# Optimal Software Pipelining Through Enumeration of Schedules * 

Erik R. Altman ${ }^{1}$ and Guang R. Gao ${ }^{2}$<br>${ }^{1}$ IBM T.J. Watson Research Center, Yorktown Heights, NY, USA ${ }^{2}$ McGill University School of Computer Science, Montréal, Québec, Canada


#### Abstract

Resource-constrained software-pipelining has played an increasingly significant role in exploiting instruction-level parallelism and has been drawing intensive academic and industrial interest. The challenge is to find schedule which is optimal: i.e. the fastest possible schedule under given resource constraints while keeping register usage minimal. One interesting problem is open: the design of an algorithm which ensures that such an optimal schedule will always be found and with a cost which is manageable in practice. In this paper, we present a novel modulo scheduling algorithm which provides a solution to this open problem.


The proposed algorithm has been successfully implemented and tested under MOST - the Modulo Scheduling Testbed developed at McGill University. Unlike the existing optimal modulo scheduling approach based on integer linear programming (ILP) [6], our approach employs a search procedure which directly exploits the program structure in terms of its dependence graphs. Experimental results on more than 1000 loops from popular benchmark programs show that our method often finds a schedule faster. In addition, with our approach many loops require a surprisingly small number of schedules be searched to obtain an optimal solution, thus making the approach quite feasible.

## 1 Introduction

Software pipelining $[1,8,9,10,11,13,12]$ is widely used for loop scheduling. Advances in computer architecture - hardware and software - yield a large solution space containing many legal software pipelined schedules. In exploring the space of good compile-time schedules, it is important to find the fastest software-pipelined schedule for a particular set of machine resources, e.g. function units and registers.

The performance of a software-pipelined schedule can be measured by the initiation interval II between successive iterations. Thus "highest performance" refers to the schedule with the minimum $I I$. A schedule with minimum initiation interval $M I I$ is called a rate-optimal schedule. In this paper, we address the following software pipelining problem:

Problem: [OPT] Given a loop $\mathcal{L}$ and a machine architecture $\mathcal{M}$, construct a schedule whose initiation interval $I I=M I I$ is the smallest possible.

Previous approaches to solving OPT fall in one of two categories:

- Inexact or heuristic, i.e. no guarantee is made that the schedule obtained has initiation interval MII [1, 8, 9, 10, 13, 12].
- Exact where a guarantee is made that the schedule found has initiation interval MII. Most previous exact approaches (that are applicable to loops with arbitrary data dependence graphs (DDG's)) were based on Integer Linear Programming (ILP) $[3,4,5,6]$. [7] used graph structure and algorithms, but did not consider registers.

[^0]proc: main (DDG)
- Create Order_List
- Set Curr_Node = Order Listfo]
- gen_sched (Curr_Node, Rank = 0)
\}
proc: gen_sched (Curr_Node, Rank)
pr
- \{Low, High $\}=$ get_time_range (Curr_Node)
- for (time = Low; time $\leq$ High; time++) \{
- t[Curr_Node] $=$ time
- if (Rank $==$ Last) $\Rightarrow$ Full Schedule
- else gen_sched (Order_List[Rank + 1],
\}
\}

Fig. 1. Basic Enumeration Algorithm
ILP is a powerful and general technique that has been applied to a wide variety of problems. Consequently many off-the-shelf packages are available for it. Furthermore, for a significant number of loops the schedules obtained are better than those obtained using heuristics [6]. Still ILP has drawbacks. (1) Use of a general purpose package hinders adoption of pruning techniques specific to a particular application (such as software pipelining). (2) The problem must be expressable in a set of linear constraints. While much of software pipelining can be expressed in this manner, use of a realistic model - such as modeling of register coloring-requires a very large set of constraints and takes a prohibitively long time, even for many small loops [2].

As an alternative to ILP, we propose enumeration of schedules. The idea of enumerating schedules is not new, having been proposed at least as early as 1981 by Rau and Glaeser [13]. However, Vegdahl's [16] is the only the enumeration approach of which we are aware which belongs to the "exact" class outlined above. Though "exact", Vegdahl's approach explored a much larger solution space than necessary, and consequently was practical on a far smaller set of loops than the approach proposed here.

The enumeration approach described here is also "exact" and is based on a natural representation of software pipelining. By natural, we mean the approach is constructive. Each complete schedule is built from successively larger partial schedules according to both the dependence constraints between operations and the resource availabilities of the target architecture. In constructing schedules our enumeration approach does not require any awkward reworking of constraints into a linear form as required by some of the ILP based exact methods $[2,3,5,6]$.

