Register Allocation for Predicated Code*

Alexandre E. Eichenberger and Edward S. Davidson

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

Abstract

Current compilers for VLIW and superscalar machines increase the instruction level parallelism of an application by merging several basic blocks into an enlarged predicated block, resulting in higher performance code but increased register requirements. We present a framework that computes precisely the interferences among virtual registers in the presence of predicated operations. Graph-coloring based register allocators can directly use the resulting interference graph. For interval-graph based register allocators, and others, we propose a technique that reduces the register requirements by allowing non-interfering virtual registers that overlap in time to share a common virtual register. Preliminary measurements on a benchmark of loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels indicate the effectiveness of this technique.

KEYWORDS: Register Allocation, Predicated Execution, Interference, Hyperblocks, Software Pipelining.

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 machines and reducing the schedule length of a code. There is generally insufficient parallelism within individual basic blocks, higher levels of parallelism can be obtained by also exploiting the instruction level parallelism among successive basic blocks. A well established approach uses predication to merge several basic blocks into a single enlarged predicated block. This approach relies on predicated operations [1], a class of operations that complete normally when a logical expression, referred to as the *predicate* of an operation, evaluates to true. Otherwise, the predicated operation is transformed into a `no-op` and has no side effect.

For scalar code, the *hyperblock* approach [2] used in the IMPACT compiler combines frequently executed basic blocks from multiple execution paths into an enlarged predicated block. This technique enables a range of compiler optimizations and facilitates the task of the scheduler, resulting in greatly improved schedules. For loop code, conditional statements within the innermost loop are merged to enable efficient software pipelining techniques [1][3][4] which exploit the instruction level parallelism present among the iterations of a loop by overlapping the execution of consecutive loop iterations. *IF-conversion* [5][6] is the enabling technique used in both scalar and loop code to translate the selected basic blocks into an enlarged predicated block.

However, predicated blocks with high levels of parallelism result in higher register requirements, for two main reasons. First, the register requirements increase as more values are needed to support more concurrent operations. This effect is inherent to parallelism and is exacerbated by wider machines and higher latency pipelines [7]. Second, we show in this paper that the register requirements increase for predicated code as current compilers do not allocate registers as well in predicated code as in unpredicated code.

Developing compiler techniques that exploit instruction level parallelism while containing the register requirements is crucial to the performance of future machines. In previous work, a virtual register in a single predicated block is considered live from the cycle in which it is first defined to the cycle in which it is last used, regardless of the predicates under which its def and use operations are guarded. In this paper, we demonstrate that the register requirements can be decreased if this assumption is relaxed, i.e. if the predicate expressions are taken into account.

*Appeared in MICRO-28, pp 180-191, November 1995.

The first contribution of this paper is a framework to analyze predicated codes. Information about predicated values are extracted from the code, live ranges and their predicate expressions are discovered, and live range interferences are computed. This framework can process arbitrary predication schemes and arbitrary relations among predicates, including correlations between IF-converted branches. This framework is applicable either before or after scheduling. The current implementation relies on a symbolic package to generate accurate results, using an approach similar to the one taken in the PARADIGM compiler [8].

For register allocators based on the classical graph coloring method, originally proposed by Chaitin [9][10], register allocation for predicted codes can be simply achieved by using a refined interference graph instead of the conventional one. However, several register allocators depart from the graph coloring method: e.g. Hendren *et al* [11] investigate a framework based on cyclic interval graphs, introducing the notion of time in the register allocator paradigm. This additional notion of time is particularly useful for the live ranges of a loop, where live ranges may cross the boundary of an iteration. Another approach, investigated by Rau *et al* [12], proposes a general framework for the allocation of registers in software pipelined loops for various code generation and hardware support schemes.

The second contribution of this paper is a set of heuristics that reduces the register requirements by allowing non-interfering virtual registers that overlap in time to share a common virtual register. We refer to this process as the *bundling* of compatible virtual registers. Bundling is similar to Chaitin-style coalescing [9] in that two or more virtual registers are combined; however, bundling differs in that it combines virtual registers with distinct values. Bundling is performed after constructing the refined interference graph, and before using a conventional register allocation based on either graph coloring, interval graphs, or Rau's software pipelining register allocator. Bundling is also applicable either before or after scheduling.

We investigate the performance of several bundling heuristics in a benchmark suite of modulo scheduled loops extracted from the Perfect Club, SPEC-89, and the Livermore Kernels. To minimize the interaction between bundling and scheduling heuristics, we used the scheduling technique described in [13] which minimizes the register requirements for a modulo schedule. Preliminary results indicate that our best bundling heuristic decreases the average register requirements from 39.3 to 36.5 registers and increases the percentage of loops with no more than 64 registers from 85% to 95%. The average register requirements of our best heuristic is 1% above a predicate-sensitive lower bound. Preliminary results indicate that bundling compatible virtual registers is more successful when applied after rather than before scheduling.

To our knowledge, the general issue of resource allocation in the context of predicated code has not been addressed in the literature. However, information about predicate values has recently been introduced in research compilers. In the IMPACT compiler, for example, information about predicates is stored in a Predicate Hierarchy Graph (PHG) [2] used to determine useful relations among predicate values. This information is used to refine dataflow analysis, optimization, scheduling, and allocation in presence of hyperblocks. Additionally, this information is used to conditionally reserve functional units when modulo scheduling under the Reverse-IF-Conversion scheme [14]. A complementary approach is taken with the Gated Single Assignment (GSA) form [15], where precise predicate information is embedded in the dataflow graph. This approach is used in the Polaris parallelizing compiler to refine data and memory dependence analysis and to aid loop parallelization [16].

In this paper, we illustrate the impact of predicated code on the register requirements and outline our general framework in Section 2. The bundling heuristics after scheduling and before scheduling are respectively introduced in Sections 3 and 4. We present experimental results in Section 5 and conclude in Section 6.

2 Predication and Live Ranges

In this section, we illustrate why current compilers succeed in accurately determining the live ranges of virtual registers in regular basic blocks, but fail when these basic blocks are converted to a single predicated block. We illustrate this negative impact by investigating the register requirements of a simple conditional statement before and after IF-conversion. We propose a solution that eliminates this negative impact by developing a precise representation of the live range in single-entry single-exit predicated blocks.

