In the Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, Philadelphia, Pennsylvania, May 22-24, 1996, pp. 1-11. Copyright © 1996 ACM (see notice below).

Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler

John Ruttenberg*, G.R.Gao†, A.Stoutchinin†, and W.Lichtenstein*

*Silicon Graphics Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043
†McGill University – School of Computer Science, 3480 University St., McConnell Building, Room 318, Montreal, Canada H3A2A7
Authors' email: rutt,wdl@sgi.com gao,stoa@cs.mcgill.ca

Abstract

This paper is a scientific comparison of two code generation techniques with identical goals — generation of the best possible software pipelined code for computers with instruction level parallelism. Both are variants of *modulo scheduling*, a framework for generation of software pipelines pioneered by Rau and Glaser [RaGl81], but are otherwise quite dissimilar.

One technique was developed at Silicon Graphics and is used in the MIPSpro compiler. This is the production compiler for SGI's systems which are based on the MIPS R8000 processor [Hsu94]. It is essentially a branch–and–bound enumeration of possible schedules with extensive pruning. This method is heuristic because of the way it prunes and also because of the interaction between register allocation and scheduling.

The second technique aims to produce optimal results by formulating the scheduling and register allocation problem as an integrated integer linear programming (*ILP*[1]) problem. This idea has received much recent exposure in the literature [AlGoGa95, Feautrier94, GoAlGa94a, GoAlGa94b, Eichenberger95], but to our knowledge all previous implementations have been too preliminary for detailed measurement and evaluation. In particular, we believe this to be the first published measurement of runtime performance for ILP based generation of software pipelines.

A particularly valuable result of this study was evaluation of the heuristic pipelining technology in the SGI compiler. One of the motivations behind the McGill research was the hope that optimal software pipelining, while not in itself practical for use in production compilers, would be useful for their evaluation and validation. Our comparison has indeed provided a quantitative validation of the SGI compiler's pipeliner, leading us to increased confidence in both techniques.

1. It is unfortunate that both *instruction Level Parallelism* and *Integer Linear Programming* are abbreviated *ILP* in the literature. To clarify, we always use the abbreviation *ILP* to mean the latter and will suffer through describing instruction level parallelism by its full name.

Little work has been done to evaluate and compare alternative algorithms and heuristics for modulo scheduling from the viewpoints of schedule quality as well as computational complexity. This, along with a vague and unfounded perception that modulo scheduling is computationally expensive as well as difficult to implement, have inhibited its incorporation into product compilers.
— B. Ramakrishna Rau [Rau94]

1.0 Introduction

1.1 Software pipelining

Software pipelining is a coding technique that overlaps operations from various loop iterations in order to exploit instruction level parallelism. In order to be effective software pipelining code must take account of certain constraints of the target processor, namely:

1. instruction latencies,
2. resource availability, and
3. register restrictions.

Finding code sequences that satisfy the constraints (1) and (2) is called *scheduling.* Finding an assignment of registers to program symbols and temporaries is called *register allocation.* The primary measure of quality of software pipelined code is called the *II*, short for *iteration* or *initiation interval*. In order to generate the best possible software pipelined code for a loop, we must find a schedule with the minimum possible II and also we must have a register allocation for the values used in the loop that is valid for that schedule. Such a schedule is called *rate–optimal*. The problem of finding rate–optimal schedules is NP–complete [GaJo79], a fact that has led to a number of heuristic techniques [DeTo93, GaSc91, Huff93, MoEb92, Rau94, Warter92, Lam88, AiNi88] for generation of optimal or near optimal software pipelined code. [RaFi93] contains an introductory survey of these methods.

1.2 The allure of optimal techniques

Recently, motivated by the critical role of software pipelining in high performance computing, researchers have become interested in non–heuristic methods, ones that guarantee the optimality of the solutions they find. The key observation is that the problem can be formulated using integer linear programming (*ILP*), a well known framework for solving NP–complete problems [Altman95, AlKePoWa83]. Given such a formulation, the problem can be given to one of a number of standard ILP solving packages. Because this framework has been effective in solving other computationally difficult problems [NeWo88, Nemhauser94, Pugh91,BiKeKr94], it is hoped that it can also be useful for software pipelining.

1.3 The showdown

The rest of this paper presents a detailed comparison of two software pipelining implementations, one heuristic and one optimal. The heuristic approach is represented by the Silicon Graphics *MIPSpro* compiler. The optimal approach is represented by *MOST*, an ILP based pipeliner developed at McGill University. We adapted the MOST scheduler to the MIPSpro compiler so that it could act as an exact functional replacement for the original heuristic scheduler. This work allowed us to perform a very fair and detailed comparison of the two approaches.

What did we discover? In short, the SGI heuristics were validated by this study. Comparison with MOST was an effective way to evaluate SGI's production quality pipeliner. In particular, we discovered that the optimal approach is only very rarely able to find a better II than the heuristic approach and benchmarks compiled with the optimal techniques did not clearly perform better than those compiled with the heuristic pipeliner.

In the remainder of this paper we describe the two approaches in detail, present our experimental results, draw some conclusions from these results, and try to shed some intuitive light on the results.

2.0 The heuristic approach — software pipelining at Silicon Graphics

2.1 MIPSpro compiler

Starting in 1990, Silicon Graphics designed a new microprocessor aimed at the supercomputing market called *R8000* [HSU94] and shipped with SGI's *Power Challenge* and *Power Indigo2* computers. It is an in–order 4-issue superscalar RISC processor featuring fully pipelined floating point and memory operations. As part of this project, a new compiler was implemented that had the basic goal of generating very high quality software pipelined inner loop code. This compiler shipped under the name *MIPSpro Compiler* with R8000 based systems starting in late summer of 1994. At the time the system shipped, it had the best reported performance on a number of important benchmarks, in particular for the floating point SPEC92. Our measurements show the compiler's software pipelining capabilities play the central role in delivering this performance on the R8000. (See Figure 2.)

The MIPSpro compiler performs a rich set of analysis and optimizations before its software pipelining phase. These fall into three distinct categories:

1. high level loop analysis and transformations, including:
    a. array dependence analysis,
    b. loop interchange, and
    c. outer loop unrolling;
2. classical intermediate code optimizations, such as:
    a. common subexpression elimination,
    b. copy propagation,
    c. constant folding, and
    d. strength reduction;
3. special inner loop optimizations and analysis in preparation for software pipelining, in particular:
    a. if-conversion to convert loops with internal branches to a form using conditional moves [AlKePoWa83, DeTo93],
    b. interleaving of register recurrences such as summation or dot products,
    c. inter iteration common memory reference elimination, and
    d. data dependence graph construction.

