Minimum Register Requirements for a Modulo Schedule

Alexandre E. Eichenberger and Edward S. Davidson
Advanced Computer Architecture Laboratory
EECS Department, University of Michigan
Ann Arbor, MI 48109-2122

Santosh G. Abraham
Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

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 minimum register requirements, given a modulo reservation table. Our method determines optimal register requirements for machines with finite resources and for general dependence graphs. This method demonstrates the potential of lifetime-sensitive modulo scheduling and is useful in evaluating the performance of lifetime-sensitive modulo scheduling heuristics.

1 Introduction

Software pipelining is a technique for exploiting the instruction level parallelism present among the iterations of a loop by overlapping the execution of several loop iterations. With sufficient overlap, some functional unit can be fully utilized, resulting in a schedule with maximum throughput.

Modulo scheduling [1][2][3] restricts the scheduling space by using the same schedule for every iteration of a loop and by initiating successive iterations at a constant rate, i.e. one *initiation interval* (II clock cycles) apart. This restriction is known as the *modulo scheduling constraint*. Under this constraint, no resource is used by one iteration more than once at times that are congruent modulo II. As a result, searching for a schedule with a given II is greatly simplified.

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][6]. Loops with conditional exits can also be modulo scheduled [7]. Furthermore, the code expansion due to modulo scheduling can be eliminated when using special hardware, such as rotating register files and predicated execution [8].

Since modulo scheduling achieves a high throughput by overlapping the execution of several iterations, it results in higher register requirements. Furthermore, the register requirements increase as the concurrency increases [9], whether due to using and exploiting machines with faster clocks and deeper pipelines, wider issue, or a combination of both. As a result, a scheduling algorithm that reduces the register pressure while scheduling for high throughput is increasingly important. Lifetime-sensitive modulo-scheduling proposed by Huff [10] is a heuristic that attempts to address this concern.

In this paper, we treat modulo scheduling as a three step procedure. Some code generation strategies combine steps to simultaneously meet distinct objectives; e.g. Huff's approach combines steps one and two below.

1. *MRT-scheduling* primarily addresses resource constraints and is best implemented by using a *modulo reservation table* (MRT) [2][11] which contains II rows, one per clock cycle, and one column for each resource at which conflicts may occur. Filling the MRT consists of packing the operations of one iteration within the II rows of the MRT to obtain a schedule with no resource conflicts that achieves the highest possible steady state performance. This step determines the modulo II distance between any two operations.

2. *Iteration-scheduling* primarily addresses dependence constraints, which specify that the distance of each pair of dependent operations is no less than the latency for such a pair. These constraints are satisfied by delaying operations by some (non-negative integer) multiple of II.

3. *Register allocation* performs the actual allocation of virtual registers to physical registers, a scheme that may vary depending on the hardware support available and the desired level of loop unrolling and loop peeling [8][12].

In this paper we present a method that performs the second step in an optimal fashion for loop iterations that consist of a single basic block with a general dependence graph. For if-converted basic blocks, we assume that all predicates are set to true when performing Step 2. Our iteration-scheduling method satisfies the dependence constraints while simultaneously minimizing MaxLive [1], 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. We present two algorithms for *iteration-scheduling*, both of which schedule for machines with finite resources. The first linear-time algorithm can handle loops whose dependence graphs

To appear in the *Proceedings of the 27th Annual International Symposium on Microarchitecture*, page 75-84, November 1994, San Jose, California

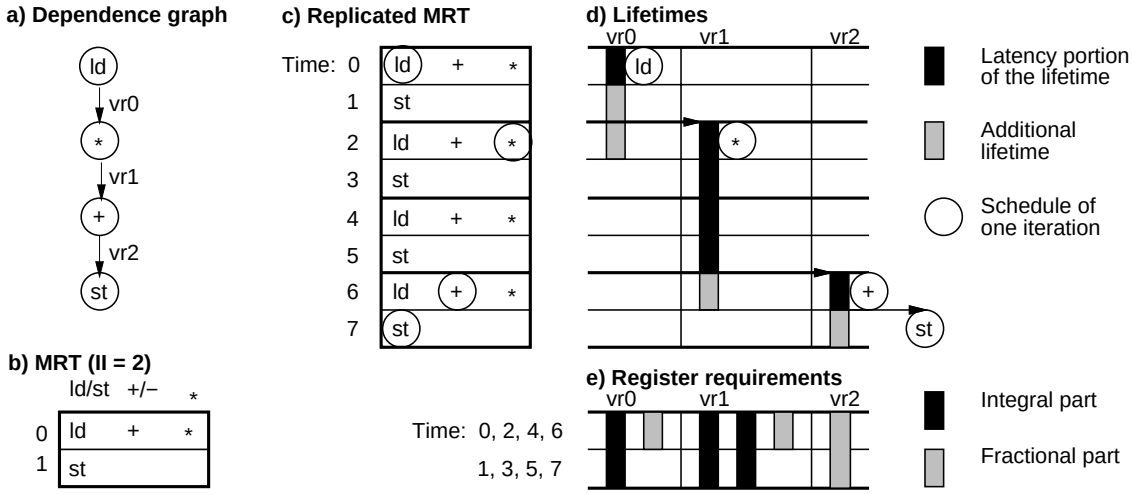

Figure 1: MRT and iteration-scheduling for the kernel `y[i] = x[i]`^{2}`+a` (Example 1).

have acyclic underlying graphs[1]. The second general algorithm uses a linear programming approach and can handle general dependence graphs with unrestricted loop-carried dependences and common sub-expressions. We can also quickly determine when the faster first method is applicable.