Example 1 Our example is based on the simple conditional statement presented in Figure 1a. In this example, the value of `y` is defined by one of two expressions of `x` depending on the value of `x`. Figure 1b shows the flow graph which has four basic blocks. Note that either `bb1` or `bb2` will be executed, but not both, and they both store their result in virtual register `vr3`. We assume that there are no further references to any of these virtual registers before the first or after the last basic block.

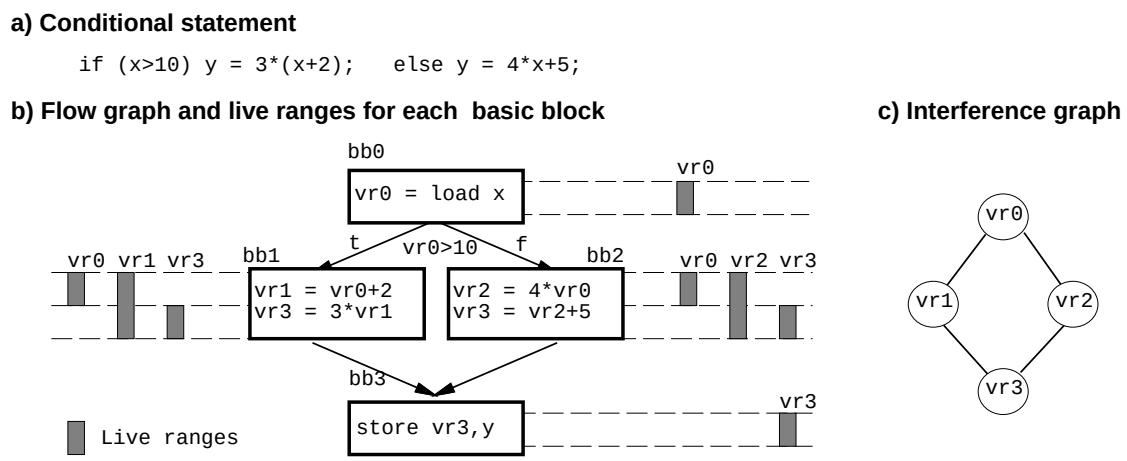

Figure 1: Register requirements of a conditional statement (Example 1).

During the register allocation phase, an interference graph [9][10] is typically constructed to determine the virtual registers that must be assigned to distinct physical registers. An *interference graph* corresponds to an undirected graph in which the vertices represent the virtual registers and an edge connects two vertices if one is live at the point where the other is defined.

To determining the live ranges, global dataflow analysis is performed on the flow graph to determine the set of virtual registers that are live-in at the entry point of each basic block, and those that are live-out at the exit point of each basic block. Then, local analysis for each basic block determines the cycle-by-cycle live range of each virtual register that is live-in, live-out, and/or referenced in the basic block [17, pp 534–525]. This second step is highly machine dependent; we assume in our examples and experiments that a virtual register is reserved in the cycle where its earliest-define operation is scheduled and becomes free in the cycle following its last-use operation. However, our formulation treats the cycles in which a register is reserved and freed as parameters.

Figure 1c presents the resulting interference graph. In `bb1` for example, `vr0` is live when `vr1` is defined; therefore, there is an edge between vertices `vr0` and `vr1` in the interference graph. Notice that there is no edge between vertices `vr1` and `vr2` since the thread of execution moves *exclusively* to one of the two basic blocks `bb1` and `bb2`. The resulting interference graph can be colored by no less than 2 colors; consequently, the (minimum) register requirements of this conditional statement is 2.

We now investigate the impact of predication on the register requirements of Example 1. Using IF-conversion, the four basic blocks of Figure 1b are merged into the single predicated block of Figure 2a. The operations previously executed in `bb1` and `bb2` are guarded, respectively, by the predicates `p1` and `p2`. The two predicates are defined in the second line by a special operation which sets `p1` to true if the comparison succeeds and false otherwise. Similarly, `p2` is defined by the same operation to true if the comparison fails and false otherwise. Note that the values of `p1` and `p2` are *disjoint*, i.e. `p1` and `p2` cannot both evaluate to true for any instance of `vr0`.

An interference graph using traditional live ranges is shown in Figure 2b. This interference graph contains an edge between `vr1` and `vr2` because `vr1` and `vr2` are considered live simultaneously, as traditional analysis does not examine the predicate expressions. As a result, the register requirements of this predicated block increase from 2 to 3.

This increase in register requirements is mainly due to two reasons. First, without knowledge about predicates, the dataflow analysis must make conservative assumptions about the side effects of the predicated operations [17]. Second, the solutions of the dataflow analysis rely heavily on the connection topology among basic blocks in the flow graph, which is altered by the IF-conversion process.

To eliminate the negative impact of predicated operations on the register requirements of a predicated block, one solution is to maintain the flow graph as predicated blocks are formed. This approach presents the advantage of using existing compiler techniques, but maintaining a consistent dataflow graph through the several optimization phases may be complex and time consuming. For example, extracting instruction level parallelism relies heavily on moving operations among predicated blocks and on speculating operations, e.g. by promoting the predicate guarding operations. Therefore, all optimizations modifying pred-

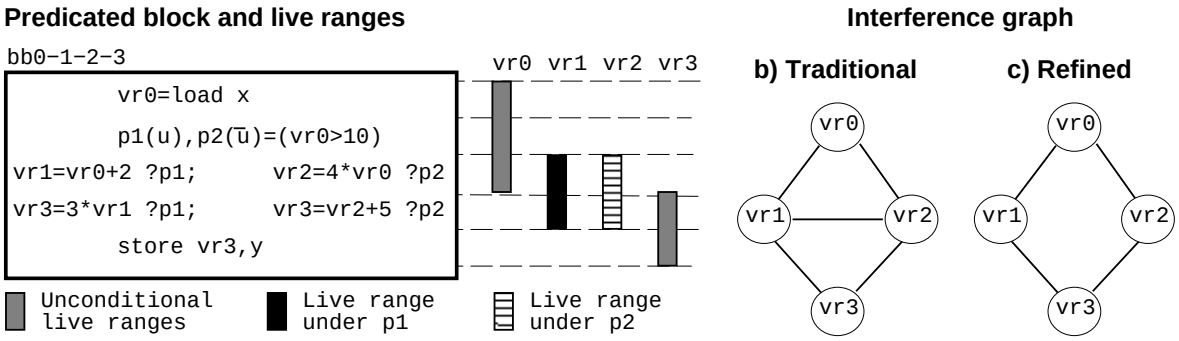

Figure 2: Register requirements of an IF-converted conditional statement (Example 1).

icates would have to maintain the consistency of the dataflow graph at each step.

