Optimum Modulo Schedules for Minimum Register Requirements

Alexandre E. Eichenberger and Edward S. Davidson

Advanced Computer Architecture Laboratory
EECS Department, University of Michigan
Ann Arbor, MI 48109-2122
alexe,davidson@eecs.umich.edu

Santosh G. Abraham

Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
abraham@hpl.hp.com

Abstract

Modulo scheduling is an efficient technique for exploiting instruction level parallelism in a variety of loops, resulting in high performance code but increased register requirements. We present a combined approach that schedules the loop operations for the highest steady state throughput and minimum register requirements. Our method determines optimal register requirements for machines with finite resources and for general dependence graphs. We compare the performance of this and other modulo schedulers for a benchmark of 629 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels. Measurements demonstrate the potential of register-sensitive modulo schedulers, which will be useful in evaluating the performance of register-sensitive modulo scheduling heuristics.

KEYWORDS: Register-sensitive modulo scheduling, software pipelining, loop scheduling, instruction level parallelism, VLIW, superscalar.

1 Introduction

Current research compilers for VLIW and superscalar machines focus on exposing more of the inherent parallelism in an application to obtain higher performance by better utilizing wider issue machines and reducing the schedule length of a code. There is generally insufficient parallelism within individual basic blocks and higher levels of parallelism can be obtained by also exploiting the instruction level parallelism among successive basic blocks. Software pipelining is a technique that exploits the instruction level parallelism present among the iterations of a loop by overlapping the execution of consecutive loop iterations. With sufficient overlap, some machine resources can be fully utilized, resulting in a schedule with maximum steady state throughput.

Modulo scheduling [1][2][3] is a software pipelining technique that results in high performance code while requiring only modest compilation time by restricting the scheduling space. It uses the same schedule for each iteration of a loop and it initiates successive iterations at a constant rate, i.e. one *Initiation Interval* (II clock cycles) apart. This restriction is known as the *modulo scheduling constraint* and forces a resource to be used by one iteration no more than once within each set of times that are congruent modulo II.

The scope of modulo scheduling has been widened to a large variety of loops. Loops with conditional statements are handled using hierarchical reduction [4] or IF-conversion [5]. Modulo scheduling has also been extended to a large variety of loops with early exits, such as while loops [6][7]. Furthermore, the code expansion due to modulo scheduling can be eliminated by using special hardware, such as rotating register files and predicated execution [8].

Since modulo scheduling exploits high levels of parallelism, it results in higher register requirements because more values are needed to support more concurrent operations. This effect is inherent to parallelism and will be exacerbated by wider machines and higher latency pipelines [9]. As a result, developing scheduling techniques that exploit instruction level parallelism while containing the register requirements is crucial to the performance of future machines. Our research aims at understanding the fundamental relationship between instruction level parallelism and register requirements for a set of benchmarks under realistic assumptions.

In this paper, we present a register-sensitive modulo scheduling method that finds a schedule with the highest steady state throughput over all modulo schedules, and the minimum register requirements among such schedules. This method applies for loop iterations that consist of a single basic block with a general dependence graph. Loop iterations with multiple basic blocks are IF-converted; however, predicates are ignored when evaluating the register requirements of a schedule. Our scheduling method satisfies arbitrary resource and dependence constraints and minimizes MaxLive [3], the minimum number of registers required to generate spill-free code. *MaxLive* corresponds to the maximum number of live values at any single cycle of the loop schedule.

To improve the execution time of register-sensitive modulo schedulers, we propose a technique that removes redundant edges of the dependence graph. We demonstrate the conditions under which scheduling edges as well as register edges may safely be removed from the dependence graph of a loop. This technique is applicable to any modulo scheduler. Also, we demonstrate that the search space for modulo schedules is bounded and we use this bound to improve the execution time of our modulo scheduler.

This paper contributes an optimum solution for register-sensitive modulo scheduling of loops with arbitrary dependence graphs on machines with arbitrary resource constraints.

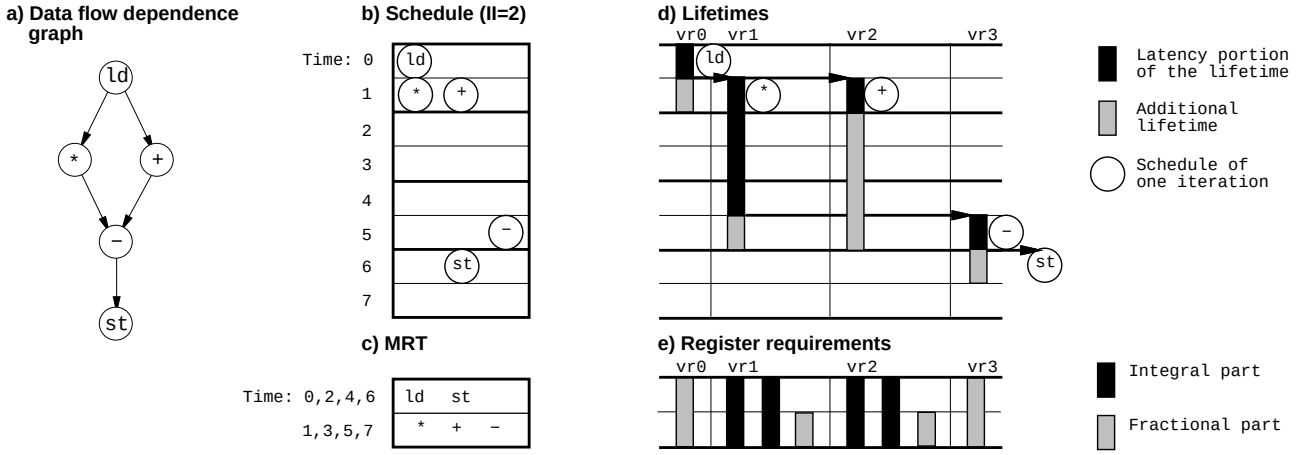

Figure 1: Modulo schedule for the kernel `y[i] = x[i]^2-x[i]-a` with minimum buffer requirements.

We compare the performance of this and other modulo schedulers for a benchmark of 629 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels for the Cydra 5 machine, a machine with complex resource usage [10]. Though the general algorithm presented in this paper may be too expensive to be integrated in a compiler, the improvements obtained in register requirements motivate the development of better heuristics for combined scheduling and register allocation for modulo-scheduled loops. This method is thus useful in assessing the performance of modulo-scheduling heuristics.

This work builds upon the modulo scheduling algorithm proposed by Govindarajan, Altman, and Gao [11] in which the entire modulo scheduling space is searched to find a schedule with minimum buffer requirements. This algorithm produces schedules for machines with finite heterogeneous functional units. Our work extends this algorithm in that we do not approximate registers by conceptual FIFO buffers that are reserved for an interval of time that is a multiple of II. Because of its accurate modeling of the register requirements, our method results in lower register requirements in 25% percent of the loops in our benchmark.