2.2 Modulo scheduling

The framework for the MIPSpro compiler's pipeliner is *modulo scheduling,* a technique pioneered by Bob Rau and others and very well described in the literature. [Lam88, RaFi93, Lam89]. Modulo schedulers search for possible schedules by first fixing a iteration interval (II) and then trying to pack the operation in the loop body into the given number of cycles. If this is unsuccessful for one of a number of reasons, a larger II may be tried.

The SGI scheduler contains a number of extensions and elaborations. Five of the most important are:

1. binary instead of linear search of the IIs,
2. branch–and–bound ordering of the search for a schedule with a great deal of heuristic pruning,
3. use of multiple heuristic orderings in (2) to facilitate register allocation,
4. generation of spill code when required to alleviate register pressure, and
5. pairing of memory references to optimize memory system utilization.

As we describe SGI's pipeliner in the following sections, it will be useful to keep these in mind.

2.3 Modulo scheduling using binary search

The SGI modulo scheduler searches the space of possible IIs using a binary instead of linear search.[1] This has no measurable impact on output code quality, but can have a dramatic impact on compile speed. The search is bounded from below by MinII, a loose lower bound based on resources required and any dependence cycles in the loop body [RaGl81]. The search is bounded from above by an arbitrary maximum —MaxII = 2MinII. We set this maximum as a sort of compile speed circuit breaker under the assumption that software pipelining has little advantage over traditional scheduling once this bound is exceeded. In practice, this has proven to be a reasonably accurate assumption. Our search also makes the heuristic assumption that if we can find a schedule at II we will also be able to find one at II+1. Although it is possible to find counter examples of this in theory, we have yet to encounter one in the course of fairly extensive testing.

The search is designed to favor the situation where we can schedule at or close to MinII, as these cases are overwhelmingly common. A simple binary search would be wasteful in cases where a schedule could be found within a few cycles of the MinII. Instead, we use a search with two distinct phases:

1. *Exponential backoff* — Initially, the search attempts to establish an upper bound at which a schedule exists. During this phase we perform an exponential backoff from MinII,

1. The use of binary search in this context has a fairly long history in the literature. Touzeau described the AP120 and FPS164 compiler and explains how binary search is used [Tou84]. Lam pointed out that being able to find a schedule at II does not imply being able to find a schedule at II+1 and used this to explain why her compiler used linear search [Lam88].

searching successively:

MinII, MinII+1, MinII+2, MinII+4, MinII+8,...

until we either find a schedule or exceed MaxII. If a schedule is found with II ≤ MinII+2, there are no better IIs left to search and the schedule is accepted. If no schedule is found, our modulo scheduling attempt has failed. (But we may still be able to find a software pipelined schedule by introducing spills and trying again. Section 2.8.)

2. *Binary search* — If phase 1 is successful and a schedule is found, but with II > MinII+2, a binary search is used over the space of feasible IIs. Information from phase 1 allows us to tighten both the upper and lower bounds for the search. For the lower bound, we use the largest II for which phase 1 tried to schedule and failed. For the upper bound, we use the II for which phase 1 succeeded in finding a schedule. These bounds are tighter than MinII and MaxII which are used to bound phase 1.

In special situations the two phase approach is abandoned in favor of simple binary search. This is done when there is reason to be pessimistic about being able to software pipeline. In particular, simple binary search is used exclusively after spills are introduced into the code. (See Section 2.8.)

2.4 Enumerating possible schedules with pruning

The process of searching for a schedule at an given II is viewed as an enumeration of the possible schedules for that II. This enumeration can be accomplished with a branch–and–bound algorithm whose outlines should be familiar. The algorithm is presented in its fully exponential form in Figure 1.

This is an exponential algorithm and is not practical in its unpruned form. In order to make it useful, it is necessary to *prune* away much of the search. Fortunately, this can be done without losing too much, since much of the backtracking it does is extremely unlikely to accomplish anything.

For example, consider two operations with unrelated resource requirements which are unrelated by data precedence constraints. Suppose one of these has been scheduled and the second one fails to schedule. What good can possibly come of allowing a backtrack from one to the other? They have nothing to do with one another and moving one of them in the schedule cannot make it possible to schedule the other.

Or consider the example of two fully pipelined operations with identical resource requirements and unrelated data precedence. Suppose the second (on the priority list) of these fails to schedule. Can it help to move the first one? Any other place we put it is a place that second operation would have found if still available.

The assumption that two operations are unrelated by data precedence is less important than might appear at first. Because we are modulo scheduling, any two operations that are not in the same strongly connected component can occupy any two cycles in the schedule; it's just a matter of adjusting the pipestages. (See Section 2.5 below.)

Step (4) in Figure 1 guides the backtracking of the algorithm. We will say that it chooses a *catch point* for the backtracking. The catch point is a scheduled operation that will advance to the next cycle of its legal range after all the operations following it on the priority list have been unscheduled. Pruning is accomplished by strengthening the requirements about which operations can be catch points.

1. Make a list of the operations to be scheduled, $L_o..L_{n-1}$.
2. After $L_0..L_{i-1}$ have been scheduled, attempt to schedule L_i as follows:
   a. Calculate a *legal range* of cycles in which L_i may be scheduled. If L_i is not part of a nontrivial strongly connected component in the data precedence graph, its legal range is calculated by considering any of its direct predecessors and successors that have already been scheduled. L_i must be scheduled late enough so that precedence arcs from its predecessors are not violated and early enough so any arcs to its successors are not violated. For operations that are part of nontrivial strongly connected components, we need to consider all scheduled members of L_i's strongly connected component. A longest path table is kept and used to determine the number of cycles by which two members must precede or follow each other. The legal range is cut off to be no more than II cycles, since searching more than this number of cycles will just reconsider the same schedule slots pointlessly.
   b. Consider the cycles in L_i's legal range in order. If a cycle is found in which L_i may be scheduled without violating resource constraints, schedule L_i in that schedule. If no such cycle is found, we have failed in this attempt to schedule L_i.
3. If (2) is successful, resume scheduling the next element of L in step (2) or if all the operations are scheduled, terminate with success.
4. If (2) is unsuccessful, find the largest $j < i$, such that L_j's legal range is not exhausted. Unschedule all the operations $L_j..L_{i-1}$. If there is no such j, the entire scheduling attempt has failed. Otherwise, set i = j and resume scheduling at step (2) with the next cycle of L_i's legal range.

FIGURE 1. Branch–and–bound enumeration of all possible schedules at a given II

The SGI pipeliner prunes by imposing the following constraints on backtracking:

1. Only the first listed element of a strongly connected component can catch.
2. Operation j may catch the backtrack caused by the scheduling failure of operation i if the resource required of i and j are non-identical and unscheduling j makes it possible to schedule i.
3. If no catch point is found under constraint (2) a slightly looser constraint is used. Under this constraint, Operation j may catch the backtrack of operation i even if i and j have identical resources. Unschedule all the operations starting with j on the priority list. If i can now be scheduled *but in a different schedule slot than j*, j may catch the backtrack.