Another approach is to reconstruct a dataflow graph from the predicated blocks each time dataflow analysis is needed [2]. This approach eliminates the complexity of maintaining a consistent dataflow graph and requires no changes to the classical dataflow analysis, traditional optimizations and register allocation. Additionally, the complexity of this approach is low, resulting in low compile time and good maintainability. However, this approach may result in conservative results. Consider the code of Figure 3a where operation **vr4=vr4+1** can be scheduled in any cycle after the load operation. If this operation is scheduled in cycle 2 (or in any cycles before or after all the predicated operations), reconstructing a flow graph results in the topology of Figure 3b. When scheduled in cycle 4, however, the graph is as in Figure 3d. Determining precise interference would then require the analysis phase to realize that both conditionals are correlated.

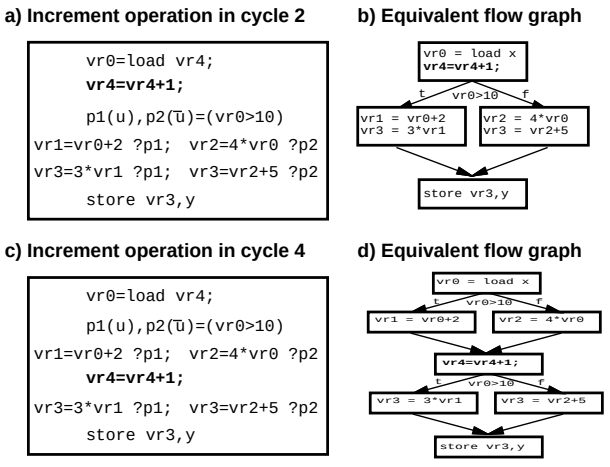

Figure 3: Reconstructing a dataflow graph.

In this paper, we investigate an approach that directly takes into account the predicate expressions under which virtual registers are live. This approach presents none of the cited drawbacks of the previous approaches; moreover, precise information about predicate expressions of live ranges may also be extremely useful to other optimizations, such as memory disambiguation [16]. The drawback of this approach is its reliance on a more expensive analysis, partly due to the use of a symbolic package to detect predicate expression disjointness and partly due to the computation of a more complicated interference relation. However, ongoing research on efficient algorithms for precise or approximate analysis of predicated code [18] will significantly improve the execution time of this approach. We now present the mechanisms required by our framework in more detail.

2.1 Predicate Extraction

The predicate extraction mechanism allows the compiler to find how predicates are defined and used. This knowledge is expressed as logically invariant expressions, or *P-facts*, that are guaranteed to hold, regardless of the execution trace, the results of comparisons, and the values of other predicates. These P-facts will be used in later phases of the analysis to determine feasible execution paths, i.e. execution paths that do not violate any of the P-facts gathered during this initial extraction phase. Note that failing to extract a P-fact does not compromise the integrity of the analysis, it merely causes in more conservative results.

This extraction mechanism is very sensitive to the instruction set architecture, since it relies on the precise semantics of the operations that define the predicate values. In this work, we adopt the predication scheme of the HPL Playdoh architecture [19]. Support for predicated execution is provided by Nx1-bit predicate register files which hold the predicate values, an additional source operand for most operations to spec-

Code fragments	Intermediate code	P-facts
`if(i>15) S1;` `else S2;`	`p1(u),p2(ū) = (i>15) ?p0` `S1 ?p1; S2 ?p2;`	$p_1 \Leftrightarrow (x_1 \wedge p_0)$ $p_2 \Leftrightarrow (\overline{x_1} \wedge p_0)$

`if(i>10) S4;`
`if(i>5) S5;`
	`p4(u) = (i>10) ?p3;`
	`p5(u) = (i>5) ?p3;`
	`S4 ?p4; S5 ?p5;`
		$p_4 \Leftrightarrow (x_2 \wedge p_3)$
		$p_5 \Leftrightarrow (x_3 \wedge p_3)$
		$x_2 \Rightarrow x_3$

Table 1: Extracting P-facts.

ify a predicate register, and a rich set of operations to define values for these predicate registers. Predicate registers are allocated like any other registers in the machine.

Each predicated operation is associated with an additional predicate register operand which specifies the predicate value under which the operation may have side effects on the processor state. The semantics of a predicated operation is defined as follows:

`Dest = <`*op*`> <`*sources*`> ? Pred` (1)

where `Dest`, `<`*op*`>`, `<`*sources*`>`, and `Pred` are respectively the destination register, the opcode, the required source operands, and the predicate register. The semantics of the predicated operation requires that the `Dest` register be overwritten by the new value produced by the operation only if the `Pred` register evaluates to true.

A rich set of operations to define values for predicate registers is implemented in the Playdoh architecture. The semantics of these `compare-to-predicate` operations is defined here as follows:

`Pout0(<`*type*`>), Pout1(<`*type*`>) =`
`(src0 <`*comp*`> src1) ? Pin` (2)

where `Pout0`, `Pout1` are the two destination predicate registers, `(src0 <`*comp*`> src1)` define a comparison, and `Pin` is the input predicate register. The value written in a destination register is a function of the predicate `<`*type*`>` associated with that destination register, the result of the comparison, and the input predicate. The supported predicate types are: `unconditional`, `conditional`, `or`, `and`, and their complements. Operations defining the predicates guarding `if-then-else` constructs typically use the `unconditional` type and its complement. More elaborate predicate types are required to evaluate several comparisons in parallel, as advocated in [20]. For example, the `or` type may be used to implement a wired-or function of several comparisons executed in parallel.