We have implemented our enumeration approach in MOST (McGill's MOdulo Scheduling Testbed) [2]. These advantages result in a significant improvement in the time required to find a schedule for many loops. The fact that such an improvement occurs suggests that a relatively small number of schedules need be enumerated to guarantee an exact solution. Since solution of the OPT Problem problem is NP-Hard, it is indeed good news that most loops require such a relatively small number of schedules be enumerated.

The remainder of this paper is organized as follows. In the following section, we illustrate our enumeration approach with the help of an example. Section 3 discusses the two key conditions needed to guarantee that the enumeration approach solves the OPT Problem. Section 4 provides a brief summary of experimental results. Concluding remarks are presented in Section 5.

## 2 Enumeration Algorithm: Outline and Illustration

The heart of the enumeration algorithm is quite simple, and is embodied in the the routine gen_sched in Figure 1(b). The full enumeration algorithm can be found in [2]. As we shall describe in Section 3, the major subtlety lies in choosing a good


Fig. 2. Example of Enumeration Algorithm.
order in which to schedule the nodes, and selecting the proper range of times at which to schedule each node.

The gen_sched routine is invoked initially by the routine gen-main shown in Figure 1(a). Subsequently gen_sched schedules the current node $X$ with which it is called at a sufficiently large set of times as to guarantee enumeration of an optimal schedule. For each of these times $t[X]$, gen_sched may invoke itself recursively with the next node to be scheduled from Order_List. This process is illustrated by the example in Figure 2. In this example, assume that $I I=2$ and that all nodes have unit execution time. (This example was chosen to illustrate the workings of our enumeration approach, and not as a difficult loop to schedule. On the other hand, the DDG in Figure 3 can present difficulties to naive methods, as will be described in Section 3.)

In Figure 2(a), gen_main begins by ordering the nodes in the DDG at the left of the Figure. For reasons that will be explained below, the order is chosen to be A-B-C-D-E. gen_main then invokes gen_sched with the first node, Order_List[0] $=A$. Gen sched first calculates the range of times at which A should be scheduled. Since A is the first node scheduled, it serves as the anchor relative to which all other nodes will move. Hence both the Low and High values of A are 0. Once the range for A is established, gen_sched iterates through it, assigning $t[A]$ each successive value of time (or in this case the single time $t[A]=0$ ). Given a value for $t[A]$, gen_sched is ready to schedule next node in Order_List, node B. This is done by recursively invoking
gen_sched as shown at the bottom of Figure 2(a).
Given that $t[\mathrm{~A}]=0, t[\mathrm{~B}] \geq 1$. However, there is no upper limit to the set of times at which B can be scheduled. Thus, to make the enumeration approach feasible some rule for pruning the set of times must be used. Since we are interested in enumerating exact schedules, this pruning must not eliminate any times which could result in a minimum register schedule fitting the function unit constraints of the target architecture. As has been previously observed [13], placing B at $\boldsymbol{I I}=2$ consecutive times is sufficient to guarantee that some offset will be found which function units are available to execute B. For acyclic DDG's, list scheduling nodes in a topological order guarantees that $\boldsymbol{I I}$ consecutive offsets will always be available. However, aside from not applying to cyclic DDG's, this list scheduling approach provides no guarantee of register minimality.

Intuitively, for enumeration of a minimum register schedule, the set of times $t[B]$ should be as close to $t[\mathrm{~A}]$ as possible. Hence if $t[\mathrm{~B}]$ is placed at $I I=2$ consecutive times for function unit purposes, for register purposes $t[\mathrm{~B}]$ should have Low=1 and High=2 - the closest values to $t[A]=0$. A fuller set of register minimization requirements is described in Section 3 and a proof of their sufficiency may be found in [2].

Gen_sched first assigns $t[B]=1$, then invokes itself recursively with node $C$ the next node in Order_List. Due to the cycle B-C-B, B places both an upper and lower bound on when $C$ may execute and hence Low=High=2. Once $t[C]$ is assigned 2, the next node in Order_List is $D$. Scheduling of $D$ is very similar to the scheduling of node $B$ - only predecessors have been scheduled and there is no upper bound for a legal $t[D]$. Thus like B, D is assigned the $I I=2$ times closest to its immediate predecessor C, i.e. Low $=3$ and High $=4$. Hence $t[\mathrm{D}]$ is assigned 3 and gen_sched is recursively invoked once more with the last node in Order_List, node E.