This work generalizes our previous work on optimizing an existing modulo schedule for a given loop to reduce its register requirements [12]. Although our previous work also modeled the register requirements of a loop accurately, a schedule with minimum register requirements was not sought over the entire search space of valid modulo schedules. Instead, our previous work considered only schedules in which operations of an existing schedule where displaced by multiples of II cycles. Also, our previous work did not present any experimental results. Because of its increased search space, the new method of this paper results in lower register requirements in 11% percent of the loops in our benchmark.

This work also generalizes the linear-time algorithm for determining minimum register requirements by Mangione-Smith *et al* [9], which requires that each virtual register be used no more than once. Our scheduling method handles general dependence graphs, including loop-carried dependence and common sub-expressions.

This work complements the register allocation algorithm presented by Rau, Lee, Tirumalai, and Schlansker [8], which achieves register allocations that are within one register of the MaxLive bound for the vast majority of modulo-scheduled loops on machines with rotating register files and predicated execution to support modulo scheduling. For machines without hardware support, their algorithm typically achieves register allocations that are within four registers of the MaxLive bound, after loop unrolling and consequent code expansion.

Finally, in contrast to the work by Ning and Gao [13] and Ramanujam [14], we allow finite machine resource constraints. Moreover, we do not approximate the register pressure by minimizing the average number of live values over all cycles, but rather we explicitly minimize the maximum number of live values at any single cycle of the loop.

In this paper, we present the impact of modulo scheduling on register requirements in Section 2. We develop an integer linear programming model that minimizes the register requirements for a given initiation interval in Section 3. We present an algorithm that finds a schedule with the highest steady state throughput over all modulo schedules, and the minimum register requirements among such schedules in Section 4. We propose a technique that removes redundant edges of the dependence graph to improve the execution time of modulo schedulers in Section 5. We demonstrate that the search space of modulo schedules is bounded in Section 6. We compare the performance of this and other modulo schedulers for a benchmark of 629 loops in Section 7. Conclusions are presented in Section 8.

2 Register Requirements

In this section, we present the impact of scheduling on the register requirements of a modulo scheduled loop. The example target machine is a hypothetical processor with three fully-pipelined general-purpose functional units. The memory latency and the `add/sub` latency is one cycle, the `mult` latency is four cycles, and the `div` latency is eight cycles. We selected these values to obtain concise examples; however, our method works independently of the numbers of functional units, resource constraints, and latencies.

Example 1 This example [12] illustrates how to compute the register requirements of a modulo-scheduled loop. This kernel is: `y[i] = x[i]^2- x[i]- a`, where the value of `x[i]` is read from memory, squared, decremented by `x[i]+a`, and stored in `y[i]`, as shown in the dependence graph of Figure 1a.

The vertices of the dependence graph correspond to operations and the edges correspond to virtual registers. The value of each *virtual register* is defined by a unique opera-

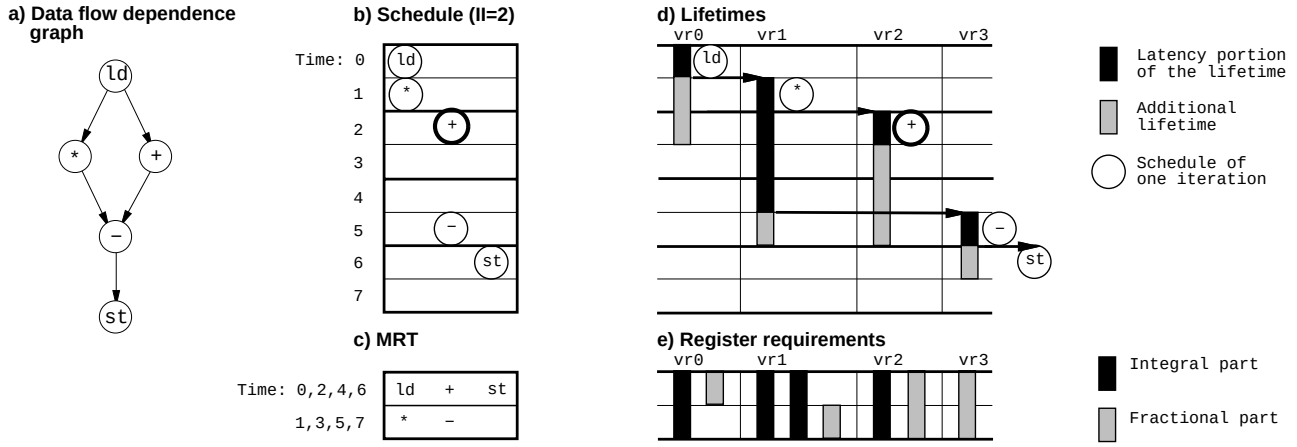

Figure 2: Modulo schedule for the kernel `y[i] = x[i]`^{2}`-x[i]-a` with minimum register requirements.

tion and once its value has been defined, it may be used by several operations. In this paper, a virtual register is reserved in the cycle where its define operation is scheduled, and remains reserved until it becomes free in the cycle following its last-use operation. In our machine model, the additional reserved time beyond the last-use operation cycle is one clock cycle. Other machines may have an additional reserved time which is operation dependent. The *lifetime* of a virtual register is the set of cycles during which it is reserved.

The scheduler places each operation of an iteration so that both resource constraints and dependence constraints are fulfilled. Figure 1b illustrates a schedule with an II of 2 for the kernel of Example 1 on the target machine. In this schedule, the `load`, `mult`, `add`, `sub` and `store` operations of the iteration starting at time 0 are respectively scheduled at time 0, 1, 1, 5, and 6.

The *modulo reservation table* (MRT) associated with a schedule is obtained by collapsing the schedule for an iteration to a table of II rows, using wraparound. Figure 1c illustrates the MRT associated with the schedule of Figure 1b. The resource constraints of a modulo schedule are satisfied if and only if the packing of the operations within the II rows of the MRT does not exceed the resources of the machine. For our target machine, the resource constraints allow up to 3 operations of any kind to be issued in each row of the MRT. The MRT can also model specialized resources by allowing only certain types of operations in each column.

The virtual register lifetimes associated with this iteration are presented in Figure 1d. The black bars correspond to the initial portion of the lifetime that minimally satisfies the latency of the defining functional unit; the grey bars show the additional lifetime of each virtual register. Ideally, the additional lifetime should not be longer than one cycle; but, because of resource constraints, it is not always possible to schedule each use operation immediately after its input operand latencies expire. The dependence constraints are satisfied if and only if no operation is placed during the portion of the lifetime that overlaps with the black bar of a virtual register that it uses.

Formal Problem Definition: Among all modulo schedules that employ the smallest feasible initiation interval and satisfy all resource constraints, find one that minimizes MaxLive.