Based on the precise semantics of the predicate type used by a `compare-to-predicate` operation, we may construct a corresponding logically invariant expression that is guaranteed to hold regardless of the run time program values. Table 1 presents the P-facts extracted from two simple code fragments. The first example illustrates a simple predicated `if-then-else` construct similar to Example 1. Following the `unconditional` type semantics (`u` type), the first P-fact asserts that p_1 is true when both the comparison and the input predicate p_0 are true. Following the `negated unconditional` type semantics (`ū` type), the second P-fact asserts that p_2 is true when the comparison is false and the input predicate p_0 is true.

P-facts do not make any assumptions on the result of a comparison: P-facts merely assume that a comparison will evaluate either to true or false, resulting in the described effect on the predicate values. Therefore, comparisons are simply abstracted as logical variables; in our first example, the comparison `(i>15)` is abstracted as the variable x_1.

The second example in Table 1 illustrates two correlated predicates. We notice that the two conditions `(i>10)` and `(i>5)` are correlated, since `i` larger than 10 implies `i` larger than 5. The third P-fact expresses this correlation.

2.2 Live Range Analysis

In this section, we compute the predicate expression under which a virtual register is live. We consider live ranges defined and used by one or more operations within a single predicated block.

Consider a virtual register vr and operation i, producing a value of vr during the cycle interval tpe_i to tpl_i, namely from the earliest production time to the latest in which operation i may write back its results in vr. For example, the earliest production time of a divide operation corresponds to the earliest time where the divide operation may write its result back (e.g. divide by 1), and the latest production time corresponds to the latest possible time. Consider also operation j, consuming a value of vr during the cycle interval tse_j to tsl_j, namely from the earliest sampling time to the latest sampling time in which operation j may read the value of vr.

In a single unpredicated basic block, a value flows from operation i to operation j only if no intervening operation produces a value for the same virtual

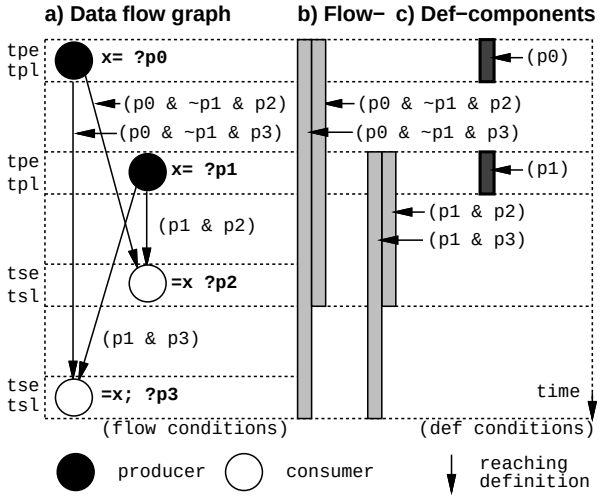

Figure 4: Flow and Def component of a live range.

register between operations i and j. In a predicated block however, the predicate under which operations execute must be taken into consideration. A value effectively flows from operation i to j if the predicates of operations i and j both evaluate to true and the predicates of all producer operations that could overwrite that value evaluate to false. The exact condition under which a value flows from operation i to operation j is referred to as the *flow condition* and is determined by the following conjunction of predicates:

$$p_i \wedge p_j \bigwedge_{\substack{tpe_k > tpl_i \\ tpl_k < tse_j}} \overline{p_k} \qquad (3)$$

where the p_k are the predicates of all the def-operations of vr that may potentially overwrite vr.

Figure 4a illustrates a dataflow graph with two producer and two consumer operations of virtual register `x`. The flow condition associated with each edge is also presented in this figure. For example, a value effectively flows from the first producer to the last consumer when the condition $p_0 \wedge \overline{p_1} \wedge p_3$ evaluates to true, i.e. when both the first producer and the last consumer operations execute, and the intervening producer operation does not.

We investigate now the contribution to the live range of vr generated by operations i and j. One contribution to the live range of vr is associated with the dataflow between operations i and j. This contribution is associated with the cycle range $[tpe_i, tsl_j]$. This contribution is referred to as the *flow component* and is live under the flow condition associated with the edge between operations i and j. Figure 4b illustrates the flow component for each of producer-consumer pair of `x`.

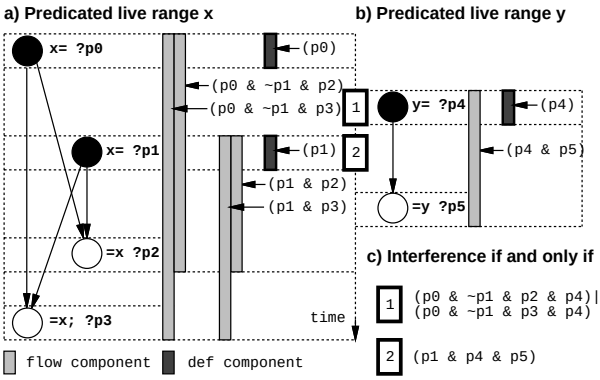

Figure 5: Determining interferences.

Another contribution is associated with the cycle range $[tpe_i, tpl_i]$. This contribution is referred to as the *def component* and is live under the predicate guarding operation i, referred to as the *def condition*. Note that the def components exist regardless of whether the produced values are used. Figure 4c illustrates the def components for producer of `x`.

2.3 Interference

In this section, we compute the interference between two predicated live ranges based on Chaitin's definition: two live ranges interfere if one of them is live at a definition point of the other [9]. This definition implies that overlapping def components do not interfere, provided no operations use these values. In the context of predicated blocks, this definition implies that simultaneous writes to a unique physical register should be tolerated by the hardware of the machine. In this paper, we assume that simultaneous writes are allowed, resulting in an unspecified value.

Figures 5a and 5b illustrate the predicated live ranges of virtual registers `x` and `y`, respectively. We notice that two def components, marked 1 and 2, each overlap flow components of the other virtual register. We use the flow and def conditions associated with each component to determine the predicate expression under which `x` and `y` may interfere, as shown in Figure 5c.

