# Efficient Formulation for Optimal Modulo Schedulers

Alexandre E. Eichenberger

ECE Department
North Carolina State University
Raleigh, NC 27695-7911
alexe@eos.ncsu.edu

Edward S. Davidson

EECS Department
University of Michigan
Ann Arbor, MI 48109-2122
davidson@eecs.umich.edu

**Abstract**

Modulo scheduling algorithms based on optimal solvers have been proposed to investigate and tune the performance of modulo scheduling heuristics. While recent advances have broadened the scope for which the optimal approach is applicable, this approach increasingly suffers from large execution times. In this paper, we propose a more efficient formulation of the modulo scheduling space that significantly decreases the execution time of solvers based on integer linear programs. For example, the total execution time is reduced by a factor of 8.6 when 782 loops from the Perfect Club, SPEC, and Livermore Fortran Kernels are scheduled for minimum register requirements using the more efficient formulation instead of the traditional formulation. Experimental evidence further indicates that significantly larger loops can be scheduled under realistic machine constraints.

## 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. Modulo scheduling [1][2][3] is a technique that exploits the instruction level parallelism present among the iterations of a loop by overlapping the execution of consecutive loop iterations. 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.

In order to efficiently exploit the available instruction level parallelism, modulo scheduling algorithms must take into account the constraints of the target processor, such as the latencies of the operations, the number of available resources, and the size of the register files. In addition to satisfying the above machine constraints, several potentially conflicting objectives are typically considered, such as minimizing the initiation interval of the modulo schedule, minimizing the schedule length of a loop iteration, and minimizing the register requirements of the resulting modulo schedule.

Because of the potentially conflicting nature of these above objectives, and to investigate the best feasible schedules for a given loop iteration and set of machine constraints, modulo scheduling algorithms based on optimal solvers have been proposed. These algorithms, referred to as *optimal modulo schedulers* are only optimal with respect to their objective functions. Examples of objective functions found in the literature are "minimum $II$," "minimum schedule length among all minimum-$II$ modulo schedules," and "minimum register requirements among all minimum-$II$ modulo schedules." Note also that the code is generally assumed to be optimized (including transformations such as load-store elimination, strength reduction, and loop unrolling) and no further code transformations are performed while scheduling.

Recent advances in optimal modulo scheduling algorithms have broadened the scope of the machines for which this approach is applicable: for example, machines with arbitrary patterns of resource usages can be handled using the formulation proposed in [4] or [5]. The formulation in [5] may also map each operation to

To appear at the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas, Nevada, 15-18 June, 1997.

a given instance of a resource (e.g. map a multiply operation to one of the multiply functional units). Recent advances have also refined the secondary objectives that may be minimized; for example, the actual register requirements of a modulo schedule (i.e. the maximum number of live variables at any cycle of the schedule, referred to as MaxLive) may be minimized using the formulation proposed in [4].

While the optimal modulo scheduling approach may be further extended to, for example, architectures with clusters of functional units or scheduling models that integrate spill code generation or loop transformations, such extensions will only be practical if loops of reasonable size may be solved in a reasonable amount of time. Unfortunately, the present state-of-the-art optimal modulo schedulers based on integer linear programming solvers has not yet reached this stage, when accounting for all the machine constraints, even when scheduling for traditional machines and without considering spill code generation or loop transformations. As recently illustrated by Ruttenberg *et al* [6], a modulo scheduler that minimizes the register requirements among all minimum-$II$ modulo schedules has difficulty finding solutions in reasonable time for medium-sized loops when precisely modeling the machine constraints. However, they show that if simplifying assumptions are made (by minimizing an approximation of the register requirements and by ignoring the memory bank contentions present on the actual machine), the resulting schedules, although optimal under the simplifying assumptions, are frequently slower on the actual machine and have often higher register requirements than the schedules obtained by a carefully tuned heuristic. As a result, we may conclude from their study that modulo schedulers based on optimal solvers provide useful insight only if they are both efficient, to solve problems of reasonable size, and precise, to find solutions that are relevant to the actual machine.

In this paper, we address this major concern by proposing a more efficient formulation of the modulo scheduling space, which can be used by *integer linear programming solvers* (IP solvers) to significantly decrease the computation time required to find an optimal schedule under realistic machine constraints. Compared to the traditional formulation of the modulo scheduling space used in previous work [4][5][6][7], this novel formulation decreases the number of branch-and-bound nodes visited by the IP solver by two orders of magnitude, on average, when searching for schedules with minimum register requirements among all minimum-$II$ modulo schedules, for the 782 loops (out of 1327 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels) that were successfully scheduled by both the traditional and the proposed formulations. This decrease results, in turn, in a reduction of the total computation time of the solver by a factor of 8.6, from 870.2 to 101.0 seconds.

Furthermore, this more efficient formulation enables us to increase the number of loops successfully scheduled for minimum register requirements among all minimum-$II$ modulo schedules from 782 to 917 loops. When searching for a minimum $II$ modulo schedule only, up to 1179 or 88.8% of the 1327 loops can be scheduled in reasonable time (which was set to 15 minutes per loop in our experiments). Note also that these results are all obtained when scheduling for the Cydra 5 machine, a machine with complex resource requirements.

Using the proposed technique in conjunction with a commercial integer linear programming solver, one should be able to build in a few days an optimal modulo scheduler, which can be used to evaluate and fine tune the performance of modulo scheduling heuristics. In this paper, we use our formulation to evaluate the performance of the schedules produced by the Iterative Modulo Scheduler [3][8] as well as the register requirements of the schedules produced by the Stage Scheduling heuristics [9][10] used in conjunction with Iterative Modulo Scheduler.

In this paper, we present the background concepts and the traditional formulation of the modulo scheduling space in Sections 2 and 3, respectively. We derive a more efficient formulation of the dependence constraints in Section 4. We evaluate the benefits of our technique in Section 5 and conclude in Section 6.