Because of the modulo constraint, MaxLive can be quickly computed, as illustrated in Figure 1e. Figure 1e shows the number of live virtual registers in each of the II cycles of a steady-state loop execution and is constructed by replicating Figure 1d with a shift of II cycles between successive copies until the pattern in II successive rows repeats indefinitely. Figure 1e then displays these II rows in a compact form. Figure 1e can be constructed quickly by collapsing Figure 1d to II rows, with wraparound. In Figure 1e, we see that exactly six virtual registers are live in the first row and eight in the second, resulting in a MaxLive of eight.

Figure 1e distinguishes between two distinct contributions to each virtual register lifetime: the fractional and the integral part. For virtual registers with several uses, only the last use is considered. In our machine model, the additional reserved time is one clock cycle and is included, for simplicity, in the fractional part. The *fractional part* thus spans the rows in the MRT inclusively from the def operation row forward with wraparound through the last-use operation row. Its length ranges from 1 to II and is equal to 1 plus the modulo II distance between these rows, as shown in Figure 1e. The *integral part* spans the entire MRT exactly s times, where s corresponds to the number of times the row number of the last-use operation appears in the time interval starting inclusively from the def operation schedule time until, but not including, the last-use operation schedule time.

The *integral MaxLive* is defined as the number of integral bars in a row, summed over all virtual registers. Similarly, the *fractional MaxLive* is the number of fractional part bars in the row with the most fractional part bars. Finally, *MaxLive* is the sum of fractional and integral MaxLives.

The schedule presented in Figure 1b is a schedule that results in the minimum buffer requirements for this kernel and target machine, as presented by Govindarajan *et al* [11]. In their study, registers are approximated by conceptual FIFO buffers which are reserved for a time interval that is a multiple of II. It thus corresponds to searching for a schedule that minimizes integral MaxLive. However, minimizing integral MaxLive does not in general result in a schedule with minimal MaxLive.

For example, Figure 2 presents a schedule for the same kernel, resulting in the same integral MaxLive but resulting in a lower total MaxLive. The schedule of Figure 2b differs only by the schedule time of the `add` operation, which is delayed by one cycle from time 1 to time 2. The register

requirements associated with this schedule are shown in Figure 2e. Comparing the register requirements of this schedule to the one of Figure 1e, we see that both have an integral MaxLive of four. However, this schedule results in a lower fractional MaxLive as fractional parts are more evenly distributed among the II rows. As a result, MaxLive decreases from eight to seven.

3 Scheduling for Minimum MaxLive

In this section, we present an integer linear programming model that schedules the operations of an innermost loop while minimizing MaxLive. The definition of the schedule search space used in this section is similar to the one presented by Govindarajan *et al* [11]. All variables of the model presented in this section are summarized in Table 2.

We represent a loop by a dependence graph $G = \{V, E_{sched}, E_{reg}\}$, where the set of vertices V represents operations and the sets of edges E_{sched} and E_{reg} correspond, respectively, to the scheduling dependences and the register dependences among operations. A scheduling edge enforces a temporal relationship between dependent operations or between any operations that cannot be freely reordered, such as load and store operations to ambiguous memory locations. A scheduling edge from operation i to operation j, w iterations later, is associated with a latency $l_{i,j}$ and a dependence distance $\omega_{i,j} = w$. A register edge corresponds to a data flow dependence carried in a register. Initially (until Section 5), each register edge is also a scheduling edge. However, there are scheduling edges that have no corresponding register edge, e.g. the dependence between a store and a load to an ambiguous memory (or the same) location results in a scheduling edge but not in a register edge.

Consider a loop with N operations and an initiation interval of II. We represent a schedule for this loop by a $II \times N$ binary matrix, called A, where $a_{r,i} = 1$ if and only if operation i is scheduled in row r and 0 otherwise. Valid modulo schedules must satisfy two sets of conditions. First, each operation must be scheduled exactly once per iteration:

$$\sum_{r=0}^{II-1} a_{r,i} = 1 \qquad \forall i \in [0, N-1] \tag{1}$$

Second, the resource constraints of the target machine must be satisfied in each row of the MRT. Consider an operation i which consumes a resource q for one cycle exactly c cycles after being issued. A resource q is consumed in row r if operation i is scheduled c cycles earlier, namely if $a_{(r-c) \bmod II,i} = 1$. For a machine with M_q machine resources of type q, the resource constraint for row r and resource q simply states that no more than M_q resources can be consumed concurrently among all operations, namely

$$\sum_{i=0}^{N-1} \sum_{c \in Res_{i,q}} a_{(r-c) \bmod II,i} \leq M_q \qquad \forall q, \forall r \in [0, II-1] \tag{2}$$

where $c \in Res_{i,q}$ indicates that operation i uses the resource of type q for one cycle exactly c cycles after being issued.

We now introduce two definitions that characterize the row and the time at which each operation is scheduled:

row_i	row number in which operation i is scheduled (from 0 to $II-1$).
$time_i$	time at which operation i is scheduled

Using matrix A, we may write the row number in which operation i is scheduled as:

$$row_i = \sum_{r=1}^{II-1} r * a_{r,i} \tag{3}$$

A modulo schedule also defines the *stage* in which each operation is scheduled. We represent the stage number by k, an integer vector of dimension N, where k_i is the stage number in which operation i is scheduled. Matrix A and vector k uniquely define the cycle in which operations are scheduled. The schedule time of operation i is equal to:

$$time_i = k_i * II + row_i \tag{4}$$

A modulo schedule must enforce all the dependences of its dependence graph. A dependence between operation i and operation j, $\omega_{i,j}$ iterations later, is fulfilled if operation j is scheduled at least $l_{i,j}$ cycles after operation i:

$$(time_j + \omega_{i,j} * II) - time_i \geq l_{i,j} \qquad \forall (i,j) \in E_{sched} \tag{5}$$

Equations (1), (2), and (5) represent, respectively, the modulo constraints, the resource constraints, and the dependence constraints of a modulo schedule. These constraints define the space of valid modulo schedules for that initiation interval II, target machine, and dependence graph G with dependence distance ω and dependence latency l.

We must now define the register requirements associated with a modulo schedule. Since we are minimizing MaxLive, we must compute the precise number of live virtual registers in each of II consecutive cycles. We define a new $II \times N$ integer matrix, called V, where $v_{r,i}$ corresponds to the number of live virtual registers generated by operation i in row r. Matrix V is defined as follows:

$$\sum_{z=0}^{r} a_{z,i} - \sum_{z=0}^{r-1} a_{z,j} + (k_j + \omega_{i,j}) - k_i \leq v_{r,i}$$
$$\forall (i,j) \in E_{reg}, \qquad \forall r \in [0, II-1] \tag{6}$$

Equation (6) quantifies the register requirements in row r generated by the register dependence (i,j), from the def operation i to the use operation j.

