schedule. Given two candidate instructions for scheduling next, the one that does not need a new register or frees up a register is given higher priority.

The register renaming algorithm has been found to be very useful even for basic blocks that are not loops to be software pipelined. It is thus applied whenever there are stalls in a basic block, even if the block is not software pipelinable.

6. Experimental Results and Analysis

The performance improvements obtained using the software pipelining algorithm was evaluated using several benchmarks. The benchmarks were run with full optimization, with and without software pipelining and register renaming. The tables below list the percentage improvement in each case. The tests were run on two systems with different floating point implementations. The two systems are the M/2000 (R3000,R3010) and the RC6280 (R6000,R6010).

6.1. Livermore Loops

The Livermore Loops [MCMA 72] benchmark measures the performance of 24 FORTRAN kernels. These kernels are excerpts from large FORTRAN programs that have been judged to provide a good measure of scientific application performance. The MFLOP/s rate for each kernel was measured with and without software pipelining. The MFLOP/s rates were obtained by averaging the data for ten runs. The table gives the percentage improvement in the MFLOP/s rate for each kernel, for both the R3010 and the R6010 based systems.

Some of the kernels showed no improvement because they were already being optimally scheduled (i.e. no interlocks or stalls). Kernels 15, 16 and 17 have conditional statements and were not software pipelined. Kernel 22 calls the

EXP library routine and did not improve. For a few kernels, there is a slight degradation in performance, even though cycle counts are either the same or better. This is caused by scheduling of several loads together. This exposes some data cache misses that were hidden behind some floating point operations earlier. This regression is being investigated further. The performance for the Livermore loops is presented below in Table 2.

Kernel	Percentage Improvement	
	R3010	R6010
1	30	38
2	4	0
3	3	2
4	0	17
5	-1	-2
6	-2	-5
7	36	34
8	17	15
9	43	53
10	0	0
11	0	6
12	0	23
13	0	-2
14	9	13
15	1	0
16	1	0
17	0	0
18	30	17
19	2	-5
20	3	0
21	2	42
22	0	0
23	33	19
24	1	2

Table 2: Performance of Livermore Loops (DP)

6.2. Other Benchmarks

Performance was analyzed for some other benchmarks also. For these benchmarks, cycle times were measured instead of the actual run times. The cycle times were obtained using the MIPS instruction tracing facility **pixie**. This tool measures exactly the number of machine cycles needed to execute a given program. This measures performance independent of other hardware parameters like cache misses and page faults. Performance improvement for some common benchmarks is summarized in Table 3. below:

Benchmark	% Improvement	
	R3010	R6010
linpack	0	23
la400	8.6	28
tomcatv (SPEC)	17.3	18.2
nasa7 (SPEC)	5.4	12.5
doduc (SPEC)	3.1	2.2
fpppp (SPEC)	2.4	2.3

Table 3: Performance of common benchmarks

For the linpack benchmark, the existing scheduler already generates an optimal schedule for the R3010. The improvements are due to both software pipelining and register renaming. The compile time for the above benchmarks increased between 5% and 25%, for the programs analyzed. The increase in compile time is generally proportional to the amount of benefit derived. For programs with fewer floating-point code and loops, the degradation in compile time is much less. Of course, the performance improvement is also small for those programs.

For all the benchmarks, loop unrolling is performed before software pipelining. Since unrolling removes a large number of the FP interlocks, the gains from software pipelining are not as large as they would have been if loop unrolling was not done. However, the results of applying both optimizations are better than applying either of them alone.

7. Conclusion

The algorithms presented in the paper allow scalar processors to take advantage of software pipelining. The algorithms provide fairly simple and highly feasible ways to extend the capabilities of a basic-block scheduler. The optimizations performed yield reasonable improvement over already optimized code from a production quality

compiler, without adding substantially to the compilation time. The benefit is larger in most benchmarks for the R6010 as compared to the R3010. This is because of the longer latencies in the R6010 that provide better opportunities for the software pipelining algorithm. Better results are expected with future implementations of the architecture that offer either deeper pipelining or more instruction level parallelism.

8. Acknowledgements

The author gratefully acknowledges significant contributions by Sun Chan, Fred Chow, Earl Killian, Sin Lew and Alex Wu in developing the ideas described in the paper. Fred also provided me with invaluable help in writing and editing the paper.

9. References

[BoCh 90] F. Bodin, F. Charot, "Loop optimization for Horizontal Microcoded Machines", Proc. of International Conference on Supercomputing, June 1990.

[Char 81] A.E. Charlesworth, "An Approach to Scientific Array Processing: The Architecture Design of the AP-120B/FPS-164 Family", Computer, December 1981.

[CHKW 86] F. Chow, M. Himelstein, E. Killian, L. Weber, "Engineering a RISC Compiler System", Proceedings COMPCON, IEEE, March 1986.

[DHB 89] J.C. Dehnert, P.Y.-T. Hsu, J.P. Bratt, "Overlapped Loop Support in the Cydra 5", Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989.

[GiMu 86] P. B. Gibbons, S.S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture", Proc. of the SIGPLAN Symposium on Compiler Construction, June 1986.

[GrHe 83] T.R. Gross, J.L. Hennessy, "Postpass Code Optimization of Pipeline Constraints", ACM Transactions on Programming Languages and Systems, July 1983.

[Ferr 87] J. Ferrante, "What's in a Name, Or the Value of Renaming for Parallelism Detection and Storage Allocation", Technical Report #12157, IBM Thomas J. Watson Research Center, January 1987.

[HePa 90] J.L. Hennessy, D.A. Patterson, "Computer Architecture, A Quantitative Approach", Morgan Kaufmann Publishers, Inc. (1990)

[Kane 87] Gerry Kane, "MIPS RISC Architecture", Prentice-Hall, Inc. (1987)

[Lam 88] M. Lam, "Software Pipelining: An effective scheduling technique for VLIW machines", Proc. of the ACM SIGPLAN Conf. on Programming Languages Design and Implementation, June 1988.

[MCMA 72] F. H. McMahon, "FORTRAN CPU Performance Analysis", Lawrence Livermore Laboratories, 1972.

[RaGl 81] B.R. Rau, C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High-performance Scientific Computing", MICRO-14, October 1981.

[SDX 86] B. Su, S. Ding, J. Xia, "URPR - An Extension of URCR for Software Pipelining", Proc. 19th Annual Workshop of Microprogramming, October 1986.

[Toma 67] R.M. Tomasulo, "An Efficient Algorithm for Exploring Multiple Arithmetic Units", IBM Journal of Research and Development, January 1967.

[Touz 84] R. F. Touzeau, "A FORTRAN Compiler for the FPS-164 Scientific Computer", Proc. of the ACM SIGPLAN Symposium on Compiler Construction, June 1984.

[WeSm 87] S. Weiss, J.E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers", Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems, October 1987.

A patent application has been filed by MIPS Computer Systems, Inc. on some of the algorithms described in this paper.