## 2 Backgrounds on Modulo Scheduling

In this section, we present the background concepts used in modulo scheduling. 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, and the `mult` latency is four cycles. We selected these values to obtain a concise example; however, our method works independently of the numbers of functional units, resource constraints, and latencies.

**Example 1** This example [11] illustrates the scheduling constraints and 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 operation and once its value has been defined, it may be

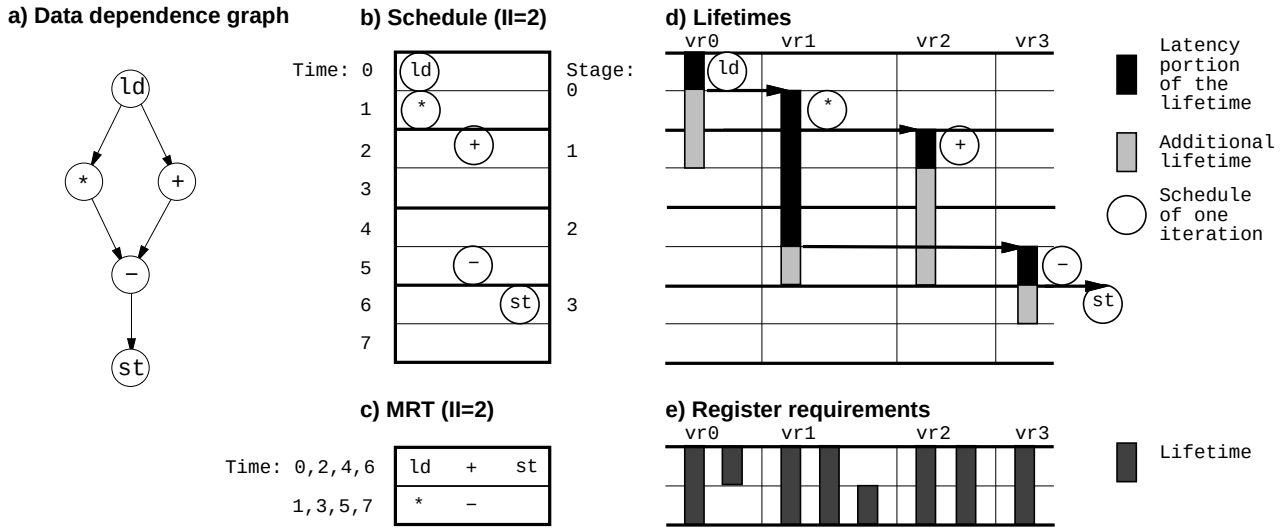


Figure 1: Modulo schedule for the kernel `y[i] = x[i]`$^2$`-x[i]-a` with minimum register requirements.

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. 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, 2, 5, and 6. The schedule can be divided into *stages* of $II$ cycles each. In this example, the above operations are respectively scheduled in stage 0, 0, 1, 2, and 3.

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 initiation interval is bounded by the *minimum initiation interval* (*MII*) [1], which is a lower bound on the smallest feasible value of $II$ for which a modulo schedule can be found. This lower bound is constrained either by critical resources being fully utilized or by critical loop-carried dependence cycles. Note that the *MII* lower bound is not a tight lower bound as they may be no feasible modulo schedules that achieves *MII*, possibly due to the presence complex resource patterns or to the interference between resource and dependence constraints [8].

The virtual register lifetimes associated with this iteration are presented in Figure 1d. The register requirements can also be computed by collapsing Figure 1d to $II$ rows, with wraparound, as shown in Figure 1e. We see that exactly 7 virtual registers are live in the first row and 7 in the second. Thus the register requirements, which are determined by the row with the maximum number of live values, and referred to as *MaxLive* [12], are thus 7 in this example.

## 3 Backgrounds on Optimal Modulo Scheduling

In this section, we first present the traditional formulation that is used by optimal modulo schedulers based on integer linear programming solvers. We then present a general framework used by optimal modulo schedulers to find a minimum-$II$ modulo schedule.

The traditional formulation used by optimal modulo schedulers consists of two types of variables per operation: one type describing the MRT row and one type describing the stage associated with each operation. The traditional formulation consists of three types of scheduling constraints: the assignment constraints, the dependence constraints, and the resource constraints. The formulation of the variables and the first two types of constraints (presented in Sections 3.1 and 3.2) were proposed by Govindarajan *et al* [7], and the last type of constraints (shown in Section 3.3) was proposed by us in [4].

Relating the more efficient formulation investigated in this paper to the traditional formulation, the more efficient formulation reuses the same variables, assignment constraints, and the resource constraints as the traditional formulation; however, the more efficient formulation differs in its formulation of the dependence constraints, which will be presented in Section 4.

In this work, 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.

### 3.1 Assignment Constraints

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 $r$ of the MRT and 0 otherwise.

The first condition that a valid modulo schedule must satisfy is that each operation is scheduled exactly once in the MRT:

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

Equation (1) defines all the assignment constraints, i.e. the constraints that assign each operation to exactly one row of the MRT.

### 3.2 Dependence Constraints

While the $A$ matrix defines the row in which each operation is scheduled, we must also select the stage in which each operation is placed. We represent the stage numbers 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 each operation is scheduled.

We now introduce two derived parameters that characterize the MRT row and the time at which each operation is scheduled, which are defined as follows:

$$row_i = \sum_{r=1}^{II-1} r * a_{r,i} \qquad time_i = k_i * II + row_i \qquad (2)$$

Note that since $a_{r,i} = 1$ precisely in the row in which operation $i$ is scheduled, and is 0 otherwise, $row_i$ is correctly computed and satisfies the following property: $row_i \in [0, II)$.