To demonstrate the validity of Equation (6), we compute the fractional and integral parts of the lifetime associated with a register edge (i,j), namely:

$frac_{r,i,j}$	fractional part of the lifetime in row r associated with a register edge (i,j).
$int_{i,j}$	integral part of the lifetime associated with a register edge (i,j).

We will demonstrate that their sum is exactly equal to $v_{r,i}$ if operation j is the last-use of the operation i result. We introduce three definitions specific to def operation i and use operation j:

$$d_r = \sum_{z=0}^{r} a_{z,i} \qquad u_r = \sum_{z=0}^{r-1} a_{z,j}$$
$$x = \begin{cases} 1 & \text{if } row_i > row_j \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Recall that only one $a_{z,i}$ is non zero for a given operation i among all rows. As a result, both d_r and u_r are non-decreasing functions, with minimal value 0 and maximal value 1. As defined in Equation (7), d_r becomes 1 exactly in the row z where def operation i is scheduled ($a_{z,i} = 1$). Similarly, u_r becomes 1 exactly in the row z following the

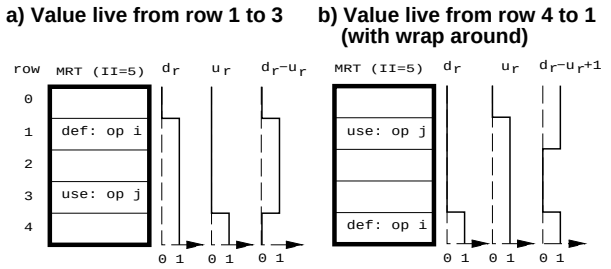

Figure 3: Determining the fractional and integral part of a lifetime.

row where use operation j is scheduled ($a_{z-1,j} = 1$). Once d_r or u_r become 1, they remain 1 through row $II - 1$.

When computing the fractional part of the lifetime associated with the def operation i and the use operation j, two cases must be considered. In the first case, row_i is no greater than row_j. Therefore, the initial value of the term $d_r - u_r$ is initially 0, becomes 1 in the row of the def operation, and returns to 0 in the row after the row of the use operation. As a result, the term $d_r - u_r$ is 1 exactly where the fractional part of the lifetime associated with (i, j) would contribute to the register requirements if operation j is the last use; otherwise, $d_r - u_r$ is 0. In Figure 3a for example, the def operation i is scheduled in row 1 ($a_{1,i} = 1$) and the use operation j is scheduled in row 3 ($a_{3,j} = 1$). The plots of d_r, u_r, and $d_r - u_r$ are also shown in Figure 3a. Notice that the fractional part of this lifetime is correctly determined to be live in rows 1, 2 and 3.

In the second case, row_i is larger than row_j. The initial value of the term $d_r - u_r + 1$ is 1, becomes 0 in the row following the row of the use operation, and returns to 1 in the row of the def operation. As a result, the term $d_r - u_r + 1$ is 1 exactly where the fractional part of the lifetime associated with (i, j) would contribute to the register requirements if operation j is the last use; otherwise, $d_r - u_r + 1$ is 0. In Figure 3b for example, the def operation i is scheduled in row 4 ($a_{4,i} = 1$) and the use operation j is scheduled in row 1 ($a_{1,j} = 1$). The plots of d_r, u_r, and $d_r - u_r + 1$ are shown in Figure 3b. We see that the fractional part of this lifetime is correctly determined to be live in rows 4, 0, and 1.

Using the definitions of Equation (7), we can express the fractional part of the lifetime associated with operations i and j in row r as:

$$frac_{r,i,j} = d_r - u_r + x \tag{8}$$

where $frac_{r,i,j}$ is 1 if and only if row r contributes to the fractional part of the lifetime associated with operations i and j.

To compute the integral part of the lifetime associated with operations i and j, we must also consider two cases. In the first case, row_i is no greater than row_j, and the integer part corresponds directly to the difference between the stage numbers of the use and the def operations, namely $(k_j + w_{i,j}) - k_i$. In Figure 3a, the integral part is 0 when $k_i = k_j$ and $\omega_{i,j} = 0$. If the use operation was delayed by II cycles, k_j would increase by 1 and so would the integral part.

In the second case, row_i is larger than row_j, and the integer part corresponds directly to $(k_j + w_{i,j}) - k_i - 1$, where the -1 term is introduced to counterbalance the wraparound effect. In Figure 3b, the integral part is 0 when $k_i = k_j + 1$ and $\omega_{i,j} = 0$. Again, delaying the use operation by II cycles would increase the integral part by 1.

As a result, we can express the integral part of the lifetime associated with operations i and j as:

$$int_{i,j} = k_j + w_{i,j} - k_i - x \tag{9}$$

By summing the fractional and integral parts of the lifetime, as defined in Equations (8) and (9), the x terms cancel each other, resulting in the left hand side of Equation (6). Finally, the register requirement contribution of an operation i in row r corresponds to the maximum register requirements among all uses of operation i.

Using the previous result, MaxLive is defined as the sum of the fractional and integral part of each virtual register in the row with the largest fractional plus integral part:

$$\sum_{i=0}^{N-1} v_{r,i} \quad \leq \quad MaxLive \qquad \forall r \in [0, II-1] \tag{10}$$

This completes the description of the objective function and we can now present the algorithm that will schedule an iteration for maximal throughput and minimum register requirements.

4 Optimum Modulo Schedules

We present in this section an algorithm that finds a schedule with the highest steady state throughput over all modulo schedules and the minimum register requirements among such schedules. The model, from Equation (1), (2), (5), and (6), and its variables are respectively summarized in Figure 4 and Table 2. Note that the model finds a schedule with minimum MaxLive for a given initiation interval. Therefore, a schedule with minimum register requirements for the smallest feasible initiation interval is obtained by solving a series of problems, starting with the minimum initiation interval.

Algorithm 1 (MinReg Modulo Scheduler) *Consider a machine description M_q, $Res_{i,q}$, and $l_{i,j}$ and consider a data dependence graph $G = \{V, E_{sched}, E_{reg}\}$ and $\omega_{i,j}$. The following algorithm finds a schedule with the highest steady state throughput over all modulo schedules, and the minimum register requirements among such schedules:*

1. *Compute the Minimum Initiation Interval (MII) [3] and set the tentative II to MII.*
2. *Build the integer linear programming system as indicated in Figure 4 and search for a solution.*
3. *If the system built in Step 2 fails to find a feasible solution, increase II by one and return to Step 2. Otherwise, an solution found is optimal.*

Minimize	Variables	Fundamental Constraints
Int MaxLive	$N(II+2)$	$N + m * II + e_{reg} + e_{sched}$
Tot MaxLive	$N(2II+1)$	$N + m * II + (e_{reg}+1)II + e_{sched}$
Increase	$N * (II-1)$	$(e_{reg}+1)II - e_{reg}$

Table 1: Minimizing integral versus total MaxLive.