For example, the second def component of `x` interferes with the flow component of `y` (interval labeled 2) if there is a legal execution path in which expression $p_1 \wedge p_4 \wedge p_5$ evaluates to true. To see if this is possible, the compiler queries whether this expression is consistent with all of the P-facts, which are known to be satisfied by all legal paths. If the conjunction of the query expression with all the P-facts does not evaluate to false, the compiler must assume that there is some legal path that may satisfy the query expres-

sion, i.e. that there may be interference between `x` and `y`. Otherwise, the conjunction evaluates to false, indicating that there is no legal execution path that satisfies consistency of the query expression with all the P-facts, i.e. these particular components of `x` and `y` do not interfere.

In this paper, we consider query expressions that are of the form $p_{i_1} \wedge \ldots \wedge p_{i_{n'}} \wedge \overline{p_{i_{n'+1}}} \wedge \ldots \wedge \overline{p_{i_n}}$, i.e. a conjunction of predicate terms and negated predicate terms. Consistency between a query expression *expr* and the P-facts is checked by function **consistent**(*expr*). This function investigates whether the following equation can be proven:

$$\left(\bigwedge_{k=1}^{m} \text{P-fact}_k \right) \wedge (expr) \quad \neq \quad \text{false} \qquad (4)$$

returning true when Equation (4) is proven true, and false otherwise. This function can be implemented in several ways; in this work, we use a symbolic package that simplifies logic equations.

An interference algorithm based on Chaitin's definition of interference is presented in Figure 6. Note that this algorithm relies on the specific scheduling time of each operation. When interferences are computed before scheduling, we must make a conservative assumption on the scheduling time of each producer and consumer operation. Function **Live-at-def** investigates if any of the def components of `x` may interfere with any of the flow components of `y`. These tests are performed in function **Def-at-flow**. Consider `prod-x`, `prod-y`, and `cons-y` to be, respectively, a producer of `x`, a producer of `y`, and a consumer of `prod-y`. First, **Def-at-flow** determines if the def component of `prod-x` overlaps in time with the flow component associated with the producer-consumer pair (`prod-y`, `cons-y`). If it does, the flow condition `expr` is computed, as defined in Section 2.2, and the conjunction of the def condition and the flow condition is checked for consistency with the P-facts. When it is, the compiler must assume that `prod-x` may interfere with the value flowing from `prod-y` to `cons-y` and thus consider the live ranges `x` and `y` to interfere. If no def component of `x` or `y` is detected to interfere with a flow component of the other, then `vrx` and `vry` do not interfere and the edge between them is removed from the interference graph.

As presented in Figure 6, the algorithm has a computational complexity of $O(n^4)$, where n is the number of producers and consumers. However, the loop in line 6 can be computed in advance for each live range, thus reducing the complexity of the algorithm to $O(n^3)$. To reduce the cost of detecting live range

```
procedure Def-at-flow (Op prod-x, Op prod-y, Op cons-y)
{
1:    /* Determine if prod-x overwrite the value flowing
2:       from prod-y to cons-y. */
3:    if def-component(prod-x) overlaps with
4:       flow-component(prod-y, cons-y) then
5:       expr = predicate(prod-y) and predicate(cons-y);
6:       for each other producer of y strictly in
7:          interval(prod-y, cons-y): prod-y' do
8:          expr = expr and not predicate(prod-y');
9:       endfor
10:      return consistent(expr and predicate(prod-x));
11:   endif
12:   return false;
}
procedure Live-at-def (Live Range x, Live Range y)
{
13:   /* Determine if the producers of x interfere with
14:      live range y */
15:   for each producer of x: prod-x do
16:      for each producer of y: prod-y do
17:         for each consumer of prod-y: cons-y do
18:            if Def-at-flow(prod-x, prod-y, cons-y) then
19:               return true;
20:            endif
21:         endfor
22:      endfor
23:   endfor
24:   return false;
}
procedure Interference (Live Range x, Live Range y)
{
25:   /* Determine if live ranges x and y interfere */
26:   return Live-at-def(x, y) or Live-at-def(y, x);
}
```

Figure 6: Interference algorithm.

interferences, a regular interference algorithm can be used first, followed by a predicated analysis phase used to remove spurious interference edges.

3 Post-scheduling Bundling

For register allocators based on Chaitin's graph coloring framework [9][10], register allocation for predicated code can be achieved simply by using the refined interference graph instead of the conventional one. However, several register allocators depart from the graph coloring method [11][12] as graph coloring methods do not provide a notion of time that is particularly useful for the live ranges of a loop, which may cross the boundary of an iteration. Also, non-traditional constraints such as the one presented in [12] to support various code generation and hardware support schemes are difficult to express within the graph coloring framework.

To address the allocation of registers in loops with predicated code, we propose a technique that reduces the register requirements by allowing non-interfering virtual registers that overlap in time to share a common virtual register. Once virtual registers have been

suitably bundled, a traditional predicate-insensitive register allocator can be used.

In this section, we investigate this technique, referred to as bundling, after completion of the scheduling process. The key issue is to determine which virtual registers should be bundled together. We propose a parameterized heuristic that determines advantageous bundling.

First, the interferences between virtual registers of a predicated block are computed using the algorithm presented in Section 2.3. Second, the virtual registers are placed on the list of unprocessed virtual registers, ordered according to a heuristic. Then, the bundling algorithm selects the first virtual register on that list and assigns it to a suitable bundle. Several selection heuristics can be specified to identify a suitable bundle. A simple heuristic chooses the first bundle in which the current virtual register does not interfere with any of the virtual registers already in the bundle. A more elaborate heuristic would select the bundle that minimizes some cost function. Once a bundle is selected, the current virtual register is added to the bundle and the bundling heuristic selects the next unprocessed virtual register.

This algorithm thus maps virtual registers to bundles. The virtual registers in a bundle are guaranteed to be compatible (no pairwise interference), and can be reassigned to a unique virtual register. The remainder of this section discusses the ordering and the bundle selection heuristics. In this discussion, we define the *live range of a virtual register* as the interval $[tpe_i, tsl_j]$ where operations i and j are, respectively, the earliest producer and latest consumer of that virtual register. By extension, we define the *live range of a bundle* as the union of the live ranges of all virtual registers of that bundle. Furthermore, a virtual register is *compatible with a bundle* if it does not interfere with any of the virtual registers of that bundle.