A modulo schedule must enforce all the scheduling 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 $i$:

$$(time_j + \omega_{i,j} * II) - time_i \;\geq\; l_{i,j} \qquad (3)$$

Substituting Equation (2) into Inequality (3) results in the following inequality:

$$\sum_{r=1}^{II-1} r * (a_{r,j} - a_{r,i}) \;+ (k_j - k_i) * II \;\geq\; l_{i,j} - \omega_{i,j} * II \qquad \forall (i,j) \in E_{sched} \qquad (4)$$

Inequality (4) defines all the dependence constraints of a modulo schedule for a given initiation interval $II$ with respect to the dependence distances $\omega_{i,j}$ and dependence latencies $l_{i,j}$ of a dependence graph $G$.

### 3.3 Resource Constraints

The third condition that a valid modulo schedule must satisfy is that no cycle of the schedule consumes more resources than are available in the machine. In this paper, we use the constraints derived in [4]:

$$\sum_{i=0}^{N-1} \sum_{c \in Res_{i,q}} a_{(r-c) \bmod II, i} \;\leq\; M_q \qquad \forall q \in Q,\ r \in [0, II) \qquad (5)$$

where $Q$ is the set of resource types, $M_q$ is the number of resources of type $q$, and $c \in Res_{i,q}$ indicates that operation $i$ uses a resource of type $q$ exactly $c$ cycles after being issued. Note that for machines where a mapping from each operation's resource usages to resource instances cannot be trivially found, the formulation proposed by Altman *et al* [5] should be used. A derivation of Inequality (5) as well as a precise definition of the machines for which Inequality (5) is applicable is found in [10].

### 3.4 Optimal Modulo Scheduling Framework

The traditional formulation of the modulo scheduling space is based on the assignment, dependence, and resource constraints as defined by Constraints (1), (4), and (5), respectively. In this formulation, and for a given $II$, each variable (i.e. each element of the $A$ matrix and $k$ vector) is only multiplied by a constant factor; thus an integer linear programming solver (IP solver) can be used to find a solution. However, since the primary objective is to minimize the initiation interval, a schedule with minimum $II$ is obtained by solving a series of integer programming problems until the smallest $II$ with a feasible solution is found.

A traditional framework of optimal modulo scheduler for minimum $II$ based on IP solvers is thus defined as follows. First, the minimum initiation interval (*MII*) [8] is computed, and the tentative $II$ is set to *MII*. Second, the integer linear programming system for the tentative $II$, given loop iteration, and given target machine is constructed. Third, an IP solver is used to solve the system, possibly minimizing a secondary objective function such as the schedule length of the loop iteration or the register requirements of the resulting modulo schedule. If the IP solver fails to find a feasible solution, the tentative $II$ is incremented by one, and the second and third steps are repeated; otherwise, the solution found is optimal.

## 4 Structured Formulation of the Dependence Constraints

Solving an integer linear programming system can be implemented by iteratively solving a linear programming model where additional constraints are introduced (and removed) to force each integer variable to an integer value without omitting from the solution space any optimal (integer) solution [13]. A branch-and-bound algorithm is used to determine which parts of the search space to consider, and each branch-and-bound node is evaluated by solving a linear programming model with the original constraints augmented by some additional constraints that force variables to integer values.

A key aspect for formulating an efficient integer program is to find a formulation that results in fewer branch-and-bound nodes, a goal that can be achieved in part by structuring the problem so that the linear programming solver naturally results in an integer solution for as many integer variables as possible. Recent work by Chaudhuri *et al* [14] on the structure of the scheduling problem has shown techniques to formulate efficient integer linear program for scheduling straight-line (nonloop, nonbranching) code. One beneficial property that can be derived from their theoretical results is defined here as follows:

**Definition 1 (0-1-Structured Constraints)** *A constraint is defined as 0-1-structured if each variable appears at most once, multiplied by either a* $0$, $+1$, *or* $-1$ *constant coefficient. By extension, a formulation is defined as 0-1-structured if each of its constraints are structured.*

Note that the assignment constraints defined by Equation (1) and the resource constraints defined by Inequality (5) satisfy this property, whereas the dependence constraints defined by Inequality (4) do not satisfy this property, since the elements of the $k$ vector are multiplied by $II$ and the elements of the binary $A$ matrix are multiplied by $r \in [0, II)$.

In this section, we investigate a more efficient formulation of the dependence constraints that can be used instead of Constraints (4) in the optimal modulo scheduling framework described in Section 3.4. Experimental evidence shown in Section 5 indicates that when this 0-1-structured formulation of the dependence constraints is used, the number of branch-and-bound nodes visited by the IP solver is decreased by two orders of magnitudes, on average, for a benchmark of 782 loops scheduled for minimum register requirements among all minimum-$II$ modulo schedules. The basic idea for this reformulation is due to Chaudhuri *et al* [14] which has such a reformulation for straight line (nonloop, nonbranching) code. The adaptation of this idea to modulo schedules for loop code is, however, not straightforward and substantially different in detail, as is the proof of the validity of this adaptation.

In the remainder of this section, we derive a 0-1-structured formulation of the dependence constraint associated with scheduling edge $(i, j)$ in three steps. First, we assume in Section 4.1 that we know the scheduling time of operation $i$ (i.e. $time_i$ is known and constant) and derive a structured constraint that precisely determines the scheduling times for operation $j$ that satisfy scheduling edge $(i, j)$. Second, we show in Section 4.2 how to extend this result without assuming the value of $time_i$ to be known and constant. Third, we derive in Section 4.3 the final structured formulation of the dependence constraints. Derivations in Sections 4.1 and 4.2 are original to our work, and the technique proposed by Chaudhuri *et al* for straight line code is adapted to loop code in Section 4.3.