For comparison purposes, we also investigated a model minimizing integral MaxLive. This integer programming model differs from the model of Figure 4 in that the equations labeled Lifetimes and MaxLive are replaced by an equation defining the number of FIFO buffers needed to

Minimize:	$MaxLive$	
Subject to:		
MRT:	$\sum_{r=0}^{II-1} a_{r,i} = 1$	$\forall i \in [0, N-1]$
Resources:	$\sum_{i=0}^{N-1} \sum_{c \in Res_{i,q}} a_{(r-c) \bmod II,i} \leq M_q$	$\forall q\,, \forall r \in [0, II-1]$
Dependences:	$time_j + \omega_{i,j} * II - time_i \geq l_{i,j}$	$\forall (i,j) \in E_{sched}$
Lifetimes:	$\sum_{z=0}^{r} a_{z,i} - \sum_{z=0}^{r-1} a_{z,j} + k_j + \omega_{i,j} - k_i \leq v_{r,i}$	$\forall r \in [0, II-1], \forall (i,j) \in E_{reg}$
MaxLive:	$\sum_{i=0}^{N-1} v_{r,i} \leq MaxLive$	$\forall r \in [0, II-1]$
Definitions:	$time_i = \sum_{r=1}^{II-1} r * a_{r,i} + k_i * II$	
	$MaxLive,\ a_{r,i},\ v_{r,i},\ k_i$: integer, ≥ 0	

Figure 4: Scheduling for minimum MaxLive

Machine:	(input)	Schedule:	(input)
$l_{i,j}$	latency between operation i and operation j.	II	initiation interval (constant in the integer programming model).
M_q	number of resources of type q.		
$Res_{i,q}$	set of cycles in which operation i uses resource q.		
Graph:	(input)	Results:	(output)
$G = \{V, E_{sched}, E_{reg}\}$	dependence graph with scheduling edge and register edge set.	$a_{r,i}$	binary matrix where $a_{r,i} = 1$ if operation i is scheduled in row r.
N	number of operations in graph G.	k_i	stage number of operation i.
$\omega_{i,j}$	dependence distance between operation i and operation j (in iteration).	$v_{r,i}$	number of live virtual registers in row r generated by operation i.

Table 2: Variables for the modulo scheduling model.

store the result of operation i, as proposed by Govindarajan *et al* [11]. Algorithm 1 is directly applicable to this integer programming model as well.

The complexity of these models, in number of variables and fundamental constraints, is shown in Table 1 for a loop with N operations, an initiation interval of II, e_{reg} register dependence edges, e_{sched} scheduling dependence edges, and a machine with m kinds of machine resources. The major complexity increase, when minimizing MaxLive instead of integral MaxLive, is that $N * II$ variables are needed to keep track of the register requirements of each operation in each row of the MRT instead of N variables for each operation over the entire MRT. We will subsequently refer to a modulo scheduler minimizing total MaxLive as *MinReg Scheduler* and a modulo scheduler minimizing integral MaxLive as *MinBuf Scheduler*.

5 Removing Redundant Edges

The dependence graph, as defined in Section 3, may contain edges that result in redundant constraints. Removing these redundant constraints is important because while they do not restrict the search space, they consume a significant amount of time to verify them. In this section, we present a technique that removes redundant edges, i.e. those whose removal does not expand the space of modulo schedules. This technique is applicable to any modulo scheduler.

Typically, it is known that transitive scheduling edges of a single basic block can be ignored when scheduling that basic block. In this section, we extend this result in two directions. First, we handle dependence graphs with arbitrary dependence distances. Second, we remove redundant scheduling edges and redundant register edges. Removing redundant register edges is particularly important since excluding one register edge results in II fewer redundant constraints, as indicated in Table 1.

Formally, we remove redundant edges of the original graph $G = \{V, E_{sched}, E_{reg}\}$ to get a new graph $G' = \{V, E'_{sched}, E'_{reg}\}$, with fewer redundant edges. Edges are removed from G' if and only if we can prove that scheduling for G or G' is guaranteed to result in defining the same scheduling search space with the same register requirements. While E_{reg} was defined as a subset of E_{sched}, this inclusion property does not hold in G', as some edges may be removed only from E'_{reg} or E'_{sched}.

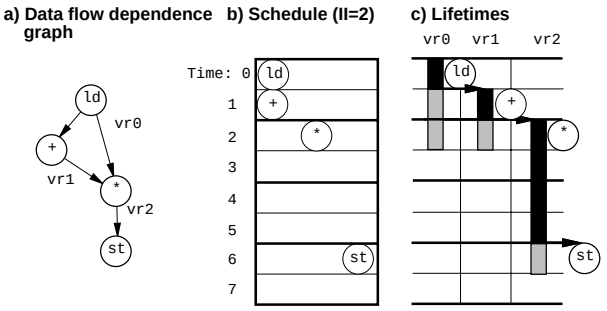

Figure 5: Modulo schedule for Example 2.

Example 2 This example illustrates a case where two redundant edges can safely be removed from G'. This kernel is: `y[i] = x[i] *(x[i] +a)` as shown in the dependence graph

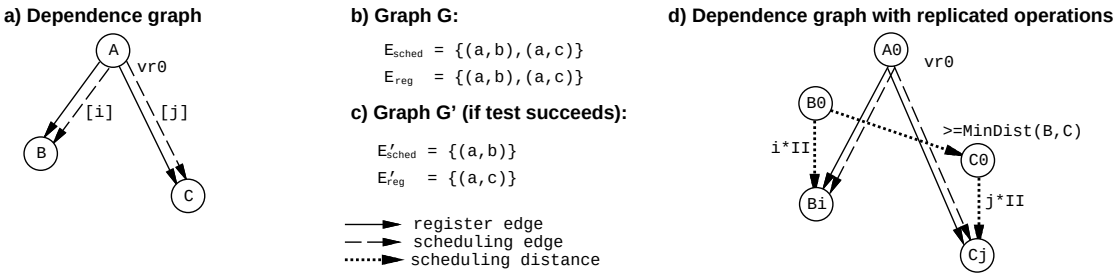

Figure 6: Removing edges.

of Figure 5a. This example is a basic block with no loop-carried dependence. In this example, the `mult` operation is guaranteed to be scheduled no earlier than one cycle after the `add` operation. As a result, the scheduling edge (`load`, `mult`) can be safely removed from E'_{sched} because enforcing the dependence constraints of scheduling edges (`load`, `add`) and (`add`, `mult`) is necessarily more constraining than enforcing the dependence constraints of scheduling edge (`load`, `mult`). Similarly, the register edge (`load`, `add`) can be removed from E'_{reg} since the `mult` is the last-use operation of `vr0` which implies that the register lifetime generated by the (`load`, `add`) edge is necessarily included in the lifetime generated by the register edge (`load`, `mult`).