MaxLive can be reduced to a minimum by jointly solving Steps 1 and 2; however, this remains a difficult problem under realistic constraints. This paper contributes an efficient, optimum solution for Step 2 given an MRT which by itself can significantly reduce register requirements. 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 extends the linear-time algorithm for determining minimum register requirements by Mangione-Smith et al. [9], which requires that each virtual register is used at most once. Our linear-time algorithm permits multiple uses of each virtual register, provided that the underlying dependence graph is acyclic. Our second (more complex) method handles general dependence graphs, including loop-carried dependence and common sub-expressions.

This work complements the register allocation algorithm presented by Rau et al. [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. Their algorithm, after extensive loop unrolling and consequent code expansion, typically achieves register allocations that are within four registers of the MaxLive bound on machines without hardware support. Huff [10] presents a lower bound that is independent of the MRT-schedule. Our minimum MaxLive value is dependent on the given MRT-schedule; however, our MaxLive is tight since we also produce an actual iteration schedule that achieves this MaxLive.

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.

[1]The underlying graph is an undirected graph formed by ignoring the direction of each arc.

2 Register Requirements

In this section, we present the dependence constraints and illustrate the choices faced by the iteration-scheduler with a few examples. The example target machine is a hypothetical processor with three independent and fully pipelined functional units, viz. `load/store`, `add/sub`, `mult/div`. 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. Our first example uses a simple kernel to illustrate how to compute the register requirements of a modulo-scheduled loop. This kernel is: `y[i] = x[i]`^{2}`+a`, where the value of `x[i]` is read from memory, squared, incremented by a constant, 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 used by several operations. This graph thus defines the definition-use chain (du-chain) of operations that define and use each virtual register. 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 MRT-scheduler places each operation of an iteration somewhere within the modulo reservation table (MRT). Figure 1b illustrates an MRT-schedule with $II = 2$ for the kernel of Example 1 on the target machine. A `load`, an `add`, and a `mult` operation are scheduled in the first row and a `store` operation is scheduled in the second row of the MRT.

Once the MRT-schedule is completed, the iteration scheduler delays each operation by some integer multiple of II cycles so that it is scheduled only after its input values have

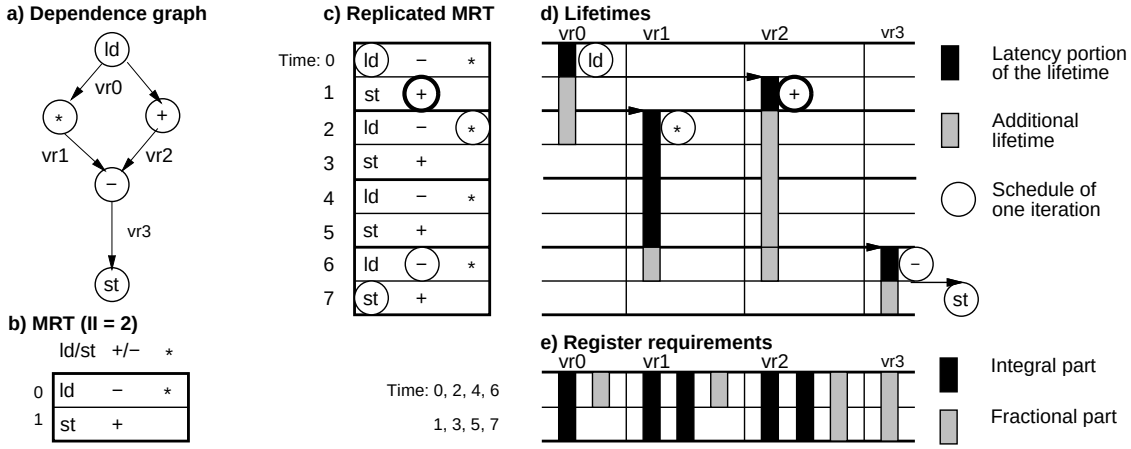

Figure 2: MRT and iteration-scheduling for the kernel `y[i] = x[i]`^{2}`-x[i]-a` (Example 2) with `add` scheduled early.

been calculated and made available to it by its predecessor operations. An iteration-schedule for Example 1 is shown in Figure 1c. This figure is an execution trace; the MRT is replicated sufficiently to show one complete iteration. The circles highlight the operations that belong to the iteration initiated at a time 0. The circles must be placed so as to satisfy the specified latencies.

The virtual register lifetimes for this iteration are presented in Figure 1d. The black bars correspond to the initial portion of the lifetime which 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 MRT-schedule 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 circled operation is placed during the portion of the lifetime that overlaps with the black bar of a virtual register that it uses. For example, the `add` operation uses `vr1` and thus cannot be scheduled at time 2 or 4, but can be scheduled at time 6. Since the `add` operation is placed in row 0 of the MRT, only the rows in the replicated MRT that are congruent to 0 (mod II) are considered. The `add` will thus be delayed by an integer multiple of II, $2*II$ in this example.

We represent the time interval between two dependent operations by counting the number of times that a use operation is *skipped* before being scheduled. For example in Figure 1c, from the row of the `load` operation (row 0) to the row of the `mult` operation (row 2), the `mult` operation is skipped once in row 0. Therefore, we say that the `load`/`mult` operation pair has a *skip factor* of 1. Similarly, the skip factor is 2 for the `mult`/`add` operation pair and 0 for the `add`/`store` operation pair. The skip factor is within 1 of the *iteration difference* used in [9] [13].