### 4.1 Case with known $time_i$

Recall that a dependence constraint associated with scheduling edge $(i, j)$ enforces the scheduling dependence between operation $i$ and operation $j$, $\omega_{i,j}$ iterations later, and was defined in Inequality (3) as

$time_j + \omega_{i,j} * II - time_i \geq l_{i,j}$. Since each term in the dependence constraint has an integer value, we may reformulate this constraint as a strict inequality, i.e. $time_j + \omega_{i,j} * II - time_i > l_{i,j} - 1$. We may thus write:

$$time_j + \omega_{i,j} * II \quad > \quad time_i + l_{i,j} - 1 \qquad (6)$$

The left hand side of Inequality (6) corresponds to the times where the value produced by operation $i$ can legally be used by operation $j$ ($\omega_{i,j}$ iterations later). For conciseness, we refer to this value as $time_u$ (for the time of *use*) in the remainder of Section 4. Using the relations $time_j = k_j * II + row_j$ and $time_u = k_u * II + row_u$, we may thus define the stage and row of $time_u$ as, respectively,

$$k_u = k_j + \omega_{i,j} \qquad row_u = row_j \qquad (7)$$

Similarly, the right hand side of Inequality (6) corresponds to the latest time in which the value produced by $i$ is forbidden from use. We refer to this value as $time_f$ (for the last *forbidden* time) in the remainder of Section 4. Using the relations $time_i = k_i * II + row_i$ and $time_f = k_f * II + row_f$, we may thus define the stage and row of $time_f$ as, respectively,

$$\begin{aligned} k_f &= k_i + \left\lfloor \frac{row_i + l_{i,j} - 1}{II} \right\rfloor \\ row_f &= (row_i + l_{i,j} - 1) \bmod II \end{aligned} \qquad (8)$$

Note that since we assume here that the value of $time_i$ is known and constant, by extension, the values of $k_f$ and $row_f$ are known and constant as well.

Using the definitions from Equations (7) and (8), we may write the dependence constraint expressed in Inequality (6) as:

$$k_u * II + row_u \quad > \quad k_f * II + row_f \qquad (9)$$

We may transform Inequality (9) to isolate the two stage numbers, $k_u$ and $k_f$:

$$k_u - k_f \quad > \quad \frac{row_f - row_u}{II} \qquad (10)$$

Interestingly, we can show that the row difference $row_f - row_u$ has values in the range $(-II, II)$ since row numbers have by definition values in the range $[0, II)$. Consequently, the right hand side of Inequality (10) has values in the range $(-1, 1)$. Therefore, we can guarantee that when the integer valued stage difference $k_u - k_f$ is 1 (or larger), the dependence constraint is satisfied. Similarly, we can guarantee that when the stage difference $k_u - k_f$ is -1 (or smaller), the dependence constraint is violated. We may thus write:

$$(k_u > k_f) \quad \Rightarrow \quad \text{dep. is satisfied} \qquad (11)$$

$$(k_u < k_f) \quad \Rightarrow \quad \text{dep. is violated} \qquad (12)$$

Consequently we are able to determine the scheduling times of operation $j$ that satisfy the dependence constraint when $k_u \neq k_f$. Otherwise $k_f = k_u$, i.e. both the use time of the value produced by operation $i$ and the latest forbidden time for that value occur in the same stage of the schedule, and thus the specific values of $row_u$ and $row_f$ must be take into account to determine whether $time_j$ is feasible.

To evaluate this case, let $k_f = k_u$ and substitute $k_f$ for $k_u$ in Inequality (9), obtaining the following dependence constraint:

$$row_u \quad > \quad row_f \qquad (13)$$

Inequality (13) simply states that operation $j$ cannot be assigned to any of the rows $x \in [0, row_f]$ when $k_f = k_u$. Reformulating Inequality (13) using the binary $a_{r,j}$ variables, we obtain:

$$z = \sum_{x=0}^{row_f} a_{x,j} \quad = \quad 0 \qquad (14)$$

where the value of the sum in Equation (14) is referred to as $z$. Because of the assignment constraints, we know that only one $a_{r,j}$ variable is equal to 1, and all other $a_{r,j}$ variables are equal to 0. Thus, we know that $z$ is either 0 or 1, depending on the row in which operation $j$ is scheduled. Consequently, we can clearly see that Equation (14) is satisfied if and only if operation $j$ is not assigned to any of the rows in the range $[0, row_f]$. As a result, Inequality (13) and Equation (14) are equivalent. We may thus write:

$$(k_u = k_f)\ \&\ (z = 0) \quad \Rightarrow \quad \text{dep. is satisfied} \qquad (15)$$

$$(k_u = k_f)\ \&\ (z \neq 0) \quad \Rightarrow \quad \text{dep. is violated} \qquad (16)$$

To summarize our findings, we have shown that operation $j$ satisfies the scheduling edge $(i, j)$ if either Relation (11) is satisfied (i.e. $k_u > k_f$) or Relation (15) is satisfied (i.e. $k_u = k_f$ and $z = 0$). Otherwise, we have shown by Relations (12) and (16) that the dependence constraint is violated.

We may now combine the two disjoint Relations (11) and (15) by formulating the following constraint:

$$k_f - k_u + z \quad \leq \quad 0 \qquad (17)$$

Inequality (17) is equivalent to the union of Relations (11) and (15) because when $k_u > k_f$, Inequality (17) holds regardless of the value of $z$ since $z \in [0, 1]$ and when $k_u = k_f$, Inequality (17) holds precisely when $z = 0$.

Using the definitions in Equations (7), (8), and (14) in Inequality (17), we obtain the following constraint:

$$k_i + \left\lfloor \frac{row_i + l_{i,j} - 1}{II} \right\rfloor - k_j - \omega_{i,j} + \sum_{x=0}^{(row_i + l_{i,j} - 1) \bmod II} a_{x,j} \leq 0 \quad (18)$$

Consequently, we have derived a constraint that determines, for a given $time_i$ (expressed in terms of $k_i$ and $row_i$), the scheduling times for operation $j$ (expressed in terms of $k_j$ and $a_{x,j}$) that satisfy the scheduling edge $(i,j)$. This constraint is 0-1-structured in that its variables (i.e. $k_j$ and $a_{x,j}$) appear only once and are only multiplied by $+1$, 0, and $-1$ constant coefficients.