3.1 Ordering Heuristics

Arranging the virtual registers in a suitable order is critical for heuristics that consider the bundling of registers only one virtual register at a time. We present here some of the ordering heuristics we investigated. Ordering heuristics can be combined to break the ties.

Start-Time: Arrange the virtual registers by increasing start-time of their live ranges. This heuristic tries to capture the time locality in the dependence graph.

Decreasing-Length: Arrange the virtual registers by decreasing length of their live ranges. Bundling virtual registers with long live ranges first is expected to result in a larger decrease in register requirements.

Decreasing-Interference: Arrange the virtual registers by decreasing number of interferences. Bundling virtual registers that interfere with the largest number of virtual registers first is expected to result in a larger decrease in register requirements.

3.2 Selection Heuristics

The selection heuristics place the current virtual register in a suitable bundle. When a selection heuristic tries every bundle unsuccessfully, a new bundle is created at the end of the bundle list and the current virtual register is added to it.

First-Compatible Bundle: Choose the first bundle for which the current virtual register is compatible and their live ranges intersect.

Last-Compatible Bundle: Choose the last bundle for which the current virtual register is compatible and their live ranges intersect. This heuristic presumes that more recently created bundles are more likely to accept new virtual registers than older bundles.

Best-Compatible Bundle: Choose the best bundle in which the current virtual register is compatible and their live ranges intersect. The best bundle is the one that maximizes the intersection of the current virtual register live range and the bundle live range (empty intersection not considered).

Best Bundle: Choose the best bundle in which the current virtual register is partially compatible and their live ranges intersect. This heuristic first computes for each bundle the set of virtual registers that interfere (*inter*) and the set the set of virtual registers that do not interfere (*comp*) with the current virtual register (*vr*). Then, it computes the number of cycles (*vr_in*) in which the live ranges of *comp* and the live range of *vr* intersect. It also computes the number of cycles (*vr_out*) in which the live ranges of *comp* and the live ranges of *inter* intersect. The best bundle is the one that maximizes *vr_in* - *vr_out* (values ≤ 0 not considered). If a bundle is selected, the virtual registers in *inter* are removed from the bundle and restacked on the list of unprocessed virtual registers. This selection heuristic is guaranteed to complete because virtual registers are restacked only if the overall benefit strictly increases (overall benefit is finite).

4 Pre-scheduling Bundling

We investigate in this section the process of bundling compatible live ranges before scheduling. The motivation for bundling before scheduling is that a register-sensitive scheduler may better schedule the operations of a bundle if the register-sensitive scheduler is aware of which virtual registers belong to a bundle.

With pre-scheduling bundling heuristics, interferences between virtual registers of a predicated block cannot be computed as precisely as in Section 2.3 since the scheduling time of the operations is unknown. Instead, we must rely on the earliest and latest feasible scheduling time of an operation [21] [22]. In our current implementation, the pre-scheduling bundling heuristics rely on a weaker form of interference where two virtual registers are considered to interfere if any pair of their producer operations do not execute under mutually disjoint predicates.

A more elaborate approach might alleviate this restriction by adding scheduling edges in the dependence graph to enable advantageous bundles. In this scheme, additional scheduling edges would supply some further temporal constraints on the subsequent scheduling. Some edges are guaranteed not to actually constrain the scheduler, as shown by Pinter [23]. Others, however, may constrain the scheduler. In the context of modulo scheduled loops, the danger of this approach resides in the fact that adding scheduling edges may, if not carefully applied, overly constrain the schedule of predicated blocks and may thus result in schedules with increased schedule length or decreased throughput. For this reason, we did not investigate this approach.

5 Measurements

We investigated the register requirements of the integer and floating point register file for a benchmark of loops obtained from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels (references, see [24]). Our input loops consist exclusively of innermost loops with no early exits, no procedure calls, and fewer than 30 basic blocks, as compiled by the Cydra 5 Fortran77 compiler [25]. The input to our scheduling algorithms consists of the Fortran77 compiler intermediate representation after load-store elimination, recurrence back-substitution, and IF-conversion. The input set consists of 1327 loops successfully modulo scheduled by the Cydra 5 Fortran77 compiler, with 1002 from the Perfect Club, 298 from SPEC, and 27 from the Livermore Fortran Kernels. From this input set, we generated a benchmark suite containing all the loops with at least a pair of mutually disjoint predicate values, resulting in a benchmark of 21 loops. In the current implementation, we did not attempt to correlate any of the IF-converted branches.

Table 2 summarizes the principal characteristics of the benchmark suite. The loops in the benchmark suite are significantly more complex than the average loop in the input set, with 51.4 operations and 65.3 register edges on average, versus 17.3 and 22.0 for the input set, respectively. As a result of this bias, the loops of the benchmark suite require an average of 39.3 registers versus 27.9 in the input set using the scheduling technique described in [13] which minimizes the register requirements for a modulo schedule. It contains loops with larger register requirements; consequently, decreasing the register requirements of the loops in the benchmark suite is crucial.

	Ops.	Reg. Edges	Sched. Edges	Pred.	Disjoint Pairs
min.	11	11	16	2	1
avg.	51.4	65.3	90.5	3.1	2.0
max.	107	138	218	9	13

Table 2: Characteristics of the benchmark (21 loops).

In this paper, we used modulo scheduling to software pipeline the loops of the benchmark suite. Modulo scheduling [1][3] has been shown to result in high performance code while requiring only modest compilation time [4] by restricting the scheduling space: it uses the same schedule for each iteration of a loop and it initiates successive iterations at a constant rate, i.e. one *Initiation Interval* (II clock cycles) apart. We present the register requirements, i.e. the maximum number of live values at any single cycle of the loop schedule, associated with two distinct schedulers:

Iterative Modulo Scheduler[4]: This modulo scheduler has been designed to deal efficiently with realistic machine models while producing schedules with near optimal steady state throughput. Experimental findings on a large benchmark suite show that this algorithm required the scheduling of only 59% more operations than with acyclic list scheduling and resulted in schedules that are optimal in II for 96% of the loops in the input set. In its current form, it does not attempt to minimize the register requirements of its schedules; however, the register requirements of its schedules may be reduced significantly by simple heuristics [24].