2.5 Adjusting the pipestages

Our branch–and–bound algorithm does not require that the priority list be a strict topological ordering of the data dependence graph. The legal range calculation in Figure 1 only takes account of such predecessors and successors of an operation as are actually scheduled already, i.e., ahead of the current operation on the priority list. Why doesn't this result in "schedules" that violate data precedence constraints?

The answer is that it does. A simple postpass is used to compensate, ensuring validity. The postpass performs a single depth first search of the strongly connected component tree starting with the roots (operations with no successors, such as stores) and proceeding to predecessors. This visitation *is* topological. When each strongly connected component is visited, all its successors have already been visited, and their times in the schedule made legal in terms of *their* successors. Now it may be required to move the component to an earlier time in the schedule in order to make its results available on time to its predecessors. This can be done without changing the resource requirements of the schedule as a whole by moving the component by multiples of II. Once this has been done for every strongly connected component, we have a schedule that is valid both in terms of its resource and data precedence constraints.

Of course, this process may have the negative effect of lengthening live ranges. It is better from a register allocation perspective to use a topological ordering for the schedule priority list. But sometimes that is untenable in terms of actually finding a schedule. Fortunately the scheme of using multiple priority heuristics ensures that both topological and non-topological approaches can be applied to every piece of code, in search of what works best. (See Section 2.7.)

2.6 Register allocation and modulo renaming

Once a legal schedule is found, an attempt is made to allocate registers for it using fairly well known techniques. The R8000 has no explicit support for software pipelining. In particular, it has only conventionally accessed registers. Lam describes a technique called *modulo renaming* that allows effective pipelining with conventional architectures by replicating the software pipeline and rewriting the register references in each replicated instance [Lam89]. The SGI pipeliner borrows this technique. The modulo renamed live ranges so generated serve as input of a standard global register allocator that uses the *Chaitin-Brigg*s algorithm with minor modifications. [BrCoKeTo89, Briggs92].

2.7 Multiple scheduling priorities and their effect on register allocation

Early on the SGI team discovered that the ordering of operations on the schedule priority list has a significant impact on whether the schedules found can be register allocated. On reflection this is not very surprising. For example, a priority list that is a backward topological sort of the data dependence graph has the advantage that each operation is considered for scheduling only after any of its uses have already been placed in the schedule. When this is the case, we know the latest point in the schedule at which it is valid to place the operation and still have its result available in time for its uses. We can shorten live ranges by trying to place each operation as low in the schedule as possible, given this limitation.

A more interesting discovery was that different search orders seem to work well with different loop bodies. No single search order works best with all loop bodies. Backward topological search order, as described in the previous paragraph, works well in many cases since it tends to group the operands of each operation close together shortening live ranges from their beginnings. On the other hand, we have found cases where *forward* topological search order works better. The reason is exactly symmetric to the one given in the previous paragraph. Forward orders allow us to group all the uses of each operation close together and as high as possible in the schedule, thus shortening live ranges from their ends. This can be particularly useful in some codes with common subexpressions of high degree.

In some cases, shortening the live ranges has to take a back seat to producing code at all. In loops with many operations that are not fully pipelined or large strongly connected components, it is crucial to move these to the head of the scheduling list.

What scheduling heuristic could take account of so many different factors? Perhaps it would be possible to devise such a heuristic, but instead of searching for the *one* right heuristic, the SGI team took a different approach. A number of different scheduling list heuristics would be tried on every loop, ordered by likelihood to succeed. In common cases only one heuristic would be tried. Subsequent heuristics need only do well in cases where the other are weak.

The MIPSpro pipeliner uses 4 scheduling priorities. Experimental results show that lower quality results are obtained if any one is omitted. (See Section 4.2). Here is a brief description of the two primary ordering heuristics.

1. *Folded depth first ordering* — In the simple cases, this is just a depth first ordering starting with the roots (stores) of the calculation. But when there are difficult–to–schedule operations or large strongly–connected components, they are *folded* and become virtual roots. Then the depth first search proceeds outward from the fold points, backward to the leaves (loads) and forward to the roots (stores).
2. *Data precedence graph heights* — The operations are ordered in terms of the maximum sum of the latencies along any path to a root.

These two fundamental heuristics are modified in one or both of two ways to derive additional heuristics:

1. *Reversal* — the list can be reversed. Forward scheduling is particularly useful with the heights heuristic.
2. *A final memory sort* — pulling stores with no successors and loads with no predecessors to the end of the list.

The four heuristics actually in use in the MIPSpro compiler are:

1. *FDMS* — folded depth first ordering with a final memory sort,
2. *FDNMS* — folded depth first ordering with no final memory sort,
3. *HMS* — heights with a final memory sort, and
4. *RHMS* — reversed heights with a final memory sort;

See Section 4.2 for experimental results that show the complementary effect of these four.

2.8 Spilling

The SGI pipeliner has the ability to spill in order to alleviate register pressure. If a modulo scheduling pass is unsuccessful because of failure to register allocate, spills and restores are added to the loop and another attempt is made. Spills are added exponentially; the first modulo scheduling failure results in a single value being spilled; the second failure spills two additional values; the third spills 4 more, and so on. The process is capped at 8 modulo scheduling failures, implying that up to 255 values may be spilled before giving up. In practice this limit is never reached.

Spill candidates are chosen by looking at the best schedule that failed to register allocate. For each live range, a ratio is calculated: the number of cycles spanned divided by the number of references. The greater this ratio, the greater the cost and smaller the benefit of keeping the value in a register. Thus values with the largest ratios are spilled first.

2.9 Memory bank optimization

The MIPS R8000 is one of the simplest possible implementations of an architecture supporting more than one memory reference per cycle. The processor can issue two references per cycle, and the memory (specifically the second level cache which is directly accessed by floating point memory references) is divided into two banks of double-words, the even address bank, and the odd address bank. If two references in the same cycle address opposite banks, then both references are serviced immediately. If two references in the same cycle both address the same bank, one is serviced immediately, and the other is queued for service in a 1-element queue called the *bellows*. If this hardware configuration cannot keep up with the stream of references, the processor stalls. In the worst case there are two references every cycle all addressing the same bank, and the processor stalls once on each cycle, so that it ends up running at half speed.

Scheduling to avoid memory bank conflicts must address two issues, which are independent of the details of any particular banked memory system. First, at compile-time we rarely have complete information on the bank assignments of references. For example the relative banks of consecutive elements of a row of two-dimensional Fortran array depend on the leading dimension of the array, which is often not a compile-time constant. Second, modifying a schedule to enforce rules about the relative bank assignments of nearby references may increase register pressure, and therefore increase the length of the schedule more than the savings in memory stalls shortens the schedule.