The problem addressed in this paper can now be defined formally as follows: choose a skip factor (place a circle in the replicated MRT) for each operation in one iteration so as to generate a valid iteration-schedule that minimizes MaxLive. Because of the modulo constraint, MaxLive can be quickly computed, as illustrated in Figure 1e, by wrapping the lifetimes for the first iteration around a vector of length II. In particular, Figure 1e shows the number of live registers in steady-state loop execution and is be 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. In Figure 1e, we see that exactly six virtual registers are live in the first row and five in the second, resulting in a MaxLive of six.

Figure 1e distinguishes between two distinct contributions to each virtual register lifetime: the integral and the fractional part. For virtual registers with several uses, only the last use and its skip factor, s, is considered. The *integral part* spans the entire MRT exactly s times as shown in Figure 1e. The *fractional part* spans the rows in the MRT inclusively from the def operation row forward with wraparound through the last-use operation row. The length of the fractional part thus 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 MaxLive* is defined as the number of integral part 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. The total MaxLive is the sum of the fractional and integral MaxLives.

The iteration-schedule presented in Figure 1 results in the minimum register requirements for this kernel, MRT, and set of functional unit latencies. Mangione-Smith et al. [9] have shown that for dependence graphs with at most a single use per virtual register, as in Example 1, minimizing each skip factor individually always results in the minimum register requirements. However, this result does not apply to general dependence graphs with unrestricted common subexpressions and loop carried dependences. Our next example illustrates such a dependence graph, in which minimizing each skip factor individually does not result in the minimum register requirements.

Example 2. Our second 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 2a. We notice that two operations use `vr0` and that two distinct du-chains connect the `load` to the `sub` operation, resulting in a cycle in the underlying graph. We call such cycles *underlying-cycles*.

Figure 2b illustrates an MRT-schedule for the kernel of Example 2 and Figure 2c presents an execution trace with individually minimized skip factors.

We have already seen that a valid iteration-schedule must enforce the dependence constraints. In the presence of a cy-

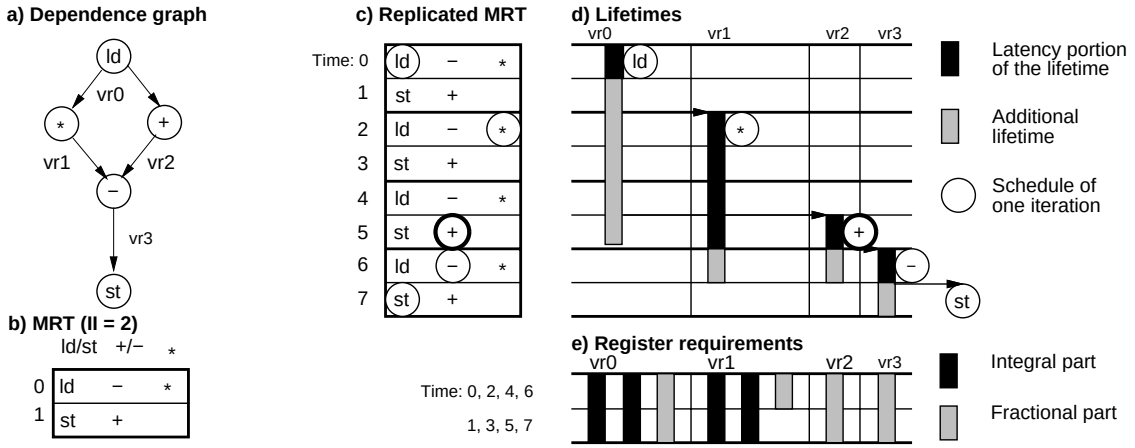

Figure 3: MRT and iteration-scheduling for the kernel `y[i] = x[i]^2-x[i]-a` (Example 2) with `add` scheduled late.

cle in the underlying dependence graph, we must also insure that the cumulative distances on a path that traverses an underlying-cycle is zero. The *cumulative distance* of a path is the algebraic sum of the distances of consecutive def/use operation pairs (edges) along that path, where the distance is taken as negative if the edge is traversed in the direction from `use` to `def`. In Figure 2d, this new constraint insures that the two skip distances arriving to the `sub` operation are chosen consistently. We will refer to these new constraints as the *underlying-cycle* constraints.

In Figure 2 the `add` operation was scheduled as soon as the `load` completed. However, we can also schedule the `add` later without delaying the `sub`, namely at time 3 or 5 since the result of this operation must be kept alive through time 6 in either case. In Figure 3, the `add` operation is scheduled at time 5. By comparing Figures 2e and 3e, we see that scheduling the add operation later increases the lifetime of `vr0` by $1\frac{1}{2}$ columns, but decreases the lifetime of `vr2` by 2 columns. As a result, MaxLive is reduced from 9 to 8 by scheduling the add operation late. This result is surprising since we might expect the lifetime increase of `vr0` to match the 2-column lifetime decrease of `vr2`. However, with an early `add`, the `mult` is the last use of `vr0` and the first cycle of additional `add` delay (row 2) does not increase the `vr0` lifetime.

Example 2 shows that scheduling each operation as early as possible does not necessarily result in minimum register requirements for dependence graphs that have cycles in their underlying graphs, and that a more global iteration-scheduling algorithm is needed.

To summarize the findings of this section, we have seen that the iteration-scheduler has an impact on the register requirements for a given kernel, MRT, and set of functional unit latencies. This iteration-schedule must enforce the dependence constraints, scheduling each operation only after all its input operands are available. In the presence of cycles in the underlying dependence graph, we must also insure that the cumulative distance that traverses the edges of each underlying-cycle is zero. Since underlying-cycles can interact with one another, minimizing the register requirements can only be achieved in general by reconciling these interactions globally for general dependence graphs.