MinReg Stage Scheduler[13]: This stage scheduler minimizes the register requirements (without bundling) of a modulo schedule by shifting the operations by multiples of II cycles. This scheduler finds a schedule with the lowest achievable register requirements for the given machine, loop, and Modulo Reservation Table (MRT) [1]. In this paper, we use the MRT produced by the Iterative Modulo scheduler as input to this scheduler.

The machine model used in these experiments corresponds to the Cydra 5 machine [26]. This choice was motivated by the availability of quality code for this

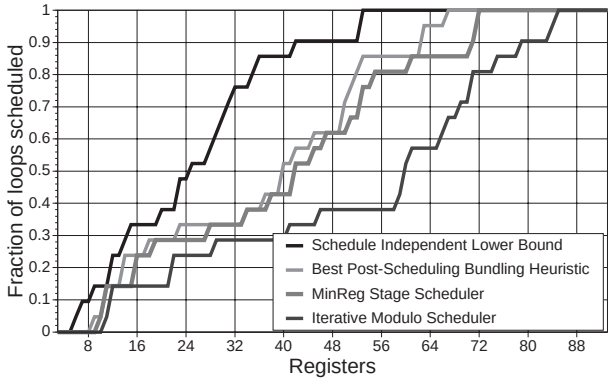

Figure 7: Register requirements (21 loops).

machine. In particular, the machine configuration is the one used in [4], with 7 functional units (2 memory port units, 2 address generation units, 1 FP adder unit, 1 FP multiplier unit, and 1 branch unit).

We first investigate the register requirements of the two schedulers and the register requirement of our best bundling heuristic[1]. Figure 7 presents the fraction of the loops in the benchmark suite that can be scheduled for a machine with any given number of registers without spilling and without increasing II.

The "Schedule Independent Lower Bound" curve corresponds to [22] and is representative of the register requirements of a machine without any resource conflicts, but does not support bundling, i.e. it ignores the possible decrease in register requirements due to the bundling of compatible virtual registers. Notice that none of the schedulers, including our best bundling heuristic, approaches the register requirements of the Schedule Independent Lower Bound. This large gap is partly explained by the strong simplification (no resource conflict) of the lower bound, a gap that is particularly apparent for large loops and negligible for small loops[27].

The second and third curves illustrate the decrease in register requirements due to the bundling of compatible virtual registers when applied to the schedules produced by the MinReg Stage Scheduler. Using our best bundling heuristic, the average register requirements decrease from 39.3 to 36.5 registers and the number of loops with no more than 64 registers increases from 18 to 20 of 21 loops. Note that this improvement is obtained only for schedules for the given MRT. The lowest curve illustrates the register requirements associated with the Iterative Modulo Scheduler and is indicative of the register requirements of a good modulo scheduler that currently does not attempt to minimize the register requirements.

To investigate the performance of the bundling heuristics, we introduce a lower bound on the register requirements:

Predicate-Sensitive Lower Bound: A lower bound on the register requirements for a given loop schedule and set of P-facts is obtained by bundling the virtual registers on a cycle-by-cycle basis. This clearly represents a lower bound, as normally virtual registers are allowed to be bundled on a register-by-register basis only.

We focus here on the decrease in register requirements achieved by a heuristic, normalized to the decrease in register requirements from no-bundling to the Predicate-Sensitive Lower Bound. The schedule used to investigate the post-scheduling bundling heuristics was obtained by the MinReg Stage Scheduler. In our benchmark suite, the average register requirements without bundling was 39.3 and the Predicate-Sensitive Lower Bound was 36.2. In the following paragraphs, a decrease in register requirements of 100% corresponds to a bundling heuristic that achieves the lower bound and 0% corresponds to a bundling heuristic that does not decrease the register requirements at all. All numbers are averaged over the 21 loops of the benchmark suite.

Cost Function	none	intersection
Bundle Selection Heuristic		
First-Compatible	72%	—
Last-Compatible	**75%**	—
Best-Compatible	—	72%
Best	—	**90%**

Table 3: Normalized decrease in register requirements.

We first investigate the performance of the bundle selection heuristics presented in Section 3.2. Table 3 presents the normalized decrease in register requirements associated with each of four selection heuristics, averaged over all loops and ordering heuristics. Note that the last two selection heuristics require a cost function to evaluate the intersections of the virtual register live range and the bundle live range. We may conclude from Table 3 that the Last-Compatible Bundle Heuristic is the best selection heuristic without cost function, and that Best Bundle Heuristic, which is the least sensitive to the ordering of the virtual registers, is the best selection heuristic overall.

The performance of the ordering heuristics described in Section 3.1 is summarized in Table 4. Each table entry corresponds to the normalized decrease in

[1]The best bundling heuristic corresponds to a post-scheduling bundling heuristic using the Length ordering (Interference tie-breaking) and the Best-Bundle selection.

Cost Function	none	intersection
Ordering Heuristic		
Reference	61%	76%
Start-Time	80%	85%
Length	**85%**	86%
Interference	76%	84%
Length+Interference	**85%**	**87%**

Table 4: Normalized decrease in register requirements.

register requirements achieved by an ordering heuristic, averaged over all loops and selection heuristics that share the same cost function. The first row, labeled "Reference" corresponds to the decrease in register requirements averaged over the initial and reverse orderings. Comparing the numbers of the other ordering heuristics to the reference numbers provides insight on the achieved decrease in register requirements due to the ordering heuristics. We conclude from Table 4 that the ordering heuristics do affect the register requirements, and that the largest decrease is generated by the Length+Interference ordering heuristic.

The individual results confirm the above findings, with the best results for the Best/Length-+Interference, Best/Length, and Best/Start-Time heuristics, decreasing the average register requirements from 39.3 to 36.52 registers, followed by Best/-Interference and Last-Compatible/Length, decreasing the average register requirements form 39.3 to 36.57 registers. All of these five post-scheduling bundling heuristics are less than 1% above the Predicate-Sensitive Lower Bound.