### 4.2 Case with unknown $\mathbf{time_i}$

We now extend the previous result to the case where the scheduling time of operation $i$, $time_i$, is not assumed to be known. Since the row in which operation $i$ is scheduled is also unknown, we must eliminate $row_i$ from Inequality (18). The main idea used in this section is to substitute $row_i$ by an arbitrary row $r \in [0, II)$, and transform the right hand side of Inequality (18) such that the new inequality is only constraining when operation $i$ is effectively scheduled in row $r$. Our claim is that the following inequality:

$$k_i + \left\lfloor \frac{r + l_{i,j} - 1}{II} \right\rfloor - k_j - \omega_{i,j} + \sum_{x=0}^{(r + l_{i,j} - 1) \bmod II} a_{x,j} \leq 1 - a_{r,i} \quad \forall r \in [0, II) \quad (19)$$

is equivalent to Inequality (18) when $r = row_i$ for some row $r \in [0, II)$ and is trivially satisfied otherwise. The sketch of a proof is as follows. Consider a row $r' \in [0, II)$. If operation $i$ is scheduled in row $r'$, by definition $a_{r',i} = 1$, and thus the right hand side of the Inequality (19) with $r = r'$ is 0. This inequality is thus identical to Inequality (18) since in this case $r = r' = row_i$. Otherwise, if operation $i$ is not scheduled in row $r'$, by definition $a_{r',i} = 0$, and thus the right hand side of the Inequality (19) with $r = r'$ is 1. This right hand side value is just large enough to ensure that this inequality is trivially satisfied. A detailed proof is provided in [10].

### 4.3 Final Formulation of the Structured Dependence Constraints

While we may simply formulate structured dependence constraints for each dependence edge $(i,j) \in E_{sched}$ using Inequality (19), we may further tighten the formulation of the scheduling space using an observation made by Chaudhuri *et al* [14] for dependent operations in straight line (nonloop, nonbranching) code.

Consider operation $i$, with latency $l$, that produces a value used by operation $j$. When operation $i$ is assigned to cycle $t$, *or any subsequent cycles* [14], operation $j$ must be assigned in a cycle $t' \geq t + l$. Using a similar observation here, we may replace $a_{r,i}$ in the right hand side of Inequality (19) by the sum of the $a_{x,i}$ variables over $x \in [r, II)$. Thus, we obtain the following final form of the structured dependence constraints:

$$\sum_{x=r}^{II-1} a_{x,i} + \sum_{x=0}^{(r + l_{i,j} - 1) \bmod II} a_{x,j} + k_i - k_j \leq \omega_{i,j} - \left\lfloor \frac{r + l_{i,j} - 1}{II} \right\rfloor + 1 \quad \forall r \in [0, II), (i,j) \in E_{sched} \quad (20)$$

At first, it may appear counterintuitive to replace Inequality (4) with Inequality (20) in order to obtain a more efficient formulation of the problem, since it corresponds to replacing each constraint with $II$ new constraints. The crucial point, however, is that the new constraints are 0-1-structured, since each variable, namely each $k$ and each element of $A$ appears at most once, and is multiplied by only $+1$, 0, or $-1$ coefficients.

## 5 Measurements

In this section, we evaluate four modulo scheduling algorithms based on integer linear programming formulation.

**MinReg Modulo Scheduler.** This scheduler finds a schedule with the minimum $II$ over all modulo schedules, and with the minimum register requirements among such schedules. It uses the integer programming model based on Constraints (1), (4), and (5) when evaluating the traditional formulation, and Constraints (1), (5), and (20) when evaluating the structured formulation of the modulo scheduling space. The formulation of the secondary objective function (for minimum register requirements) is based on the 0-1-structured formulation found in [4]. Recall that this algorithm achieves the minimum feasible register requirements for a given loop iteration, initiation interval, and set of machine constraints.

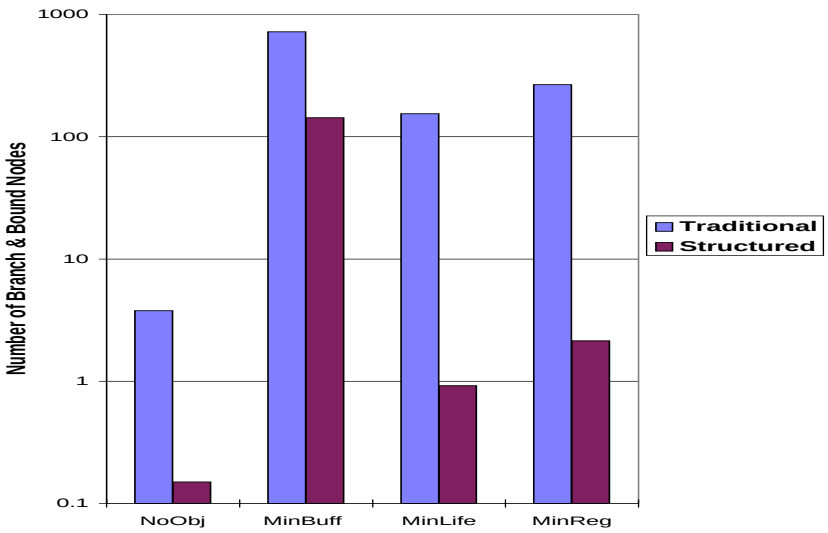


Figure 2: Average number of branch-and-bound nodes visited by the CPLEX solver.

**MinBuff Modulo Scheduler.** This scheduler finds a schedule with minimum $II$, and the minimum buffer requirements among all such schedules. Recall that unlike registers, buffers are reserved for integer multiples of $II$ cycles. When considering the traditional formulation of the modulo scheduling space, we use Constraints (4), (1), and (5,) as well as the formulation of the minimum-buffer secondary objective function found in [7]. When evaluating the structured formulation, we use Constraints (1), (5), and (20); in addition, we reformulate the minimum-buffer objective function as proposed by Dupont de Dinechin [15] in order to obtain a 0-1-structured formulation[1]. Although the MinBuff algorithm minimizes buffers, in our comparisons we always present the actual register requirements associated with these schedules.