The MIPSpro heuristic attempts to find known even–odd pairs of references to schedule in the same cycle — it does not model the bellows feature of the memory system. Before scheduling begins, but after priority orders have been calculated, for each memory reference *m* it forms the prioritized list *L(m)* of all other references *m'* for which *(m,m')* is known to be an even-odd pair. If *L(m)* is non–empty then we say *m* is *pairable*. Given the iteration interval, the number of memory references, and the number of pairable references, we can tell how many references should ideally be scheduled in known pairs, and how many references will have to be scheduled together even though they may turn out at runtime to reference the same bank. Until enough pairs have been scheduled together, whenever the compiler schedules a pairable memory reference *m*, it immediately attempts to schedule the first possible unscheduled element *m'* of *L(m)* in the same cycle as *m*. If this fails, it tries the following in order:

1. try to schedule another element of *L(m)* in the same cycle as *m*, or
2. try scheduling *m* in a different cycle, or
3. backtrack and try changing the scheduling of earlier operations in scheduling priority order.

Note that this process may change the priority ordering of scheduling, since memory reference with higher priority than *m*' are passed over in favor of scheduling *m*' with *m*.

The MIPSpro scheduler has two methods of controlling the impact of memory bank scheduling on register pressure. First, it measures the amount that priorities are changed due to pairing attempts during the entire scheduling process. If this measurement is large enough, and if register allocation fails, it tries scheduling again with reduced pairing requirements. Second, in the *adjusting pipe stages* part of scheduling (see Section 2.5), preserving pairing may require issuing a load one or more pipe stages earlier than would be necessary for simple legality of a schedule. Again, register allocation history is used to guide policy — if there has been trouble allocating registers, the scheduler is less willing to add pipe stages to preserve pairing.

Finally, since the minimal II schedule found first may not be best once memory stalls are taken into account, the algorithm makes a small exploration of other schedules at the same or slightly larger II, searching for schedules with provably better stalling behavior.

3.0 Software pipelining at McGill University — the optimal approach

3.1 Development of the ILP formulation

The interest in software pipelining at McGill stemmed from work on register allocation for loops on dataflow machines. This work culminated in a mathematical formulation of the problem in a linear periodic form [GaNi91,NiGa92]. It was soon discovered that this formulation can also be applied to software pipelining for conventional architectures. This formulation was then used to prove an interesting theoretical result: the minimum storage assignment problem for rate-optimal software pipelined schedules can be solved using an efficient polynomial-time method provided the target machine has enough functional units so resource constraints can be ignored [NiGa93]. In this framework, FIFO buffers are used to model register requirements. A graph coloring method can be applied subsequently on the obtained schedule to further decrease the register requirements of the loop. This is referred to as the *integrated* formulation and results in rate–optimal schedules with minimal register usage.

Subsequently, the McGill researchers extended their framework to handle resource constraints, resulting in a unified ILP formulation for the problem for simple pipelined architectures [NiGa92,GoAlGa94a]. The work was subsequently generalized to more complex architectures [AlGoGa95]. By the spring of 1995, this work was implemented at McGill in *MOST*, the *Modulo Scheduling Toolset*, which makes use of any one of several external ILP solvers. MOST was not intended as a component of a production compiler, but rather as a standard of comparison. Being able to generate optimal pipelined loops can be useful for evaluating and improving heuristics for production pipeliners (as this study demonstrates).

Because the McGill work is well represented in recent publications, we omit the details of the ILP formulation. The interested reader should consult [AlGoGa95] and then the other cited publications.

3.2 Integration with the MIPSpro compiler

The research at McGill left many open questions. Although the formulation of the software pipelining problem in ILP was appealing, could it also be useful? The McGill team did not believe that it could be part of a production compiler due to its exponential run time, but did expect that it could be used to evaluate such a compiler. How would it compare with a more specialized heuristic implementation? It was bound to be slower, perhaps even *much* slower; but how much better would its results be? Because heuristic approaches can have near–linear running time, they would certainly be able to handle larger loops. How much larger?

MOST was not a full software pipelining implementation; its output was a set of static quality measures, principally the II of the schedule found and the number of registers required, not a piece of runable code. It only targets were models that exhibited certain interesting properties, never a real commercial high performance processor. How well would it work with when targeted to a real processor? Were there any unexpected problems standing in the way of a full implementation, one that would generate runable code?

The opportunity to embed MOST in the MIPSpro compiler seemed perfect to answer these questions. In that context, MOST would enjoy a proven pipelining context — a full set of optimizations and analysis before pipelining and a robust post processing implementation to integrate the pipelined code correctly back into the program. It was particularly attractive to reuse the postprocessing code. Although modulo renaming, generation of pipeline fill and drain code, and other related bookkeeping tasks may seem theoretically uninteresting, they account for a large part of the job of implementing a working pipeliner. In the MIPSpro compiler, this postprocessing accounts for 18% of the lines of code in the pipeliner — about 6,000 out of 33,000 lines total in the pipeliner.

MOST was successfully embedded in the MIPSpro compiler over the summer of 1995.

3.3 Adjustment of McGill approach Due to this study

Over the course of this study, the McGill team found it necessary to adjust their approach in order to conduct a useful evaluation of SGI's production pipeliner. Sometimes a rate–optimal schedule with minimal register usage cannot be found in a reasonable amount of time. For the purpose of this study, we used 3 minutes as a limit on searches for rate–optimal schedules with minimal register usage. Increasing this limit doesn't seem to improve the results very much. When no optimal schedule is found within the given time limit, it is necessary to derive a good but not necessarily optimal schedule. As a result, the following adjustments were made to MOST during this study:

1. *Obtain a resource-constrained schedule first.* Using the ILP formulation of the integrated register allocation and scheduling problem was just too slow and unacceptably limited the size of loop that could be scheduled. For example, finding a schedule for the large N^3 loop in the tomcatv benchmark was far beyond the reach of the integrated formulation. The ILP formulation of resource–constrained scheduling could be solved considerably faster. This formulation finds schedules satisfying resource constraints, but does not address register allocation. When it cannot be solved within reasonable time limits, we would also not be able to solve the integrated problem, so separating the problems seems to serve as a useful filter against unnecessary compile time usage.

2. *Adjustment of the objective function* — Often it was not feasible to use the register optimal formulation (involving coloring) reported in [AlGoGa95]. Instead, the ILP formulation used minimized the number of buffers in the software pipeline. This objective function directly translates into the reduction of the number of iterations overlapped in the steady state of the pipeline.[1] The ILP solver for this phase was restricted to a fixed time limit. After that, it would accept the best suboptimal solution found, if any.

3. *Multiple priority orders* — One surprising result of this study was the discovery that the same multiple priority order heuristics that were used in the SGI pipeliner are also very useful in driving the McGill solver. The priority order in which the ILP solver traverses the branch–and–bound tree is by far the most important factor affecting whether it could solve the problem. MOST thus adopted SGI's policy of trying many different priority orders in turn until a solution is found.[2]

These adjustments have proved to be very important for our study. They enable MOST to generate R8000 code and greatly extend the size and number of loops it can successfully schedule

1. This is probably at least as important a parameter to optimize as register usage as it has a more direct impact on the size of the pipeline fill and drain code and thus on short trip count performance of the loop. Short trip count performance is the only performance impact of either minimization. Only the II affects the asymptotic performance.