The placement of E is constrained by both $t[\mathrm{~A}]=0$ and $t[\mathrm{D}]=3$, giving Low=1 and High=2. The iteration first assigns $t[E]=1$ yielding Complete Schedule 1 shown at the bottom of Figure 2(e). In the next iteration $t[\mathrm{E}]=2$ yielding Complete Schedule 2 shown at the bottom of Figure 2(e). At this point gen_sched returns and $t[D]$ is incremented to 4 in the next iteration of the for loop. Then gen sched is called again with node E , but this time $L o w=1$ and High=3. This general process continues yielding the set of 12 schedules depicted at the bottom of Figure 2(f).

## 3 Guaranteeing Enumeration of an Optimal Schedule

Having seen the basic operation of the enumeration algorithm, it remains to show that the set of schedules enumerated contains at least one that is optimal for the target architecture, i.e. fitting the function unit constraints and minimizing registers. Space limitations do not permit a formal proof that the proposed algorithm does this. However in Subsection 3.1, we state two theorems to this effect and present an intuitive outline of their proof. A full proof may be found in [2].

As just noted in Section 2, a schedule fitting function unit constraints can be guaranteed by placing each node in at least $I I$ consecutive time slots - if $I I$ consecutive time slots are legal. It remains to consider (1) when II consecutive times are not available and (2) the range of times at which each node must be scheduled to guarantee enumeration of a schedule which uses minimum registers. Luckily, both (1) and (2) can be addressed within the same framework. However, before describing this framework, we give an example to motivate it.

To see that scheduling a node in II consecutive offsets does not guarantee a minimum register schedule, consider the DDG in Figure 3. Assume that nodes A and G have been scheduled, but not node $J$. As nodes $\mathrm{H}, \mathrm{J}$, and K are on the shorter path from $A$ to G (assuming all nodes have unit execution time), they can "slide" between A and G. Furthermore, they should clearly slide as close to $G$ as possible in order to have a

When must these nodes be scheduled for a minimum register schedule?


Fig. 3. DDG in which $I I$ offsets are insufficient for register minimization.
minimum register schedule. However placing them in only the $I I$ offsets closest to A fails to accomplish this.

More generally 5 possible relationships of the current node to previously scheduled nodes must be considered when determining the time range ([Low,High]) for the current node [2]. These cases should be considered in sequence with the first matching case taking precedence when more than one fits. These five relationships are illustrated by the DDG below. (This simple DDG was constructed solely to illustrate these five relationships and is not meant to be representative of real loop DDG's.) Assume that the nodes are scheduled in alphabetical order once again.


Rel 1 A: Ist node scheduled
Rel 2 B: 1st node of connected component
Rel 3 C: Node in SCC already schedule

-Relationship 1: A is the first node scheduled and serves as the anchor, i.e. Low $=0$ and High=0.

- Relationship 2: B is the first node scheduled in its connected component (in the undirected sense). Hence the placement of $A$ in no way constrains $B$ nor does it affect register usage of B . Hence to guarantee a schedule fitting function unit requirements $B$ is placed in $I I$ consecutive offsets, i.e. Low $=0$ and $H i g h=I I-1$.
-Relationship 3: C is in the same strongly connected component (in the directed sense) as node $B$, so $t[B]$ places both a lower and upper bound on $t[C]$ and consequently Low and High are set to these bounds.
- Relationship 4: D is on no undirected cycle with nodes which have already been scheduled. ( $D$ is on an undirected cycle with nodes $E, F, G$, and $H$, but none of these nodes has been scheduled.) Increasing $t[D]$ increases register requirements on edge $A-D$, but cannot decrease them elsewhere since $E, F, G$ and $H$ have not been scheduled. Likewise placement of $D$ does nothing to limit the offsets in which $E, F$, G and H may be placed. The fact that register effects from scheduling node D are isolated to edge A-D reflects a general property of nodes with Relationship 4 -
that placement of the node affects register requirements on only a single edge [2]. Consequently placing $D$ at the $I I$ the consecutive times closest to $t[A]$ is sufficient for both register and function unit purposes.
- Relationship 5: E is on an undirected cycle with nodes D, F, G, and H; and node D has been scheduled. Thus the "join" node $F$ of the cycle must be within a fixed range of $D$ in order to minimize registers. Likewise, for each of the possible assignments to nodes F, G, and H, E must assume all II offsets. In order for node E to "slide" from as close to $D$ as possible to as close to $F$ as possible, the possible range of $F$ must be computed, even though $F$ has not yet been scheduled. This can be done by assuming first that node $H$ adjoining scheduled node $D$ is scheduled as far as need be from D . Then given this maximum time for $t[\mathrm{H}]$, a similar maximum can be calculated for node G . Finally, given the maximum for $t[\mathrm{G}]$, the maximum for $t[\mathrm{~F}]$ can be computed. Then the High for $t[E]$ is the maximum $t[F]$ less the distance from $E$ to $F$. Likewise Low for $t[E]$ is $t[D]$ plus the distance from $D$ to $E$. More generally a node can be on multiple undirected cycles, and in this case the widest bounds arising from each of them must be used [2].
Topological Scheduling Order: A, B, C
Let $t[\mathrm{~A}]=0$
$\rightarrow$ No constraint on $t[\mathrm{~B}]$
$\rightarrow$ Choose $t[\mathrm{~B}]=100$
$\rightarrow$ More than min registers on edge AC