**MinLife Modulo Scheduler.** This scheduler finds a schedule with the maximum steady-state throughput, and the minimum cumulative length of the lifetimes among all such schedules. When considering the traditional formulation of the modulo scheduling space, we use Constraints (1), (5), and (4,) as well as the formulation of the minimum-lifetime secondary objective function found in [16]. When evaluating the structured formulation, we use Constraints (1), (5), and (20); we also modify the formulation of the objective function (as devised for the MinBuff Modulo Scheduler) in order to obtain a formulation that is entirely 0-1 structured. In our comparisons, we also present the actual register requirements associated with these schedules.

**NoObj Modulo Scheduler.** This scheduler simply finds a schedule with minimum $II$, without minimizing any secondary objective function. It uses the same formulation of the modulo scheduling space as the MinReg Modulo Scheduler, and simply returns the first schedule that it finds.

In this study, we use a benchmark of 1327 loops obtained from the Perfect Club [17], SPEC-89 [18], and the Livermore Fortran Kernels [19], as compiled by the Cydra 5 Fortran77 compiler [20] after load-store elimination, recurrence back-substitution, and IF-conversion. The resource requirements of the Cydra 5 [21] are precisely modeled, using the reduced machine description produced in [22]. Note that we only model, for a given loop, the resources that are used by at least two operations, since the other resources, if any, pose no resource conflicts. Furthermore, we limit the schedule length of the modulo schedules sought to 20 cycles beyond the minimum schedule length, in order to achieve schedules with high transient performance. We use here the CPLEX solver, a widely available commercial integer linear programming solver, and never search for a schedule for more than 15 minutes per loop on a HP-9000/715 workstation. This time limit is arbitrary, and was fixed to complete the scheduling of the 1327-loop benchmark in reasonable time.

[1] As formulated in [7], the minimum-buffer objective function uses additional variables that are defined by constraints which are not 0-1-structured. By transforming the problem using a technique proposed in [15], however, we may define these additional variables using constraints of the same type as Constraints (20). As a result, we obtain a formulation that is entirely 0-1 structured.

In our first experiment, we investigate the benefits of using the structured formulation instead of the traditional formulation of the dependence constraints. We present data for the 653 loops that were successfully scheduled (using no more than 15 minutes per loop) by each algorithm and formulation of the modulo scheduling space.

We present in Figure 2 the average number of branch-and-bound nodes visited by the solver when using either the structured or the traditional formulation. Recall that the CPLEX solver used in our experiment explores branch-and-bound nodes when it must force variables to integral values. Note also that the Y-axis of Figure 2 is logarithmic.

The first observation is that the four schedulers benefit significantly from using the structured formulation of the dependence constraints which, for example, decreases the average number of branch-and-bound nodes visited by the MinLife and MinReg Modulo Scheduler by a factor of 167.4 and 124.5, respectively. The second observation is that the three algorithms that result in the lowest average number of branch-and-bound nodes are the NoObj, MinReg Modulo Schedulers with the structured formulation of the dependence constraints.

Because of this significant decrease in number of visited branch-and-bound nodes, schedulers based on the structured dependence constraints run significantly faster; for example, the structured MinReg Modulo Scheduler completes in 11.6% of the time (i.e. 101.0 instead of 870.2 seconds) when scheduling all the loops successfully scheduled by the MinReg Modulo Scheduler with unstructured dependence constraints. Consequently, the IP solver can schedule a larger fraction of the loops in the benchmark suite when using the structured dependence constraints; e.g. 1179 versus 1084 loops for the NoObj Modulo Scheduler, 898 versus 859 loops for the MinLife Modulo Scheduler, and 917 versus 782 loops for the MinReg Modulo Scheduler. We thus exclusively employ the structured dependence constraints in the remainder of this section.

In our second experiment, we investigate the performance of each modulo scheduling algorithm with the structured formulation of the dependence constraints in a benchmark containing all successfully scheduled loops. We present the performance characteristics of the four modulo scheduling algorithms with the structured formulation in Table 1. In addition to the number of branch-and-bound nodes, the table also lists data the numbers of variables and constraints prior to any simplifications that might be performed by the CPLEX solver, as well as the number of simplex iterations performed by the CPLEX solver, the number of operations, $N$, and the initiation interval, $II$. For comparison, we provide the same data for the four algorithms with the traditional formulation of the dependence constraints in Table 2; however, we only consider the structured formulation in the discussion below.

First, observe the distribution of the data in Table 1. As indicated by the low median values, relative to the average numbers, there is a large number of simple loops in the benchmark suites. However, the solvers clearly succeed in solving some rather large problems, as indicated by the large max values.

The second observation is that the NoObj Modulo Scheduler processes loops with significantly larger average numbers of operations, $II$, variables, and constraints than the three other algorithms. It also handles the loops with the largest maximum number of operations and $II$, i.e. 80 operations and 118 cycles, respectively. Note also that since it simply returns the first valid integral solution, the number of visited branch-and-bound nodes directly corresponds to the nodes that are needed by the solver to force all variables to integer values. As indicated in the table, very few branch-and-bound nodes are required (0 node in 73.9% of the loops, 7.93 nodes on average, and and never more than 337 nodes) compared to the traditional formulation (0 node in 37.4% of the loops, 249.64 nodes on average, and and never more than 29746 nodes) This data confirms again the benefit of structured formulations.

The third observation is that the MinReg Modulo Scheduler processes loops that are nearly as large as those the MinLife Modulo Scheduler can process, and significantly larger than those of the MinBuff Modulo Scheduler, even though MinReg Modulo Scheduler precisely minimizes the register requirements.

In our third experiment, we use the NoObj Modulo Scheduler to investigate the performance of the Iterative Modulo Scheduler proposed by Rau [8]. Since the Iterative Modulo Scheduler results in schedules with optimal throughput for 1274 (or 96.0%) of the 1327 loops in the benchmark suite, we only need to investigate and compare the remaining 53 loops.