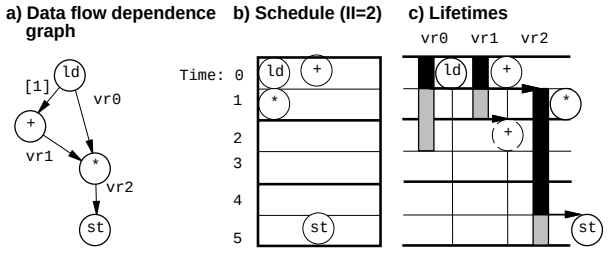

Figure 7: Modulo schedule for Example 3.

Example 3 This example shows that the notion of transitive edges cannot be extended directly to dependence graphs with loop-carried dependences. This kernel is: `y[i] = x[i] *(x[i-1] +a)` as shown in the data flow dependence graph of Figure 7a. This example differs from Example 2 only by the dependence of distance 1 associated with the edge (`load`, `add`). Figure 7b presents a schedule for this kernel where the `add` operation is scheduled at time 0, a valid schedule since the `add` operation input is generated by the `load` operation of the previous iteration. Figure 7c illustrates the lifetimes associated with this schedule. Since the result of the `load` operation is used both by the `mult` operation (time 1) and the `add` operation of the next iteration, (time $0 + II = 2$), the lifetime of $vr0$ lasts through time 2. This schedule shows that the lifetime associated with the register edge (`load`, `add`) is not necessarily included in the lifetime associated with the register edge (`load`, `mult`), even in presence of a scheduling edge from the `add` operation to the `mult` operation. As a result, the register edge (`load`, `add`) cannot be removed from G'.

To determine the conditions under which edges can be safely removed, we introduce the $MinDist$ relation as proposed by Huff [15]. $MinDist(x, y)$ represents the minimum number of cycles, possibly negative, that must occur between the time operation X is scheduled until the time operation Y is scheduled, if there is a path in the dependence graph from X to Y. Otherwise, $MinDist(x, y)$ is defined as minus infinity. Computing $MinDist$ is an all-pairs longest-path problem where each dependence edge (i, j) is assigned a cost of $l_{i,j} - \omega_{i,j} * II$, the minimum schedule distance as defined in Equation (5). $MinDist$ is guaranteed to converge to a solution because II is chosen such that all cycles in the dependence graph have nonpositive costs [15].

Theorem 1 (Edge Removal Test) *Consider the operations A, B, and C in V with edges (a, b) and (a, c) in E_{sched} and E_{reg} of G. When the following inequality holds:*

$$(\omega_{a,c} - \omega_{a,b}) * II + MinDist(b, c) \quad \geq \quad 0$$

we can guarantee that if the edge (a, b) is a scheduling edge in graph G', the edge (a, c) can be removed from the scheduling edge set of graph G'. Similarly, we can guarantee that if the edge (a, c) is a register edge in graph G', the edge (a, b) can be removed from the register edge set of graph G'.

Figure 6a illustrates Theorem 1. In this partial dependence graph, the value produced by operation A is used i iterations later by operation B ($\omega_{a,b} = i$) and j iterations later by operation C ($\omega_{a,c} = j$). The sets of edges associated with this dependence graph are presented in Figure 6b. When the inequality of Theorem 1 holds, the sets of edges shown in Figure 6c are guaranteed to result in defining the same scheduling search space with the same register requirements.

Proof: Consider the dependence graph with replicated operations shown in Figure 6d, which corresponds to the same dependence graph as illustrated in Figure 6a. In this graph, operations $A0$, $B0$, and $C0$ are operations of the first iteration, operation Bi is an operation of iteration i, and operation Cj is an operation of iteration j. As defined in Figure 6a, the result of operation $A0$ is used by operations Bi and Cj.

Several scheduling distances can be inferred from Figure 6d. First, the modulo constraint guarantees that operation Bi is scheduled exactly $i * II$ cycles after $B0$. Similarly, operation Cj is guaranteed to be scheduled $j * II$ cycles after $C0$. We can also use the minimum distance relation to assert that operation $C0$ must be scheduled at least $MinDist(b, c)$ cycles after operation $B0$.

We now demonstrate the conditions that must hold to guarantee that operation Cj is scheduled no earlier than operation Bi. Without loss of generality, we state that operation Bi is scheduled at time $t_{B0} + i * II$. Summing the scheduling distance from operation $B0$ to Cj, we can guarantee that operation Cj can be scheduled no earlier than $t_{B0} + MinDist(b, c) + j * II$. Using these results, we conclude that operation Cj is scheduled no earlier than opera-

tion Bi if the difference of their schedule times is nonnegative, namely if $(t_{B0}+MinDist(b,c)+j*II)-(t_{B0}+i*II) \geq 0$. Simplifying the t_{B0} terms and substituting $\omega_{a,b}$ and $\omega_{a,c}$ for i and j, respectively, we obtain the inequality of Theorem 1. Note that this inequality is independent of t_{B0}, the time at which operation $B0$ is scheduled.

When operation Cj is guaranteed by the edge-removal test to be scheduled no earlier than operation Bi, we know that the scheduling edge (a,c) is guaranteed to be fulfilled and can therefore be removed. Similarly, we know that the register requirements of edge (a,b) are satisfied by register edge (a,c). □

Algorithm 2 (Edge Removal) *For a given dependence graph $G = \{V, E_{sched}, E_{reg}\}$ and initiation interval II, we construct an equivalent graph $G' = \{V, E'_{sched}, E'_{reg}\}$ that is guaranteed to result in defining the same scheduling search space with the same register requirements and fewer edges as follows:*

1. *Initialize $G' = G$ and compute the $MinDist$ matrix for the given II.*
2. *Apply Theorem 1 to each pair of outgoing edges of a vertex v, removing redundant edges in both E'_{sched} and E'_{reg} when feasible.*
3. *Repeat Step 2 for each vertex of the graph.*

6 Bounding the Schedule Space

As presented in Figure 4, the integer linear programming search space is unbounded because there is no limit on the k_i values and hence on the maximal schedule length. To bound the schedule length of an iteration and thereby significantly improve the efficiency of the integer programming solver, we introduce two pseudo operations in the dependence graph: `start` is a predecessor of every operation in the graph, and `stop` is a successor to every operation in the graph. The latency and dependence distance of the outgoing edges from `start` and of the incoming edges to `stop` are 0. Using the methodology presented by Huff [15], the $MinDist$ relation can be used to determine a lower bound and an upper bound on the schedule time of each operation. We know that an operation i cannot be scheduled earlier than $MinDist(start, i)$ cycles after the `start` pseudo operation. Similarly, we know that an operation i cannot be scheduled earlier than $MinDist(i, stop)$ cycles before the `stop` pseudo operation.