3 Iteration-Scheduling for Minimum Integral MaxLive

In this section, we present an algorithm that finds an iteration-schedule resulting in the minimum integral MaxLive for a given kernel, MRT, and set of functional unit latencies. We first present the variables that characterize an MRT and an iteration-schedule. Then, we present the conditions that a valid schedule must fulfill and the method that searches for an iteration-schedule with the minimum integral MaxLive. We conclude by extending this method to dependence graphs with loop-carried dependencies.

We omit here the fractional MaxLive because its behavior is highly non-linear. This omission results in iteration-schedules that require no more than one additional register per virtual register used multiple times and used at least once by a use operation in an underlying-cycle. However, our final algorithm will take both the integral and the fractional MaxLive into account.

From the given latency set and MRT, we calculate the following parameters which describe how each virtual register, vr, is defined and used:

d_i	row number of the MRT containing the operation that *defines* vr_i
u_i^x	row number of the MRT containing the x$^{\text{th}}$ operation that *uses* vr_i
n_i	*number* of operations that use vr_i
l_i	*latency* of the operation that defines vr_i

We saw in Section 2 that an iteration-schedule is defined by the skip factor associated with each def/use operation pair of the dependence graph. We distinguish here between two causes that lead to skipping an operation:

s_i^x	minimum (non-negative integer) skip factor necessary to *hide* the latency l_i for the def/use operation pair d_i/u_i^x
p_i^x	additional (non-negative integer) skip factor used to *postpone* the u_i^x operation further

The p_i^x variables are the fundamental unknowns to be decided by the iteration-scheduling algorithm. They will be assigned non-negative integer values that minimize MaxLive,

subject to the underlying cycle constraints. We now develop an algebraic problem statement for the algorithm.

The distance between an ordered pair of rows of an MRT with II rows is defined as:

$$dist(X,Y) = (Y - X) \bmod II \tag{1}$$

and corresponds to the distance from row X to the next Y row, possibly in the next instance of the MRT. For example, if $II = 6$, $dist(1,3) = 2$ and $dist(3,1) = 4$. In a valid iteration-schedule, the distance of a def/use operation pair, d_i/u_i^x, must be no less than the latency of the def operation:

$$dist(d_i, u_i^x) + s_i^x * II \geq l_i \tag{2}$$

namely that the distance from the def operation MRT row number to the use operation MRT row number, increased by skipping s_i^x entire instances of the MRT, must be larger than the latency of the def operation of vr_i. Since we are interested in the smallest non-negative integer value of s_i^x that satisfies Equation (2), we obtain:

$$s_i^x = \left\lceil \frac{l_i - dist(d_i, u_i^x)}{II} \right\rceil \tag{3}$$

All dependence constraints are satisfied when each use operation is skipped at least s_i^x times before being scheduled.

Finally, we need to guarantee that all underlying-cycle constraints are also fulfilled. Consider a directed closed path in the dependence graph that traverses some underlying-cycle, called *cycle*. Define $sign_i^x$ to be $+1$ if the closed path traverses the edge from the d_i operation to the u_i^x operation (referred to as (i,x) below) in the positive (def-to-use) direction, and -1 if the closed path traverses an edge in the negative (use-to-def) direction. $Sign_i^x$ is undefined if the corresponding edge, (i,x), is not in the underlying-cycle. The iteration-schedule distance of a def/use operation pair is now:

$$dist(d_i, u_i^x) + (s_i^x + p_i^x) * II \tag{4}$$

The cumulative distance of an underlying-cycle is thus:

$$\sum_{(i,x) \in cycle} sign_i^x(cycle) \{dist(d_i, u_i^x) + (s_i^x + p_i^x) * II\} \tag{5}$$

By setting this cumulative distance to zero we obtain the following constraint:

$$\sum_{(i,x) \in cycle} sign_i^x(cycle) p_i^x = -\delta_{cycle} - \sum_{(i,x) \in cycle} sign_i^x(cycle) s_i^x \tag{6}$$

where δ_{cycle} is:

$$\delta_{cycle} = \frac{1}{II} \sum_{(i,x) \in cycle} sign_i^x(cycle)\ \ dist(d_i, u_i^x) \tag{7}$$

The first noticeable fact is that all but the p_i^x variables are fully defined by the MRT parameters. The direction chosen for the path traversal determines the $sign_i^x$ values, but both directions result in the same Equations (6) and (7). We are therefore free to set the p_i^x variables to any positive integer values, as long as they satisfy Equation (6). The second important fact is that Equation (6) has a solution only if δ_{cycle} is itself integer valued, since all the s_i^x and p_i^x are integer valued. This is fortunately always the case, because the cumulative distance from an operation to itself is by definition a multiple of II.

Example 3. Figures 4a and 4b, respectively, show the dependence graph and an optimal iteration schedule of the kernel `y[i] = (x[i]`^{4}`+ x[i]+a)/ (x[i]+ b)`. By postponing the schedule of the $\texttt{add}_2$ and $\texttt{add}_3$ operation from time 3 and 2 to time 9 and 8 respectively, the register requirements decrease from 15 to 13. Furthermore, selecting the right iteration schedule is even more important with wider-issue machines; for example, with a five-wide machine, postponing these two `add` operations reduces the register requirements from 23 to 19 for this kernel and set of latency values. Moreover, this result is achieved with the best MRT for this kernel and set of latencies.[2]

We can now use Equation (3) to compute all the s_i^x values. The resulting skip factors are shown in Figure 4a. By convention, we number the multiple def/use operation pairs in the dependence graph for a single virtual register from left to right starting with $x = 0$.