DDG in which topological node ordering is bad
As mentioned in Section 2, it is also essential to schedule nodes in a proper order to be certain of enumerating an optimal schedule. The essential characteristic of a proper order is that except for the first node scheduled of each (undirected) connected component, each node must have an immediate predecessor and/or an immediate successor previously scheduled. This order may or may not be a topological order. To see why this ordering is needed, consider the DDG above. It is clear that if we wish to minimize registers, the placement of A limits the placement of B. For example, if $\boldsymbol{I I}=2$ and $t[A]=0$, then assigning $t[\mathrm{~B}]=100$ will result in a schedule using far more registers than necessary since the value of $A$ will need to be kept live for 50 iterations. If the nodes are scheduled in the order $A-C-B$, then the effect of scheduling node $A$ is carried through node $C$ and an appropriate range for $t[B]$ may be chosen. However, if the topological order is $\mathrm{A}-\mathrm{B}-\mathrm{C}$ is used, the placement of A places no upper bound on $t[\mathrm{~B}]$ and consequently a non-minimum register schedule may result.

### 3.1 Optimality Theorems

By scheduling nodes in a proper order, and by scheduling each node $X$ at a set of times based on the relationship of $X$ to already scheduled nodes, it can be proved that an optimal schedule will be enumerated [2]. Although the full proof is too lengthy to present here, we state two definitions and sketch the proof of two theorems.

Definition 3.1 Let $t[\mathrm{X}]$ be first time each node X in a loop executes. The set of $t[\mathrm{X}]$ for each node in the loop form the loop schedule.

Figure 2 illustrates a DDG and 12 possible schedules (sets of $t[\mathrm{X}]$ values).
Definition 3.2 Given a loop schedule, $t[\mathrm{X}]$, a loop kernel is the set of offsets $o[\mathrm{X}]=$ $t[X] \bmod I I$.

Since $I I=2$, in Schedule 1 from Figure 2,

The loop kernel specified by the $o[\mathrm{X}]$ values reveals that when the loop is in its steady state, nodes $A$ and $C$ execute together at offset 0 , while nodes $B, D$, and $E$ execute together at offset 1.

Theorem 3.1 The enumeration algorithm enumerates at least one schedule for all legal loop kernels.

Enumeration of all legal loop kernels insures that a schedule will be found fitting function unit constraints (if such a schedule exists). The essential idea in the proof makes use of the 5 relationships described at the start of Section 3, between the current node and previously scheduled nodes. The proof is inductive and shows that for each of the 5 relationships, that if the previously scheduled nodes were given a sufficiently large range of times, the enumeration algorithm will give the current node a sufficiently large range of times as well.

Theorem 3.2 At least one of the schedules enumerated for each loop kernel uses the minimum possible number of registers of any schedule corresponding to that kernel.

Since a minimum register schedule is enumerated for each kernel, a schedule meeting function unit constraints and using the minimum number of registers must be enumerated if such a schedule exists for a particular II. This proof is also inductive and is carried out along the lines of the proof for Theorem 3.1.

## 4 Experimental Results

The enumeration approach described in Section 3 has been implemented in MOST (our MOdulo Scheduling Testbed). MOST also implements several other modulo scheduling approaches including another exact approach based on integer linear programming [2, 4, $5,6]$, as well as several leading inexact heuristic approaches. As input to MOST, we used 1008 single basic block benchmark loops with less than 64 low level operations. The loops were taken from Spec92fp, Spec92int, the NAS Kernels, Linpack, and the Livermore Loops. We modeled a target architecture with 4 -wide issue and with function units and latencies representative of the values found in modern superscalar architectures $[14,15]$. Because of space limitations, full results cannot be included here, however they may be found in [2]. The major findings are summarized below.