To bound the search space of the integer linear programming solver, we must relate the schedule time of these two pseudo operations to one another. Without loss of generality, we define the maximal schedule length of a schedule to be $MinDist(start, stop) + S$ where S is a nonnegative integer value. Using this result, we bound the schedule time of operation i as follows:

$$MinDist(start, i) \leq time_i$$

$$time_i \leq MinDist(start, stop) + S - MinDist(i, stop) \quad (11)$$

where S represents the *schedule slack* or degree of freedom given to the scheduler.

Theorem 2 *Let R_S be the minimum MaxLive for a dependence graph G and initiation interval II among all schedules of schedule length no greater than $MinDist(start, stop)+S$. The minimum MaxLive values are bounded as follows:*

$$R_0 \geq R_1 \geq \ldots \geq R_{y-1} \geq R_y = R_{y+1} = \ldots = R_\infty, \text{ where}$$

$$y = (N-1) * (l_{max} + II - 1) + 1 - MinDist(start, stop)$$

where l_{max} corresponds to the longest minimal schedule distance among pairs of adjacent operations.

Proof: Since the set of schedules of length no greater than $MinDist(start, stop) + S + 1$ necessarily contains all the schedules of length no greater than $MinDist(start, stop) + S$, we can guarantee that the relation on the minimum MaxLive $R_S \geq R_{S+1}$ holds for any schedule slack S.

Consider now a schedule of length $z > (N-1) * (l_{max} + II - 1) + 1$. We can guarantee that there is a shorter schedule of length $z - II$ that still fulfills both resource and dependence constraints and results in lower register requirements. The proof is based on the pigeonhole principle.

Consider the operations scheduled at time t_i and select the operations scheduled at time $t_j > t_i$ such that there are no operations scheduled in the time interval (t_i, t_j). Since there are N vertices, there are at most $N-1$ such intervals. Because the length of the schedule, z, is larger than $(N-1) * (l_{max} + II - 1) + 1$, there must be at least one interval of length larger than $l_{max} + II - 1$. Consider (t_i, t_j) to be such an interval. In this case, we can trivially reschedule all operations scheduled strictly after t_i, II cycles earlier without violating any dependence constraints. Moreover, rescheduling operations II cycles earlier is guaranteed to fulfill the resource constraints as the MRT is unchanged by the new schedule. Consequently, interval (t_i, t_j) is reduced by II. By finite induction, we can get $z \leq (N-1) * (l_{max} + II - 1) + 1$.

Furthermore, reducing the schedule length by II is guaranteed to result in lower register requirements because it reduces all the lifetimes that span the interval (t_i, t_j) by II. Therefore, limiting the schedule length to $(N-1) * (l_{max} + II - 1) + 1$, i.e. limiting the scheduling slack to $y = (N-1) * (l_{max} + II - 1) + 1 - MinDist(start, stop)$, does not preclude any optimal solutions. □

7 Measurements

In this section, we investigate the register requirements of four modulo scheduling algorithms. For the purpose of comparison, we also present the register requirements of an efficient modulo scheduling algorithm that presently does not attempt to minimize the register requirements. Unfortunately, we are unable to provide a comparison with Huff's scheduling algorithm [15] since his machine model differs slightly from ours and his latest scheduler, presented in [15], was not available to us.

MinReg Scheduler: This scheduler minimizes MaxLive over all valid modulo schedules using the integer programming model developed in Section 3 and summarized in Figure 4. The resulting schedule has the lowest register requirements of any modulo schedule for the given machine, loop, and initiation interval.

MinBuf Scheduler: This scheduler minimizes Integral MaxLive over all valid modulo schedules using an integer programming model similar to the one proposed by Govindarajan [11]. The resulting schedule has the lowest buffer requirements of any modulo schedule for the given machine, loop, and initiation interval. Buffers, as opposed to registers, must be reserved for a time interval that is a multiple of II. We present here the register requirements associated with these schedules in our comparisons.

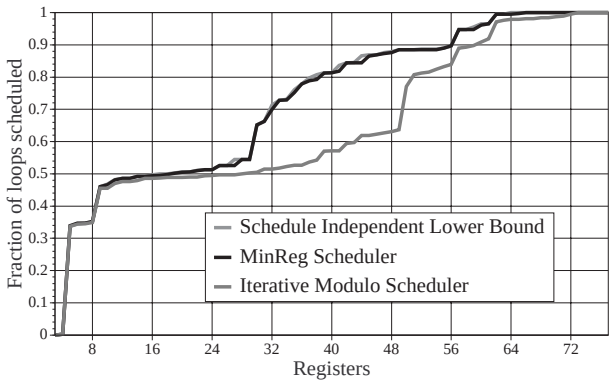

Figure 8: MaxLive.

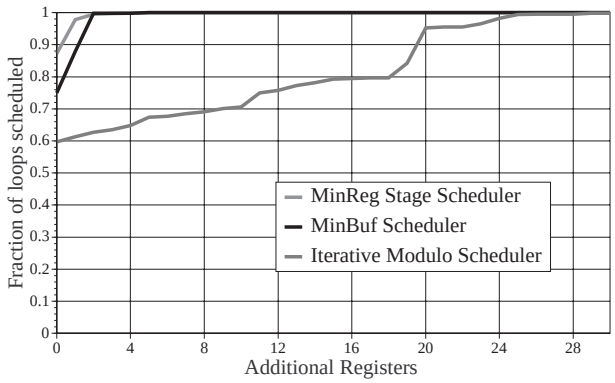

Figure 9: MaxLive increase over the MinReg Scheduler.

MinReg Stage Scheduler[12]: This scheduler minimizes MaxLive over all valid modulo schedules that share a given MRT. This scheduler can only select the stage in which each operation is scheduled[1]; however, it can generate a schedule in linear time for loops with acyclic underlying dependence graphs and in polynomial time otherwise. The resulting schedule has the lowest achievable register requirements for the given machine, loop, and MRT.

Iterative Modulo Scheduler[3]: This scheduler has been designed to deal efficiently with realistic machine models while producing schedules with near optimal steady state throughput. Experimental findings show that this algorithm requires the scheduling of only 59% more operations than does acyclic list scheduling while resulting in schedules that are optimal in II for 96% of loops in their benchmark [3]. In its current form, it does not attempt to minimize the register requirements of its schedules; however, preliminary investigations show that the register requirements of its schedules may be reduced significantly by simple heuristics.

In this section, we use a benchmark of loops obtained from the Perfect Club [16], SPEC-89 [17], and the Livermore Fortran Kernels [18]. Our benchmark consists exclusively of innermost loops with no early exits, no procedure calls, and fewer than 30 basic blocks, as compiled by the Cydra 5 Fortran77 compiler [19]. The input to the four scheduling algorithms consists of the Fortran77 compiler intermediate representation after load-store elimination, recurrence back-substitution, and IF-conversion. Of the 1327 loops successfully modulo scheduled by the Cydra 5 Fortran77 compiler, we selected all the loops with up to 12 operations and with II up to 5, resulting in a benchmark of 629 loops. The cutoff limit was selected such that all scheduling algorithms ran to completion in a reasonable amount of time[2].