Using these skip factors and computing the δ values according to Equation (7), as shown in Figure 4a for counter-clockwise traversal of underlying cycles A and B, the constraints for this kernel correspond to the following set of underlying-cycle constraints:

$$\begin{aligned} &\text{Constraint A:} \quad p_0^0 + p_1^0 + p_4^0 - p_2^0 - p_0^1 = -2 \\ &\text{Constraint B:} \quad p_0^1 + p_2^0 + p_5^0 - p_3^0 - p_0^2 = 0 \end{aligned} \tag{8}$$

It is sufficient to introduce one constraint per elementary cycle of the underlying graph, because the constraints of arbitrary complex cycles are simply a linear combination of the constraints of their elementary cycles. For example, the constraint associated with the outer cycle of Figure 4 as computed with Equation (7): $p_0^0+p_1^0+p_4^0+p_5^0-p_3^0-p_0^2 = -2$ can be obtained by adding the constraints for cycles A and B. Therefore, satisfying constraints A and B will necessarily satisfy the constraint associated with the outer cycle.

Although any p_i^x values that satisfy these constraints result in a valid iteration-schedule, we are interested in the solution that minimizes integral MaxLive. As shown in Section 2, the integral part of a virtual register lifetime is directly proportional to the sum of the skip and postpone factors of the last-use operation. Therefore, the integral part of the lifetime of vr_i is:

$$\max_{x=0..n_i-1} (s_i^x + p_i^x) \tag{9}$$

Using Function (9), we compute the integral MaxLive of the kernel of Example 3 as the sum of the lifetimes over all virtual registers:

$$\begin{aligned} &\max(p_0^0, p_0^1 + 1, p_0^2) + p_1^0 + p_2^0 + p_3^0 + \\ &p_4^0 + p_5^0 + p_6^0 + 5 \end{aligned} \tag{10}$$

We can now reduce the problem of finding an iteration-schedule that results in the minimum integral MaxLive to a well-known class of problems, solved by a linear programming (LP) solver [15]. Note, however, that (10) is not acceptable as the input to an LP-solver, because the objective function cannot contain any max functions. However, since we are minimizing the objective function, we can remove the max function by using some additional inequalities, called *Max Constraints*. Finally, we can remove the constant term

[2] We performed an exhaustive search over all feasible MRT's to determine the optimal MRT for this kernel.

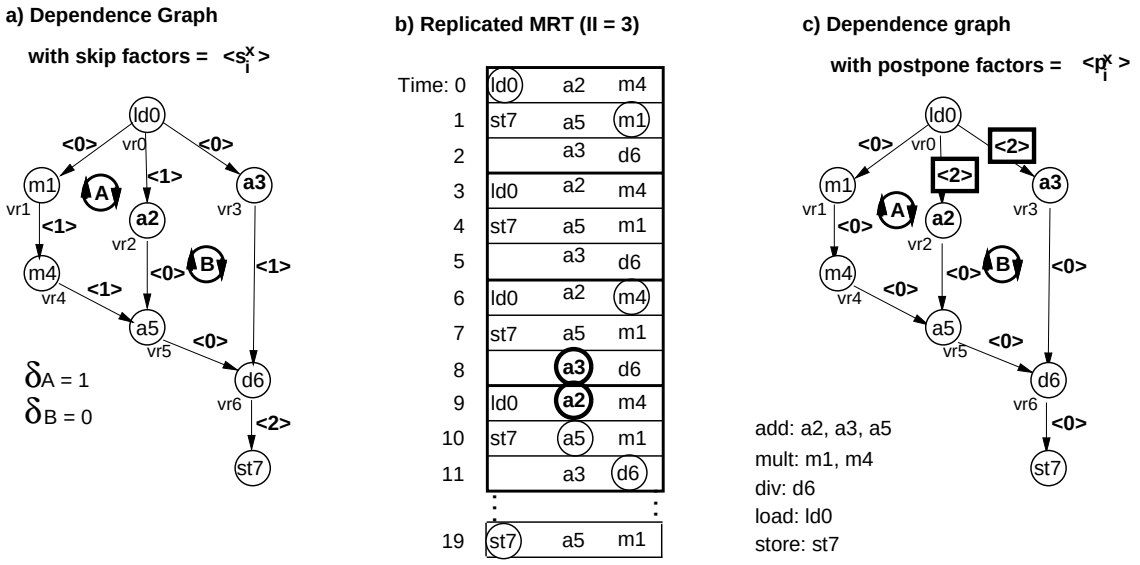

Figure 4: Skip, postpone, and delta factors for the kernel `y[i] = (x[i]^4+x[i]+a)/(x[i]+b)` (Example 3).

in the objective function. The LP-solver input for the kernel and MRT presented in Figure 4 is therefore:

$$\begin{aligned}
\textbf{Minimize:} \quad & m_0 + p_1^0 + p_2^0 + p_3^0 + p_4^0 + p_5^0 + p_6^0 \\
\text{Constraint A:} \quad & p_0^0 + p_1^0 + p_4^0 - p_0^1 - p_2^0 = -2 \\
\text{Constraint B:} \quad & p_0^1 + p_2^0 + p_5^0 - p_0^2 - p_3^0 = 0 \\
\text{Max Constraints:} \quad & p_0^0 \le m_0; \;\; p_0^1 + 1 \le m_0; \\
& p_0^2 \le m_0 \qquad (11)
\end{aligned}$$

The result of the LP-solver is shown in Figure 4c. We can verify that the solution for the p_i^x values, satisfies the constraints of Equation (11) and yields $m_0 = 3$ with the other p_i^x in the objective function being 0, resulting in an integral MaxLive $= m_0 + 5 = 8$ registers.