2. In fact, the final implementation of MOST embedded in the MIPSpro compiler used most of the code from the SGI scheduler directly, replacing only the scheduler proper with MOST.

4.0 Experiments and results

4.1 Effectiveness of the MIPSpro pipeliner

To lend perspective to the comparative results in Section 4.4, we first present some results to demonstrate that the pipelining technology in the MIPSpro compiler is very effective. We use the 14 benchmarks in the SPEC92 floating point suite for this purpose, as SPEC has become an important standard of comparison. (We omit the SPEC92 integer suite because our software pipeliner unfortunately has little impact on those benchmarks.)

Figure 2 shows the results for each of the 14 SPEC92 floating point benchmarks with the software pipeliner both enabled and disabled. Not only is the overall result very good for a machine with a 75MHz clock, the effectiveness of the pipeliner is dramatic, resulting in more than 35% improvement in the overall SPEC number (calculated as the geometric mean of the results on each benchmark).

To be fair, this overstates the case for SGI's scheduling technology because disabling software pipelining does not enable any very effective replacement. The MIPSpro compiler leans heavily on software pipelining for floating point performance and there has been little work in tuning the alternative paths. In particular, with software pipelining disabled, the special inner loop optimizations and analysis described in Section 2.1 are disabled; without pipelining there is also no if–conversion and vector memory reference analysis. When software pipelining is disabled a fairly simple list scheduler is used.

We are often asked why we consider it important to be able to pipeline large loops. When a loop is large, the argument goes, there should be enough parallelism within each iteration without having to overlap iterations. Perhaps this is true, but *software pipelining* as we use the term is much more than just overlapping loop iterations. Rather it is a fundamentally different approach to code generation for innermost loops. We are willing to spend much more compile time on them than other parts of programs. We have a suite of special optimizations that we apply to them. We have a scheduler that backtracks in order to squeeze out gaps that would

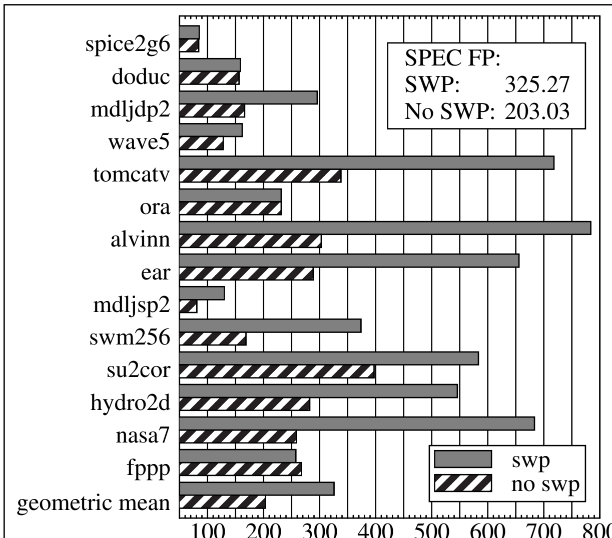

FIGURE 2. SPEC92 floating point benchmarks for the MIPSpro compiler run on an SGI Power Challenge with 75MHz R8000 processors. Results are given with software pipelining both enabled and disabled. Results are given as SPECmarks, i.e., performance multiples of VAX 780 run times.

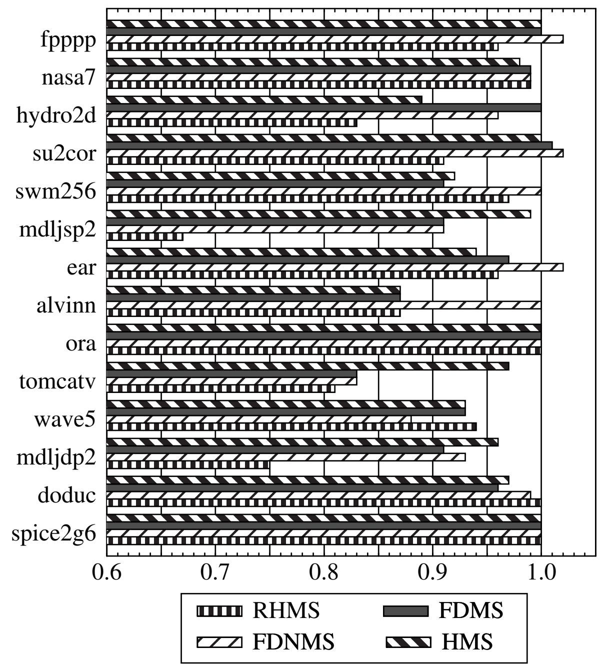

FIGURE 3. Three of the four priority list heuristics help to win at least one SPEC benchmark. Each bar gives the results of an experiment where all but one heuristic was disabled. The result is reported as the ratio over the numbers reported in Figure 2.

otherwise be left in the schedule. We have even gone so far (in this paper) as to consider an exponential scheduling technique.

4.2 The effect of multiple scheduling priority heuristics

Figure 3 shows the results of an experiment testing the effectiveness of the multiple scheduling heuristics discussed in Section 2.7. We tried compiling the SPEC benchmarks with each of the heuristics alone. No single heuristic worked best with all the benchmarks. In fact, three out of the four heuristics were required in order to achieve our best time for at least one of the benchmarks:

FDMS	hydro2d
FDNMS	alvinn, ear, swm256
HMS	tomcatv, mdljsp2

The fourth heuristic, RHMS, is useful for loops not important to any of the SPEC92 floating point benchmarks.

For several of the benchmarks, such as wave5, no single heuristic achieves a time equal to the overall best time shown in Figure 2. This is because the benchmarks consist of more than one important loop, and within a single benchmark each loop may require a different heuristic.

There are also a few benchmarks, e.g., ear, that do better by a few percent when limited to a single heuristic than when the "best" schedule is chosen from among all four heuristics. This is less strange than it may seem at first. The MIPSpro compiler uses only the II of a schedule to measure its quality and ignores some other factors that can have minor impacts on the performance of a schedule. In particular, the overhead to start-up and complete a loop is ignored when looking for a schedule, but can be relevant to the performance of short trip loops. (See Section 4.6.)