Among these 53 loops, the NoObj Modulo Scheduler finds 2 loops where $II$ (that was obtained by the Iterative Modulo Scheduler) can be decreased by 2 (but not 3) cycles, i.e. the scheduler finds a schedule with an $II$ decreased by 2 cycles and shows that decreasing $II$ by 3 cycles is infeasible. The scheduler also finds 6 loops where $II$ can be decreased by 1 (but not 2) cycles. It furthermore shows that the $II$ of 22 of these loops cannot be decreased even by 1 cycle. Finally, the scheduler does not complete the remaining 23 loops within the 15 minute execution time limit. Thus, the NoObj Modulo Scheduler succeeds in finding schedules with better throughput for 8 (or 15.1%) of the 53 interesting loops.

Using the NoObj Modulo Scheduler, we have thus shown that the Iterative Modulo Scheduler actually

| Measurements: | min | freq | median | average | max |
|---|---|---|---|---|---|
| NoObj Modulo-Sched: (1179 loops) | | | | | |
| Variables | 4.00 | 0.3% | 33.00 | 183.12 | 3880.00 |
| Constraints | 8.00 | 0.3% | 67.00 | 298.66 | 5400.00 |
| Branch-and-bound nodes | 0.00 | 74.0% | 0.00 | 8.02 | 337.00 |
| Simplex iterations | 0.00 | 37.1% | 11.00 | 345.63 | 20645.00 |
| $II$ | 1.00 | 32.0% | 2.00 | 7.48 | 118.00 |
| $N$ | 2.00 | 0.4% | 9.00 | 13.95 | 80.00 |
| MinBuff Modulo Sched: (762 loops) | | | | | |
| Variables | 6.00 | 0.5% | 20.00 | 36.58 | 748.00 |
| Constraints | 9.00 | 0.5% | 33.00 | 76.11 | 1922.00 |
| Branch-and-bound nodes | 0.00 | 51.0% | 0.00 | 276.52 | 21551.00 |
| Simplex iterations | 0.00 | 0.1% | 10.00 | 1167.72 | 144568.00 |
| $II$ | 1.00 | 49.3% | 2.00 | 2.71 | 42.00 |
| $N$ | 2.00 | 0.7% | 5.00 | 6.81 | 36.00 |
| MinLife Modulo Sched: (898 loops) | | | | | |
| Variables | 8.00 | 0.4% | 48.00 | 169.50 | 5846.00 |
| Constraints | 11.00 | 0.4% | 72.00 | 230.79 | 6934.00 |
| Branch-and-bound nodes | 0.00 | 72.7% | 0.00 | 299.37 | 27142.00 |
| Simplex iterations | 1.00 | 18.3% | 29.00 | 2181.56 | 221039.00 |
| $II$ | 1.00 | 36.3% | 2.00 | 5.84 | 118.00 |
| $N$ | 2.00 | 0.6% | 7.00 | 9.04 | 41.00 |
| MinReg Modulo Sched: (917 loops) | | | | | |
| Variables | 7.00 | 0.4% | 36.00 | 119.67 | 5810.00 |
| Constraints | 10.00 | 0.4% | 57.00 | 169.24 | 6975.00 |
| Branch-and-bound nodes | 0.00 | 63.5% | 0.00 | 124.71 | 10711.00 |
| Simplex iterations | 0.00 | 24.5% | 23.00 | 2539.64 | 167504.00 |
| $II$ | 1.00 | 41.1% | 2.00 | 4.63 | 118.00 |
| $N$ | 2.00 | 0.5% | 7.00 | 8.39 | 41.00 |

Table 1: Measurements with structured scheduling constraints.

| Measurements: | min | freq | median | average | max |
|---|---|---|---|---|---|
| NoObj Modulo Sched: (1084 loops) | | | | | |
| Variables | 4.00 | 0.4% | 27.00 | 115.95 | 3630.00 |
| Constraints | 8.00 | 0.4% | 40.00 | 74.12 | 792.00 |
| Branch-and-bound nodes | 0.00 | 37.4% | 4.00 | 249.64 | 29746.00 |
| Simplex iterations | 0.00 | 35.6% | 10.00 | 1731.96 | 424846.00 |
| $II$ | 1.00 | 35.0% | 2.00 | 6.50 | 118.00 |
| $N$ | 2.00 | 0.5% | 8.00 | 10.76 | 52.00 |
| MinBuff Modulo Sched: (762 loops) | | | | | |
| Variables | 6.00 | 0.5% | 20.00 | 49.39 | 2520.00 |
| Constraints | 9.00 | 0.5% | 26.00 | 43.36 | 698.00 |
| Branch-and-bound nodes | 0.00 | 52.1% | 0.00 | 790.17 | 26730.00 |
| Simplex iterations | 0.00 | 30.3% | 3.00 | 1668.74 | 65961.00 |
| $II$ | 1.00 | 49.6% | 2.00 | 3.93 | 118.00 |
| $N$ | 2.00 | 0.7% | 5.00 | 6.47 | 21.00 |
| MinLife Modulo Sched: (859 loops) | | | | | |
| Variables | 6.00 | 0.5% | 28.00 | 60.13 | 2520.00 |
| Constraints | 9.00 | 0.5% | 37.00 | 52.13 | 698.00 |
| Branch-and-bound nodes | 0.00 | 49.6% | 1.00 | 711.44 | 29262.00 |
| Simplex iterations | 0.00 | 26.9% | 4.00 | 1984.99 | 88811.00 |
| $II$ | 1.00 | 43.9% | 2.00 | 4.37 | 118.00 |
| $N$ | 2.00 | 0.6% | 7.00 | 7.51 | 29.00 |
| MinReg Modulo Sched: (782 loops) | | | | | |
| Variables | 7.00 | 0.5% | 26.00 | 77.59 | 4978.00 |
| Constraints | 10.00 | 0.5% | 32.00 | 77.32 | 3273.00 |
| Branch-and-bound nodes | 0.00 | 51.4% | 0.00 | 518.32 | 26524.00 |
| Simplex iterations | 0.00 | 28.5% | 3.00 | 4228.82 | 322537.00 |
| $II$ | 1.00 | 48.2% | 2.00 | 3.67 | 118.00 |
| $N$ | 2.00 | 0.6% | 5.00 | 6.81 | 25.00 |