Algorithm 1 *The minimum integral MaxLive for a kernel with a general dependence graph,*[3] *MRT, and set of functional unit latencies is found as follows:*

1. *Compute all s_i^x using Equation (3) and search for all elementary cycles in the underlying dependence graph [16].*
2. *If the underlying dependence graph is acyclic, the solution that produces the minimum integral MaxLive is obtained by setting the values of all p_i^x to zero.*
3. *Otherwise, build an underlying-cycle constraint for each elementary cycle of the underlying dependence graph using Equations (6) and (7). Derive the integral MaxLive objective function by summing Function (9) over i.*
4. *Solve the system of constraints to minimize the integral MaxLive by using an LP-solver. The solution defines the p_i^x values that result in the minimum integral MaxLive.*

[3] We have not yet dealt with loop-carried dependencies. At the end of this section however, we will show that loop-carried dependencies can be handled in a similar fashion.

We will prove the correctness of this algorithm with two theorems. The first theorem validates the correctness of the solution for general dependence graphs, and the second theorem validates the solution and the linear solution time for the case of an acyclic underlying dependence graph.

Theorem 1 *The solution of the minimum integral MaxLive LP-problem as defined in Algorithm 1 results in the minimum integral MaxLive for a kernel with a general dependence graph, MRT, and set of latencies. This solution is guaranteed to be integer valued.*

Proof: We have already shown that valid iteration-schedules must satisfy the s_i^x and the underlying-cycle constraints. Therefore, this set of equations defines the space of feasible schedules. Since the underlying-cycle constraints and the objective function (integral MaxLive) are linear, we can use an LP-solver to find a schedule that minimizes integral MaxLive. Since the solution, namely the set of p_i^x values, must be integer valued, we must show that the solution of the LP-solver is guaranteed to be integer valued for any dependence graph and MRT-schedule.

Fortunately, the minimum integral MaxLive problem is analogous to the minimum cost flow problem, a class of problems that satisfies the integer property [15]. This property holds when two conditions are fulfilled. The first condition specifies that the variables (p_i^x and m_i) can only be multiplied by the coefficients +1, -1, or 0. This condition holds since each edge occurs at most once on the path of an underlying-cycle. This property holds since we need only consider the elementary cycles. The second condition specifies that all constant terms must be integer valued, which is the case since the s_i^x and δ values are guaranteed to be integer valued. □

Theorem 2 *For a dependence graph with an acyclic underlying dependence graph, the minimum integral MaxLive for a given kernel, MRT, and set of latencies is found in a time that is linear in the number of edges in the dependence graph. The minimum integral MaxLive is found by simply setting all p_i^x values to zero.*

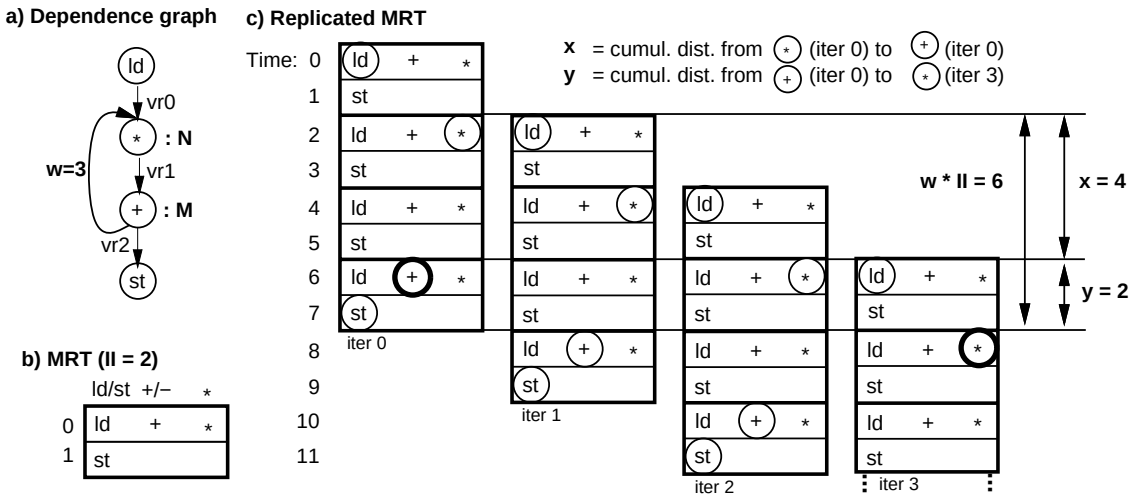

Figure 5: MRT and iteration-scheduling for the kernel `y[i] = x[i]*y[i-3]+a` (Example 4).

Proof: Consider applying the general form of Algorithm 1 (Steps 1, 3, and 4) to an acyclic underlying dependence graph. There are no underlying cycles and hence no underlying-cycle constraints in the linear program problem formulation (see (11)). The max constraints all have the form $p_i^x \leq m_i + c_i^x$ where m_i and p_i^x are to be found and c_i^x is a constant. The objective is to minimize a linear combination of the m_i and p_i^x, all of which have non-negative coefficients. Consequently the solution cost is monotonically non-decreasing in the m_i and p_i^x. Setting all $p_i^x = 0$, their minimum possible value, also allows the m_i to have their minimum feasible value and minimizes the objective function. Therefore, Steps 3 and 4 take no time and Step 2 of Algorithm 1 can be used instead.

The solution time for Algorithm 1 is thus reduced to the time for Step 1. The s_i^x are each computed in constant time using Equation (3). Since there is one s_i^x parameter per edge in the dependence graph, the solution time is linear in the number of edges. □

We conclude this section by presenting how this method is extended for dependence graphs with loop-carried dependences. In the case of a loop-carried dependence, we must enforce a precise ordering between operation pairs that may span several iterations. For example in the kernel `x[i] = x[i]+ x[i-2]`, we must guarantee that the latency of the `add` operation of iteration `i` completes before the `add` operation of iteration `i+2` is issued.