- In terms of the number of loops scheduled at the minimum initiation interval $\boldsymbol{I}_{\text {min }}$, the performance of the enumeration and ILP approaches is similar. Enumeration holds a significant advantage when register minimization is important.
- However, the median time of the enumeration approach was significantly shorter.

The enumeration approach also allowed us to discover two general characteristics of the set of modulo schedules for a loop:

- The schedule space for most loops contains at least one schedule in which registers and all function units are simultaneously minimized.
- However, most loops have many other schedules as well. In fact schedules generally exist with all combinations of registers and function units - no tradeoff between the two is evident.


## 5 Conclusion

This paper has presented a novel software pipelining approach which enumerates a sufficiently large set of schedules so as to guarantee inclusion of an optimal one, i.e. one with $I I=M I I$, using minimum registers, and fitting the target architecture. This enumeration approach is easily adapted to resource minimization, making it suitable for use in synthesis and architectural definition as well as in compilers.

An experimental testbed MOST, was constructed to evaluate the proposed approach. Enumeration compared favorably with integer linear programming approaches, especially in terms of register minimization.

## Acknowledgements

R. Govindarajan had many useful suggestions and criticisms. Kemal Ebcioglu, Artour Stoutchinin, Jian Wang, and the anonymous referees also provided many useful observations. We thank them.

## References

1. Alexander Aiken and Alexandru Nicolau. A realistic resource-constrained software pipelining algorithm. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, Advances in Languages and Compilers for Parallel Processing, pages 274-290. Mass., 1991.
2. Erik R. Altman. Optimal Software Pipelining with Function Unit and Register Constraints. PhD thesis, McGill U., Montréal, Qué., Oct. 1995.
3. Erik R. Altman, R. Govindarajan, and Guang R. Gao. Scheduling and mapping: Software pipelining in the Presence of Structural Hazards. In Proc of the SIGPLAN '95 Confon Programming Language Design and Implementation, pages 139-150, La Jolla, Calif, June, 1995.
4. Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Optimum Modulo Schedules for Minimum Register Requirements. In Conf. Proc., 1995 Intl. Conf. on Supercomputing, pages 31-40, Barcelona, Spain, Jul. 3-7, 1995.
5. P. Feautrier. Fine-grain Scheduling under Resource Constraints. In Seventh Annual Workshop on Languages and Compilers for Parallel Computing, Ithaca, USA, August 1994.
6. R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resourceconstrained rate-optimal software pipelining. In Proc. of the 27 th Annual Intl. Symp. on Microarchitecture, pages 85-94, San Jose, Calif., Nov. 30-Dec.2, 1994.
7. C. Hanen. Study of a NP-Hard Cyclic Scheduling Problem: The Recurrent Job-Shop. In European Journal of Operations Research, (72)1994, pages 82-101.
8. Richard A. Huff. Lifetime-sensitive modulo scheduling. In Proc of the SIGPLAN '93 Conf. on Programming Language Design and Implementation, pages 258-267, Albuquerque, N. Mex., June, 1993.
9. Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation, pages 318-328, Atlanta, Georgia, Jun. 22-24, 1988.
10. Soo-Mook Moon and Kemal Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In Proc. of the 25th Annual Intl. Symp. on Microarchitecture, pages 55-71, Portland, Ore., Dec. 1-4, 1992.
11. Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. In Conf. Rec. of the 20th Annual Symposium on Principles of Programming Languages, pages 29-42, Charleston, South Carolina, Jan. 10-13, 1993.
12. B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. J. of Supercomputing, 7:9-50, May 1993.
13. B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In Proc. of the 14th Annual Microprogramming Work., pages 183-198, Chatham, Mass., Oct. 12-15, 1981.
14. James E. Smith and Shlomo Weiss. PowerPC 601 and Alpha 21064: A Tale of Two RISCs. Computer, 27(6):46-58, Jun. 1994.
15. Peter Song and Marvin Denman. The PowerPC 604 RISC Microproc. Motorola; IBM, 1994.
16. Steven R. Vegdahl. A dynamic-programming technique for compacting loops. In Proc. of the 25th Ann. Intl. Symp. on Microarchitecture, Portland, USA, Dec, 1992.

[^0]:    * This work was supported by IBM and by research grants from NSERC (Canada) and MICRONET Network Centers of Excellence (Canada).