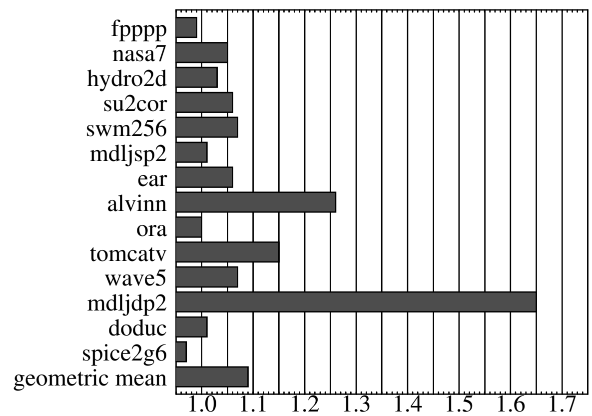

FIGURE 4. Effectiveness of the MIPSpro memory bank heuristics — Performance improvements from enabling these heuristics

4.3 Effectiveness of the MIPSpro memory bank heuristics

We measured the effectiveness of the MIPSpro compiler's memory bank heuristics discussed in Section 2.9. Figure 4 shows the performance ratio for code compiled with the heuristic enabled over the same code compiled with the heuristic disabled. Two benchmarks stand out as benefiting especially, alvinn and mdljdp2.

For alvinn, the explanation is fairly simple. This program spends nearly 100% of its time in two memory bound loops that process consecutive single precision vector elements. Because the R8000 is banked on double precision boundaries, the most natural pairings of memory references can easily cause memory bank stalls. In particular, one of the two critical loops is a dot product of two single precision vectors. Two natural memory reference patterns for this code are:

v[i+0], u[i+0]
v[i+1], u[i+1]

and

v[i+0], v[i+1]
u[i+0], u[i+1].

When (as in this case) both *u* and *v* are single precision and even aligned, both of these patterns batch 4 references to the same bank within two cycles, causing a stall. The memory bank heuristic prevents this, producing a pattern such as the following memory reference pattern instead:

v[i+0], v[i+2]
u[i+0], u[i+2]

This pattern is guaranteed to reference an even and an odd bank in each cycle and thus to have no stall.

For mdljdp2, the situation is different and more complicated. This loop is not memory bound; it has only 16 memory references out of 95 instructions. The pattern of memory references is complicated by the fact that there is a memory indirection and so the exact pattern of memory references cannot be known. Inspection of the generated code reveals that without memory bank heuristics, memory references with unknowable relative offsets are grouped together unnecessarily. The memory bank heuristics prevent that grouping and thus avoid risking stalls by preventing *any* pairing of memory references.

4.4 Comparison between ILP and SGI's heuristics

We wanted to answer the question whether an optimal approach could improve performance relative to the pipeliner in the

MIPSpro compiler. Remember that the primary goal of this study is to validate and improve SGI's heuristic techniques. Thus we wanted to give the ILP approach every possible advantage to expose weaknesses in the production compiler.

Unfortunately, there is a problem that could strongly favor the heuristic pipeliner — not every loop that can be scheduled by the SGI pipeliner can also be scheduled by the MOST scheduler *in reasonable time.* This is a particular problem because the penalty for *not* pipelining can be very high. (See Section 4.1.) We addressed this by using the heuristic pipeliner as a backup for MOST. Thus instead of falling back to the single block scheduler used when the MIPSpro pipeliner fails to schedule and register allocate a loop, it instead falls back to the MIPSpro pipeliner itself. In theory, this should reveal only deficiencies in the production compiler, never in the ILP pipeliner.

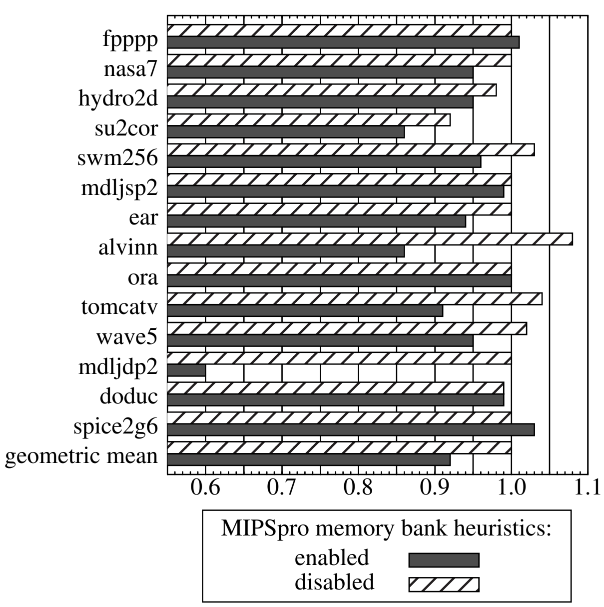

FIGURE 5. Relative performance of ILP scheduled code over MIPSpro results with and without memory bank pairing heuristics

4.5 Comparison of run time performance

The solid bars in Figure 5 show comparative results for the SPEC floating point benchmarks. This data shows the code scheduled by the SGI pipelined code outperforming the "optimal" code on 8 of the benchmarks. The geometric mean of the suite as a whole is 8% better for the heuristic scheduler than for ILP method. On one benchmark, alvinn, the code scheduled by MOST ran 15% *slower* than the code scheduled by the MIPSpro pipeliner.

How can this be? The design of the experiment should have prevented the ILP pipeliner from ever finding a worse schedule than could be found by MIPSpro. After all, the heuristics are available as a fallback when an optimal schedule cannot be found.

A large part of the answer is a dynamic factor introduced by the memory system of the R8000 and described briefly in Section 2.9. The MIPSpro pipeliner uses heuristics to minimize the likelihood of unbalanced memory references causing stalls. Currently, the ILP scheduling formulation does not optimize for this hardware. Thus it can generate code which has poorer dynamic performance than that generated by the MIPSpro scheduler.

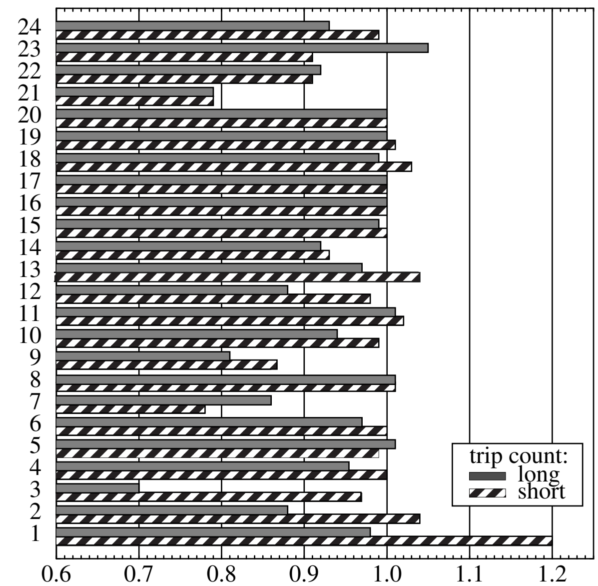

FIGURE 6. Relative performance of ILP scheduled code over MIPSpro results for each Livermore kernel.