Example 4. Our fourth kernel is: `y[i] = x[i]* y[i-3]+ a`, where the value of `x[i]` is read from memory, multiplied by `y[i-3]`, incremented by `a`, and stored in `y[i]`, as shown in the dependence graph of Figure 5a. The backward edge labeled `w=3` from the `add` operation to the `mult` operation represents a loop-carried dependence, where the result of the `add` is reused by the `mult` three iterations later.

The iteration-schedule for four consecutive iterations is shown in Figure 5c. The loop-carried dependence is satisfied because the result of the `add` operation (iteration 0, cycle 6) is available to the `mult` operation three iterations later (iteration 3, cycle 8). For vr_2, let the `store` be use 0 and the `mult` be use 1. We can then define s_2^1 for the backward edge normally using Equation (3); Equation (4) then indicates the d_2/u_2^1 distance. The loop-carried dependence of Example 4 is satisfied if the cumulative distance from the `mult` operation to the `add` operation and back to the `mult` operation along the backward edge is equal to the dependence distance, ω, multiplied by II. The value of p_2^1 must be chosen to satisfy this constraint. A formal description follows.

Consider a pair of operations, N/M, connected by a loop-carried dependence from operation M to N spanning ω iterations. Define *cycle* to be a directed closed path in the dependence graph that traverses an underlying-cycle containing the operation pair N/M and the loop-carried dependence edge with iteration-dependence distance ω. This loop-carried dependence is satisfied if the cumulative distance of the underlying-cycle *cycle* is equal to $\omega * II$. The *loop-carried dependence* constraint is thus the same as Equation (5), but is set equal to $\omega * II$, rather than to 0. Namely,

$$\sum_{(i,x)\in cycle} sign_i^x(cycle)\,\{dist(d_i, u_i^x) + (s_i^x + p_i^x) * II\} = \omega * II \quad (12)$$

We can rewrite this equation as:

$$\sum_{(i,x)\in cycle} sign_i^x(cycle) p_i^x = \omega - \delta_{cycle} - \sum_{(i,x)\in cycle} sign_i^x(cycle) s_i^x \quad (13)$$

where δ_{cycle} is:

$$\delta_{cycle} = \frac{1}{II} \sum_{(i,x)\in cycle} sign_i^x(cycle) dist(d_i, u_i^x) \quad (14)$$

and is guaranteed to be integer valued. Moreover, we see that loop-carried dependence constraints also satisfy the integer solution property for the LP-solver. Therefore, Theorem 1 can be directly extended for loop-carried dependences. Theorem 2 cannot be extended for loop-carried dependences since they introduce cycles in the underlying dependence graph.

4 Iteration-Scheduling for Minimum Total MaxLive

In this section, we extend the previous iteration-schedule algorithm to consider both the fractional and the integral register requirements. We first quantify the fractional MaxLive associated with a kernel, an MRT, and an iteration-schedule. We investigate the interaction between the fractional MaxLive and the search for an optimal iteration schedule, and then present an algorithm that takes the total MaxLive into account when searching for an optimal iteration schedule.

In machine model of this paper, we assume that a virtual register is reserved when its def operation is scheduled and freed one cycle after the last use operation is scheduled. As shown in Section 2, the fractional lifetime is highly dependent on the rows in which a virtual register is reserved and freed. We therefore introduce two new variables:

r_i	row number of the MRT in which vr_i is reserved
f_i	row of the MRT in which vr_i is freed

The last-use operation is:

$$u_i^{last} \equiv_{II} d_i + \max_{x=0,\ldots,n_i-1} \{dist(d_i, u_i^x) + (s_i^x + p_i^x) * II\} \quad (15)$$

In our machine model, we have:

$$r_i \equiv_{II} d_i \quad (16)$$

$$f_i \equiv_{II} u_i^{last} + 1 \quad (17)$$

A virtual register, vr_i, is live at row z if row z is encountered in the MRT by going forward (with wraparound) from row r_i to row f_i. Using Equations (16) and (17), we know that vr_i is live at row z if row z is closer to f_i than to r_i in modulo space. Since the fractional MaxLive is determined from the maximum number of live virtual registers in any single row, we can write the total MaxLive for a given kernel, MRT and iteration schedule as:

$$R_{totalMaxLive} = \sum_{i \in I} (s_i^{last} + p_i^{last}) + \max_{z=1,\ldots,II} \sum_{i \in I} \left\{ \begin{array}{ll} 1 & \text{if } dist(z, f_i) \leq dist(z, r_i) \\ 0 & \text{otherwise} \end{array} \right\} \quad (18)$$

where the first and second term correspond, respectively, to the integral and the fractional MaxLive.

We can now formalize the interaction between the fractional and the integral part of the register requirements. Note that when we are minimizing the integral MaxLive, we may affect which use operation is last, since there are p_i^x terms inside the max function of Equation (15). Deciding which use operation is last may in turn affect the fractional MaxLive, but only if we increase the number of live virtual registers in a critical row, as expressed by the max function in the second term of Equation (18).