Table 2: Measurements with traditional scheduling constraints.

finds a schedule with maximum throughput for 22 more loops than previously known, i.e. it results in schedules with maximum throughput for 1296 (or 97.7%) of the 1327 loops in the benchmark suite. In addition to these 22 loops, the NoObj Modulo Scheduler finds 8 more loops with maximum throughput (with a higher throughput than the schedule found by the Iterative Modulo Scheduler); thus we have now schedules with maximum throughput for 1304 (or 98.3%) of the loops in the benchmark suite. At this point, it is unknown whether the remaining 23 loops have suboptimal throughput.

To summarize the findings of this section, the algorithms based on structured formulation are clearly more efficient than the traditional formulations. The second result is that the NoObj Modulo Scheduler clearly schedules the largest fraction of the loops in the benchmark suite. This result is not unexpected since this formulation does not minimize any objective function. The third and surprising result is that using the MinReg scheduler is nearly as efficient as using the MinLife scheduler, and is much more efficient than using the MinBuff scheduler, in spite of the fact it precisely minimizes MaxLive.

## 6 Conclusions

In this paper, we contribute a more structured formulation of the modulo scheduling solution space. This more efficient formulation addresses a major concern with modulo schedulers that are based on integer linear programming solvers [6], which is their traditionally high execution time. Experimental evidence (gathered for a benchmark suite of 1327 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels, compiled for the Cydra 5 machine) indicates that the number of branch-and-bound nodes is decreased on average by a factor of 103, when using the structured instead of the traditional formulation of the MinReg Modulo Scheduler on the 782 loops successfully scheduled by both formulations. It results, in turn, in a decrease in the total execution time by a factor of 8.6, from 870.2 to 101.0 seconds. Also, using the more efficient representation enables us to successfully schedule a larger fraction of the 1327 loops; for example the coverage increases from 58.9% to 69.1% for the MinReg Modulo Scheduler and from 81.7% to 88.8% for the NoObj Modulo Scheduler. Also, significantly larger loops are successfully scheduled: for example, the maximum number of operations in a loop increases from 25 to 41 operations for the MinReg Modulo Scheduler and from 52 to 80 for the NoObj Modulo Scheduler. These results are obtained when scheduling for the Cydra 5 machine, a machine with complex resource requirements.

Using this more efficient formulation of the modulo scheduling solution space, we may carry through relevant performance evaluations of existing scheduling heuristics. For example, we have analyzed the performance of the Iterative Modulo Scheduler proposed by Rau [3][8]. While this algorithm was known to achieve the minimum initiation interval (i.e. *MII*) in 96.0% of the 1327 loops in the benchmark suite, we can show that it actually achieves *MII* in 97.7% of the loops. Furthermore, our algorithm finds schedules with lower *II* for 8 of the 31 remaining loops. We have also analyzed the register requirements of the stage scheduling heuristics proposed in [9][10] used in conjunction with the Iterative Modulo Scheduler. Using the MinReg Modulo Scheduler, we find schedules with lower register requirements for 23.6% of the 1327 loops. By evaluating the stage scheduling heuristics with the MinLife and the MinBuff Modulo Schedulers, a schedule with lower register requirements is found in 18.5% and 4.5% of the loops, respectively. However, the heuristic finds a schedule with lower register requirements in 3.2% and 12.3% of the loops, respectively. These results confirm that optimal algorithms must be both precise and efficient to provide useful insights.

## Acknowledgments

The authors would like to thank B. Ramakrishna 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. We also appreciate the suggestions of the referees which significantly improved the quality of the paper. This work was supported in part by the Office of Naval Research under grant number N00014-93-1-0163 and by Hewlett-Packard.

## References

[1] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *Fourteenth Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[2] P. Y. Hsu. *Highly Concurrent Scalar Processing*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 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] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Optimum modulo schedules for minimum register

requirements. *Proceedings of the International Conference on Supercomputing*, pages 31–40, July 1995.

[5] E. R. Altman, R. Govindarajan, and G. R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 139–150, 1995.

[6] J. R. Ruttenberg, G. R. Gao, and A. Stoutchinin. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 1–11, May 1996.

[7] 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.

[8] B. R. Rau. Iterative Modulo Scheduling. *International Journal of Parallel Programming*, 24(1):2–64, 1996.

[9] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 180–191, November 1995.

[10] A. E. Eichenberger. *Modulo Scheduling, Machine Representations, and Register-Sensitive Algorithms.* PhD thesis, University of Michigan, Department of Electrical Engineering and Computer Science, Ann Arbor, MI, 1996.

[11] 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.

[12] 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.

[13] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, New York, 1988.

[14] S. Chaudhuri, R. A. Walker, and J. E. Mitchell. Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem. *IEEE Transactions on Very Large Scale Integration Systems*, 2(4):456–471, December 1994.

[15] B. Dupont de Dinechin. Parametric computation of margins and of minimum cumulative register lifetime dates. *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, 1996.

[16] B. Dupont de Dinechin. Simplex scheduling: More than lifetime-sensitive instruction scheduling. *Proceedings of the International Conference on Parallel Architecture and Compiler Techniques*, pages 327–330, 1994.

[17] 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.

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

[19] 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.

[20] J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. In *The Journal of Supercomputing*, volume 7, pages 181–227, 1993.

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

[22] A. E. Eichenberger and E. S. Davidson. A reduced multipipeline machine description that preserves scheduling constraints. *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 12–22, May 1996.