In order to show the size of the dynamic effect introduced by the memory system, we report a second set of numbers. These compare the ILP results to code scheduled by the MIPSpro pipeliner *with its memory bank optimizations disabled.* These results are also shown in Figure 5, this time as striped bars. In this comparison, the ILP code performs within a range from just a little bit worse to about 5% better than the MIPSpro code. (See Section 4.3 for the direct comparison of MIPSpro results with and without memory bank heuristics.)

For two benchmarks, tomcatv and alvinn, the ILP does about 5% better than the MIPSpro code scheduled with memory bank heuristics disabled. Are these places where it is actually generating better code, or is it just another random factor introduced by the memory system? The latter is the case and the ILP code is just generating fewer memory stalls for these programs by chance. We know this because the performance of both programs is dominated by just a few loops which:

1. are scheduled by the MIPSpro compiler at their MinII,
2. have very long trip counts (300 for tomcatv, > 1000 for alvinn), and
3. are memory bound or nearly memory bound.

In fact these two programs were part of the original motivation for the MIPSpro pipeliner's memory bank optimization and served as a benchmark for tuning it.

4.6 Short trip count performance

Both the SGI and the McGill techniques strive to minimize the number of registers used by software pipelined steady states; however, the emphasis and motivations are somewhat different. At SGI the motivation was to make schedules that could be register allocated. Certainly this was important to the McGill team, but there was the additional goal of improving performance even for pipelines scheduled at the same II. How well do the two approaches perform in this regard, and how important is it in practice?

In fact register usage is only one of a number of factors that can affect the time to enter and exit a pipelined steady state -- its *overhead*. This overhead is constant relative to the trip count of the loop and thus increases in importance as the trip count decreases

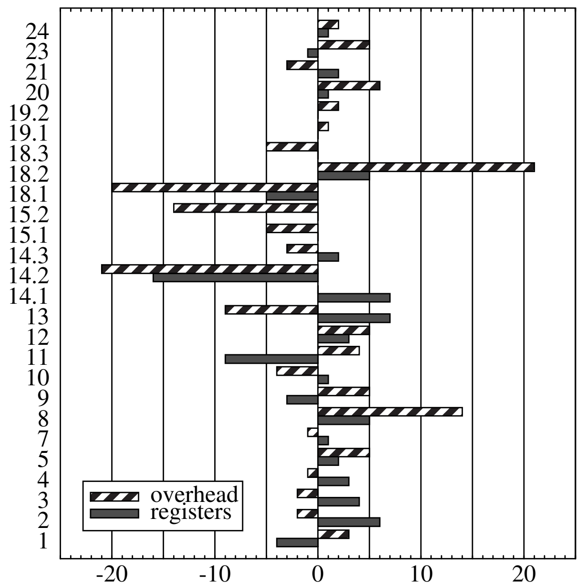

FIGURE 7. Difference MIPSpro minus ILP in second order static quality measures for each of the loops in the Livermore kernels given as absolute register numbers and instruction cycles

and asymptotically disappears in importance as the trip count increases. If we ignore dynamic factors such the memory system effect discussed in the previous section, then different schedules of a loop with identical IIs and different registers requirements differ at most in overhead so long as they both fit in the machine's available registers. After all, register usage can be compensated by spills and restores in the loop head and tail.

There are other factors beside register usage that influence pipeline overhead. Before the steady state can execute the first time, the pipeline has to be *filled*, and after the last execution of the steady state, the pipeline has to be *drained*. The number of instructions in the fill and drain code is a function of how deeply pipelined each instruction in the loop is and whether the less deeply pipelined instructions can be executed speculatively during the final few iterations of the loop.

How do the two schedulers compare with regard to the short trip count performance of the loops they generate? The well know Livermore Loops benchmark is particularly well suited for this measurement. It measures the performance on each of 24 floating point kernels for short, medium, and long trip counts. Figure 6 shows the relative performance of the two approaches on each loop for both the short and long trip count cases. These results show better performance for SGI scheduler in nearly all cases with both short and long trip counts. But as we have just seen, these results can be distorted by the effects of the machine's memory system. We'd like a way to make a more direct comparison.

To do this, we looked at some static performance information about the individual loops in the benchmark. For all the loops in the benchmark, the schedules produced by both pipeliners had identical IIs. Figure 7 shows the relative performance of the two pipeliners in terms of:

1. register usage measured in total number of both floating point and integer registers used, and
2. overall pipeline overhead, measured in cycles required to enter and exit the loop.

Overall pipeline overhead counts directly toward performance, while register usage is important *only* is so far as it impacts pipeline overhead. This chart shows two things quite clearly:

1. *Neither scheduler produces consistently better schedules by either measure of overhead.* The heuristic method uses fewer registers in 15 of the 26 loops and requires lower total overhead in 12.
2. *There is no clear correlation between register usage and overhead*. For 16 of the loops, the schedule with smaller overhead didn't use fewer registers.

(1) seems particularly surprising in light of the emphasis on register optimality at McGill, but a careful examination of Section 3.3 will shed some light. Even in the best case, the McGill schedules do not have optimal register usage, but only minimal usage of the modulo renamed registers or *buffers*, ignoring the within–iteration register usage. Moreover, for the purposes of this study, scheduling and buffer minimization were often performed in separate passes so that buffer usage is not really optimized over all possible schedules, but only over a rather small subset.

(2) should not be surprising in light of the discussion above. Saving and restoring the registers used by the steady state is only one of the things that needs doing in the loop prologue and epilog. An additional overlapped iteration in the steady state can have a far larger effect than the use of more registers.

4.7 Compile speed comparison

As expected, the ILP based pipeliner is *much* slower than the heuristic based one. Of the 261 seconds spent in the MIPSpro pipeliner while compiling the 14 SPEC benchmarks, 237 seconds are spent in the scheduler proper, with the rest spent in inner loop optimizations and post-pipelining bookkeeping. This compares to 67,634 seconds in the ILP scheduler compiling the same code.

5.0 Conclusions and future work

Only very rarely does the optimal technique schedule and allocate a loop at a lower II than the heuristics. In the course of this study, we found only one such loop. Even in that case a very modest increase in the backtracking limits of the heuristic approach equalized the situation.

The heuristic technique is much more efficient than MOST. This is especially important for larger loop bodies, where its greater efficiency significantly extends its functionality. In our experiments, the largest loop successfully scheduled by the SGI technique had 116 operations, while the largest optimal schedule found by MOST had 61 operations.

The ILP technique was not able to guarantee register optimality for many interesting and important loops. In order to produce results for these loops, the McGill team had to devise a heuristic fallback. This led to the result that neither implementation had consistently better register usage.