Let us assume that we have found an iteration schedule that results in the minimum integral MaxLive but not in the minimum fractional MaxLive; namely that the fractional register requirements can be reduced by choosing other last-use operations. For example, suppose that the fractional MaxLive would decrease by one if the y$^{\text{th}}$ use operation of vr_i were made the last-use operation. Since we are interested in finding an iteration schedule that results in the minimum achievable register requirements, we must check if there is a solution that schedules the y$^{\text{th}}$ use operation last without increasing integral MaxLive. We can search for this new iteration schedule, as presented in Algorithm 1, with an additional set of constraints that forces the y$^{\text{th}}$ use operation to be scheduled laster than some other (x$^{\text{th}}$) use operations. This new set of constraints is:

$$s_i^x + p_i^x < s_i^y + p_i^y \quad \text{for all } x \in 0 \ldots n_i - 1 \quad \text{such that } dist(d_i, u_i^x) > dist(d_i, u_i^y) \quad (19)$$

In particular, Equation (19) forces each use operation (x$^{\text{th}}$) to be scheduled after the y$^{\text{th}}$ operation if the fractional lifetime of vr_i from d_i to the x$^{\text{th}}$ use operation exceeds the fractional lifetime of the y$^{\text{th}}$ use operation. An u_i^x use operation that does not satisfy the condition of Equation (19) would do no worse a last-use than the y$^{\text{th}}$.

Algorithm 2 *The minimum total MaxLive for a kernel with a general dependence graph, MRT, and set of functional unit latencies is:*

1. *Use Algorithm 1 to compute the minimum integral MaxLive. We refer to this resulting iteration schedule as the* base *solution.*
2. *If the underlying dependence graph is acyclic, the base solution results in the minimum MaxLive.*
3. *Otherwise, compute the total MaxLive associated with the base solution using Equation (18) and determine the set of additional constraints that may improve the fractional MaxLive.*
4. *Use Algorithm 1 repeatedly to compute the minimum integral register requirements for the base system augmented by various subsets of the additional constraints as defined in Equation (19). Interesting subsets to be evaluated consist of all possible forcing combinations of an attractive last-use for some or all virtual registers whose fractional lifetime can be improved. The schedule among these that achieves the best total MaxLive is optimum.*

We prove the correctness of this algorithm with two theorems. The first theorem validates the correctness of the solution for general dependence graphs, and the second theorem for acyclic underlying graphs.

Theorem 3 *The Algorithm 2 produces the minimum MaxLive for a kernel with a general dependence graph, MRT, and set of latencies. This solution is guaranteed to be integer valued.*

Proof: Since we test all the combinations that potentially reduce the fractional MaxLive, and since Algorithm 1 produces the minimal integral MaxLive, the solutions of Algorithm 2 correspond to the minimum achievable total MaxLive for this MRT. It is easy to see that the additional set of constraints as defined in Equation (19) also satisfies the guaranteed integer solution property for the LP problems. Therefore, the solution produced by an LP-solver for each use of Algorithm 1 is guaranteed to be integer valued. □

While there is potentially an exponential number of additional constraints to test, this number is in practice relatively small for several reasons. First, the number of tests is directly dependent on the number of cycles in the underlying dependence graph, which is usually a small number. Second, we have to consider only the cases that effectively reduce the fractional MaxLive. We do not need to consider use operations unless they may impact the MRT row that could govern the second term of Equation (18). Finally, we can use inclusion properties to reduce the total number of tests.

Theorem 4 *For dependence graphs with acyclic underlying graphs, the solution that minimizes the integral MaxLive also minimizes the total MaxLive for a given kernel, MRT, and set of latencies.*

Proof: The solution that minimizes the integral register requirements may not be the solution with the minimal fractional register requirements. Recall, however that the Algorithm 1 solutions for acyclic underlying graphs has all $p_i^x = 0$. Therefore, reducing a fractional lifetime for some vr_j requires setting some p_j^x to 1, increasing the integer lifetime for vr_j by 1. Therefore, reducing the fractional register requirements for vr_j cannot decrease the MaxLive of this virtual register. Thus, we cannot reduce the register requirements of the base solution. □

5 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 a given a modulo reservation table. This method finds an iteration-schedule with the minimum register for general dependence graphs on machines with finite resources. When known to be optimal, a linear-time algorithm is used; otherwise, a linear programming approach is used. We can also quickly determine when the faster algorithm is applicable.

This paper demonstrates by example that selecting a good iteration schedule among all schedules that share the same MRT can result in lower register requirements. Though the general 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 lifetime-sensitive modulo scheduling heuristics.

Acknowledgement

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 insights and useful suggestions.

References

[1] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, overview, and perspective", *in The Journal of Supercomputing*, vol. 7, pp. 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, C. D. Glaeser, and R. L. Picard, "Efficient code generation for horizontal architecture: Compiler techniques and architecture support", *Proceedings of the Ninth Annual International Symposium on Computer Architecture*, pp. 131–139, 1982.

[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*, pp. 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] N. J. Warter, *Modulo Scheduling with Isomorphic Control Transformations*, PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[7] P. P. Tirumalai, M. Lee, and M. S. Schlansker, "Parallelization of loops with exits on pipelined architectures", *Proceedings of Supercomputing '90*, pp. 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*, pp. 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*, pp. 260–271, 1992.

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

[11] J. H. Patel and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays", *Proceedings of the Third Annual International Symposium on Computer Architecture*, pp. 159–164, 1976.

[12] C. Eisenbeis, W. Jalby, and A. Lichnewsky, "Squeezing more performance out of a Cray-2 by vector block scheduling", *Proceedings of Supercomputing '88*, pp. 237–246, November 1988.

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

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

[15] F. S. Hillier and G. J. Lieberman, *Introduction to Mathematical Programming*, McGraw-Hill, 1990.

[16] J. C. Tiernan, "An efficient search algorithm to find the elementary circuits of a graph", *Communications of the ACM*, pp. 722–726, December 1970.