The machine model used in these experiments corresponds to the Cydra 5 machine. This choice was motivated by the availability of quality code for this machine. Also, the resource requirements of the Cydra 5 machine are complex [10], thus stressing the importance of good and robust scheduling algorithms. In particular, the machine configuration is the one used in [3].

Redundant edges were removed from the dependence graph of each loop to reduce the execution time of the modulo schedulers. For the MinReg Scheduler and the MinBuf Scheduler, the scheduling slack presented in Theorem 2 was used to bound the search space. The Iterative Modulo Scheduler was used to generate the required MRT input for the MinReg Stage Scheduler. All scheduling algorithms used the same initiation interval as the one generated by the Iterative Modulo Scheduler, which turned out to be optimal for all the 629 loops of the benchmark.

We first investigated the register requirements of the MinReg Scheduler which results in the lowest register requirements over all valid modulo schedules. Figure 8 presents the fraction of the loops in our benchmark that can be scheduled for a machine with any given number of registers without spilling and without increasing II. In this graph, the X-axis represents MaxLive and the Y-axis represents the fraction of loops scheduled. This graph presents three curves. The first curve, labeled "Schedule independent lower bound," corresponds to [15] and is representative of the register requirements of a machine without any resource conflicts. The second curve presents the register requirements of the schedules generated by the MinReg Scheduler. The fact that these two first curves are nearly identical indicates that the complex resource requirements of the Cydra 5 machine does not significantly impact the register requirements of the optimal schedules for the benchmark considered. The third curve presents the register requirements of the Iterative Modulo Scheduler, which presently does not attempt to minimize the register requirements. The gap between the two last curves is indicative of the degree of improvement that is possible with register-sensitive modulo scheduling algorithms.

The next graph illustrates the number of additional registers needed when the other scheduling algorithms are used instead of the MinReg Scheduler. The X-axis of Figure 9 represents the number of additional registers and the Y-axis represents the fraction of loops that can be scheduled with this number. The curves represent the additional register requirements for the MinReg Stage Scheduler, the MinBuf Scheduler, and the Iterative Modulo Scheduler. It is interesting to note that in 89% of the loops, the MinReg Stage Scheduler finds a schedule with no additional register requirements, even though it can only choose the stage for each operation. The MinBuf Scheduler finds an optimal schedule in 75% of the loops. This result implies that in 25% of the loops, minimizing Integral MaxLive does not result in a schedule with minimum MaxLive. This result also implies that for this benchmark, MinReg Stage Scheduler performs better than MinBuf Scheduler. In other words, minimizing MaxLive for the MRT selected by the Iterative Modulo

[1] In the notation presented in Section 3, this corresponds to minimizing the register requirements for a search space with a fixed matrix A but variable stage vector k.

[2] The cumulative time to solution for the integer programming solver was 11 minutes and 30 minutes, respectively, for the MinBuf Scheduler and the MinReg Scheduler.

Scheduler results on average in lower register requirements than minimizing integral MaxLive among all MRTs. Finally, the Iterative Modulo Scheduler results in a schedule with minimum register requirements in 60% of the loops, a surprisingly high percentage for a heuristic that does not attempt to minimize the lifetimes. However, the overall performance of the Iterative Modulo Scheduler is severely degraded by a few loops that require up to 31 additional registers.

8 Conclusion

Modulo scheduling is an efficient technique for exploiting instruction level parallelism in a variety of loops. It results in high performance code, but increases the register requirements. As the trend toward higher concurrency continues, whether due to using and exploiting machines with faster clocks and deeper pipelines, wider issue, or a combination of both, the register requirements will increase even more. As a result, scheduling algorithms that reduce register pressure while scheduling for high throughput are increasingly important.

This paper presents an approach that schedules the loop operations to achieve the minimum register requirements for the smallest initiation interval. We compare the performance of this and other modulo schedulers for a benchmark of 629 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels for the Cydra 5 machine, a machine with complex resource usage. Our empirical findings show that the complex resource requirements of the Cydra 5 machine does not significantly impact the register requirements of a modulo schedule with minimal MaxLive in the benchmark considered. Also, our findings show that the Iterative Modulo Scheduler [3] which does not attempt to minimize registers nevertheless achieves the minimum MaxLive in 60% of the loops and uses an average of 14.03 additional registers over the remaining loops. The MinBuf Scheduler [11] finds a schedule with minimum MaxLive in 75% of the loops and uses an average of 1.52 additional registers over the remaining loops. The MinReg Stage Scheduler [12] finds a schedule with minimum MaxLive in 89% of the loops and uses an average of 1.45 additional registers over the remaining loops.

This paper demonstrates that selecting a good schedule results in lower register requirements. Though the optimal algorithm may be too expensive for general use in a compiler it may be extremely useful in special situations. This algorithm is also useful in evaluating the performance of register-sensitive modulo scheduling heuristics.

Acknowledgements

This work was supported in part by the Office of Naval Research under grant number N00014-93-1-0163 and by Hewlett-Packard. The authors would like to thank Bob Rau for his many useful suggestions and for providing the input data set. We are grateful to Vinod Kathail for his explanation of the resource constraints of the Cydra 5 machine.

References

[1] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. In *The Journal of Supercomputing*, volume 7, pages 9–50, Boston, July 1993. Kluwer Academic Publishers.

[2] P. Y. Hsu. *Highly Concurrent Scalar Processing*. PhD thesis, University of Illinois at Urbana-Champaign, 1986.

[3] B. R. Rau. Iterative Modulo Scheduling: An algorithm for software pipelining loops. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[4] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[5] N. J. Warter, G. E. Haab, and J. W. Bockhaus. Enhanced Modulo Scheduling for loops with conditional branches. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[6] M. S. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 40–51, November 1994.

[7] P. P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of loops with exits on pipelined architectures. *Proceedings of Supercomputing '90*, pages 200–212, November 1990.

[8] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[9] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register requirements of pipelined processors. *Proceedings of the International Conference on Supercomputing*, pages 260–271, July 1992.

[10] G. R. Beck, D. W. L. Yen, and T. L. Anderson. The Cydra 5 mini-supercomputer: Architecture and implementation. *The Journal of Supercomputing*, 7:143–180, May 1993.

[11] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 85–94, November 1994.

[12] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 75–84, November 1994.

[13] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, 1993.

[14] J. Ramanujam. Optimal software pipelining of nested loops. *Proceedings of the International Parallel Processing Symposium*, pages 335–342, June 1994.

[15] R. A. Huff. Lifetime-sensitive modulo scheduling. *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 258–267, June 1993.

[16] M. Berry et al. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[17] J. Uniejewski. SPEC Benchmark Suite: Designed for today's advanced system. *SPEC Newsletter*, Fall 1989.

[18] F. H. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, California, 1986.

[19] J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7:181–227, May 1993.