So long as a software pipelined loop actually fits in the registers, the number of registers used is not an important parameter to optimize since it is not well related to performance, even for short trip count loops. Future work in this direction should focus on the more complex task of minimizing overall loop overhead. We feel that there is much to learn in this area. Inherent in the modulo scheduling algorithm is a measure of how the results compare against a loose lower bound on optimality, the MinII (see Section 2.3.) This has always given us at least a rough idea of how much room there was for improvement in the iteration intervals of generated schedules. On the other hand, we know very little about the limits on the performance of software pipelined loops when the trip count is short. Perhaps an ILP formulation can be made that optimizes loop overhead more directly than by optimizing register usage.

We were unable to compare the MIPSpro memory heuristics with code that optimally uses the machine's memory system. This comparison would probably be useful for the evaluation and tuning of the heuristics. Frankly, we do not currently understand how well these heuristics work compared to how well they *could* work. A good solution to this problem could have interesting implications in design of memory systems.

This study had *not* produced any convincing evidence that there is much room for improvement in the SGI scheduling heuristics. But we *know* there is still room for improvement in several areas, most notably performance of loops with short trip counts and large loops bodies with severe register pressure. Without very significant strides toward a more efficient implementation of the ILP approach, we do not see how it can help with the latter problem. But short trip count performance is an important problem, especially in light of loop nest transformations such as tiling. And here we feel that the ILP approaches to software pipelining may yet have a significant role to play.

Acknowledgments

We wish to express our gratitude to a number of people and institutions. John Hennessy and David Wall provided invaluable editorial assistance — without their help, this paper would be much harder to understand. Eric Altman, Fred Chow, Jim Dehnert, Suneel Jain, Earl Killian, and Dror Maydan also helped substantially with the proofreading and editing. Monica Lam was the matchmaker of this project; she brought the SGI and McGill teams together in the first place. Members of the ACAPS lab at McGill, chiefly Eric Altman, built the MOST scheduling tool. Doug Gilmore made crucial contributions to the design of the SGI scheduler, in particular the idea of using more than one scheduling heuristic. Ross Towle was the original instigator of software pipelining at Silicon Graphics and the father of pratical software pipelining. We received financial support from Silicon Graphic and Canadian NSERC (Natural Science and Engineering Council). Ash Munshi has been primarily responsible for creating an atmosphere of great productivity and creativity at Silicon Graphics where the bulk of this work took place during the summer of 1995. Finally, we must thank the members of our families for their loving support and forbearance.

Bibliography

[AiNi88] A. Aiken and A. Nicolau. Optimal loop parallelizations. In *Proc. of the '88 SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 308-317, Atlanta, Georgia, June 1988.

[AlKePoWa83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. of the 10th Ann. ACM Symp. on Principles of Programming Languages*, pp. 177-189, January 1983.

[AlGoGa95] E. R. Altman, R. Givindarajan, and G.R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proc. of '95 SIGPLAN Conf. on Programming Language Design and Implementation*, La Jolla, Calif., June 1995.

[Altman95] E. R. Altman. *Optimal Software Pipelining with Functional Unit and Register Constraints*. Ph.D. thesis, McGill University, Montreal, Quebec, 1995.

[BiKeKr94] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer linear programming. In *Proc. of Conf. on Parallel Architectures and Compilation Techniques*, pp. 111-122, August 1994.

[BrCoKeTo89] P. Briggs, K.D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. of '89 SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 275-284, July 1989.

[Briggs92] P. Briggs. *Register Allocation via Graph Coloring*. Ph.D. thesis, Rice University, Houston, Texas, April 1992.

[DeTo93] J.C. Dehnert and R.A. Towle. Compiling for Cydra 5. *Journal of Supercomputing*, v.7, pp.181-227, May 1993.

[Eichenberger95] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Optimum modulo schedules for minimum register requirements. In *Proc. of '95 Intl. Conf. on Supercomputing,* pp. 31-40, Barcelona, Spain, July 1995.

[Feautrier94] P. Feautrier. Fine-grained scheduling under resource constraints. In *7th Ann. Workshop on Languages and Compilers for Parallel Computing*, Ithaca, N.Y., August 1994.

[GaNi91] G. R. Gao and Q. Ning. Loop storage optimization for dataflow machines. In *Proc. of 4th Ann. Workshop on Languages and Compilers for Parallel Computing*, pp. 359-373, August 1991.

[GaJo79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guideto the Theory of NP-Completeness*. W.H. Freeman and Co., New York, N.Y., 1979.

[GaSc91] F. Gasperoni and U. Schwiegelshohn. Efficient algorithms for cyclic scheduling. *Res. Prep. RC 17068*, IBM TJ Watson Res. Center, Yorktown Heights, N.Y., 1991.

[GoAlGa94a] R. Govindarajan, E. R. Altman, and G. R. Gao. A framework for rate-optimal resource-constrained software pipelining. In *Proc. of CONPAR94-VAPP VI, no. 854, Lecture Notes in Computer Science*, pp. 640-651, Linz, Austria, September 1994.

[GoAlGa94b] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture,* pp. 85-94, San Jose, Calif., November-December 1994.

[Hsu94] Hsu, P., Designing the TFP Microprocessor, *IEEE Micro,* April 1994, pp. 23-33.

[Huff93] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the '93 SIGPLAN Conf. on Programming Language Design and Implementation,* pp. 258-267, Albuquerque, N. Mex., June 1993.

[Lam88] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the ' 88 SIGPLAN Conf. on Programming Lanugage Design and Implementation*, pp. 318-328, Atlanta, Georgia, June 1988.

[Lam89] M. Lam. *A systolic array optimizing compiler.* Kluwer Academic Publishers, 1989.

[MoEb92] S. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture,* pp. 55-71, Portland, Ore., December 1992.

[NeWo88] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[Nemhauser94] G. Nemhauser. The age of optimization: Solving large-scale real-world problems. *Operations Research*, 42(1):5-13, January-February 1994.

[NiGa92] Q. Ning and G. Gao. Optimal loop storage allocation for argument-fetching dataflow machines. *International Journal of Parallel Programming,* v. 21, no. 6, December 1992.

[NiGa93] Q. Ning and G. Gao. A novel framework of register allocation for software pipelining. *In Conf. Rec.of the 20th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages,* pp. 29-42, Charleston, S. Carolina, January 1993.

[Pugh91] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proc. Supercomputing '91*, pp. 18-22, November 1991.

[RaGl81] B. R. Rau and C. D. Glaser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14 Ann. Microprogramming Workshop*, pp. 183-198, Chatham, Mass., October 1981.

[RaFi93] B. R. Rau and J.A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, v.7, pp.:9-50, May 1993.

[Rau94] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture,* pp. 63-74, San Jose, Calif., November-December 1994.

[Tou84] R.F. Touzeau. A FORTRAN compiler for the FPS–164 scientific computer. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 48–57, Montreal, Quebec, June 17–22, 1984.

[Warter92] N.J. Warter, John W. Bockhaus, Grant E. Haab, and K. Supraliminal. Enhanced modulo scheduling for loops with conditional branches. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture,* pp. 170-179, Portland, Ore., December 1992.