Figure 8 illustrates the number of additional registers needed by the bundling heuristics and other scheduling algorithms when compared to the Predicate-Sensitive Lower Bound. From top to bottom, the curves represent the additional registers needed by two post-scheduling bundling heuristics, a pre-scheduling heuristic, the MinReg Stage Scheduler, and the Iterative Modulo Scheduler.

Notice that the two post-scheduling bundling heuristics reduce the register requirements significantly, with all loops scheduled using no more than 2 additional registers. The decrease in register requirements of the pre-scheduling bundling heuristic is significantly lower, with all loops scheduled using no more than 8 additional registers. This disappointing performance is due, we believe, to the fact that our pre-scheduling bundling approach uses a more conservative definition of non-interfering live range. A more elaborate approach could alleviate this situation by adding scheduling edges in the dependence graph to enable advantageous bundling, at the risk of increasing the schedule length or decreasing the throughput

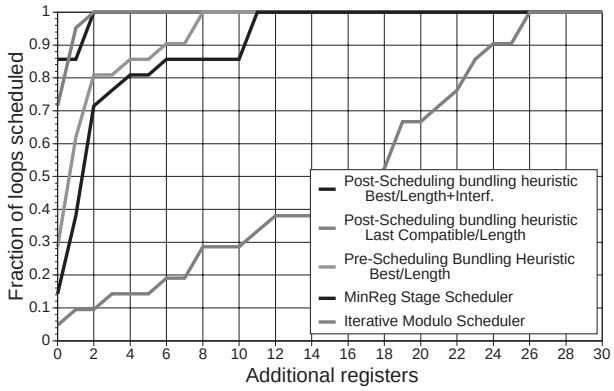

Figure 8: Additional register requirements of the pre- and post-scheduling bundling heuristics.

of the schedule. In its current form however, experimental evidence suggests that the bundling of compatible virtual registers, without adding scheduling edges in the dependence graph, is more successful as a post-scheduling optimization phase than as a pre-scheduling phase.

6 Conclusion

In this paper, we discussed the need for better analysis of predicated code by showing an example of code where the register requirements increase when several basic blocks are merged into a single predicated block. To remedy this increase in register requirements, we proposed a framework that analyzes predicated codes by extracting P-facts from the predicated block, analyzing the predicate expressions associated with live ranges, and refining Chaitin's interference definition to account for predicate expressions. This framework generalizes to arbitrary predication schemes and may capture correlations between IF-converted branches.

Register allocators based on the graph coloring framework can simply use the refined interference graph to perform register allocation on predicated code. For register allocators based on interval graphs, or others, we propose a new technique that bundles non-interfering virtual registers to lower the register requirements of a predicated block. Bundling may be performed after scheduling process to capitalizes on the availability of the scheduling time of each operation to determine precisely the interferences among virtual registers. Alternatively, bundling before scheduling capitalizes on a register-sensitive scheduler which may better schedule the operations associated with a pre-bundled virtual register.

Preliminary results indicate that using our best post-scheduling bundling heuristic, we decreased the average register requirements from 39.3 to 36.5 registers, 1% above the Predicate-Sensitive Lower Bound,

and we increased from 18 to 20 the number of loops with no more than 64 registers, in a benchmark suite of 21 modulo scheduled loops extracted from the Perfect Club, SPEC-89, and the Livermore Kernels. Our results also suggest that bundling compatible virtual registers, without adding scheduling edges in the dependence graph, is more successful when applied after rather than before scheduling.

Acknowledgements

This work was supported in part by the Office of Naval Research under grant number N00014-93-1-0163 and by Hewlett-Packard. The authors would like to thank Santosh Abraham, Richard Johnson, Scott Mahlke, Bob Rau, and Michael Schlansker for their many useful suggestions and for providing the input data set. We appreciate the input of the referees which significantly improved the quality of the paper.

References

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

[2] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using hyperblocks. *MICRO*, pages 45–54, Dec. 1992.

[3] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: Compiler techniques and architecture support. *ISCA*, pages 131–139, 1982.

[4] B. R. Rau. Iterative Modulo Scheduling: An algorithm for software pipelining loops. *MICRO*, pages 63–74, Nov. 1994.

[5] J. R. Allen, Ken Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, Jan. 1983.

[6] J. C. H. Park M. S. Schlansker. On predicated execution. Technical Report HPL-91-58, HP Laboratories, May 1991.

[7] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register requirements of pipelined processors. *ICS*, pages 260–271, July 1992.

[8] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. *ICS*, pages 424–433, July 1995.

[9] G. J. Chaitin. Register allocation and spilling via graph coloring. *Symp. on Compiler Construction*, pages 98–105, June 1982.

[10] Preston Briggs, Keith D. Cooper, Ken Kennedy, and L. Torczon. Coloring heuristics for register allocation. *PLDI*, 24(7):275–284, June 1989.

[11] L.J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *Intl. Conf. on Compiler Construction, Springer*, pages 176–191, 1992.

[12] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. *PLDI*, pages 283–299, June 1992.

[13] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. *MICRO*, pages 75–84, Nov. 1994.

[14] N. J. Warter, G. E. Haab, and J. W. Bockhaus. Enhanced Modulo Scheduling for loops with conditional branches. *MICRO*, pages 170–179, Dec. 1992.

[15] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. *PLDI*, pages 257–271, 1990.

[16] P. Tu and D Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. *ICS*, pages 414–423, July 1995.

[17] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1988.

[18] R. C. Johnson. Personal communication. June 1995.

[19] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, HP Laboratories, Feb. 1994.

[20] M. S. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. *MICRO*, pages 40–51, Nov. 1994.

[21] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocs. *ICS*, pages 442–452, 1988.

[22] R. A. Huff. Lifetime-sensitive modulo scheduling. *PLDI*, pages 258–267, June 1993.

[23] S. S. Pinter. Register allocation with instruction scheduling: a new approach. *PLDI*, pages 248–257, June 1993.

[24] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. *MICRO*, pages 180–191, Nov. 1995.

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

[26] G. R. Beck et al. The Cydra 5 mini-supercomputer: Architecture and implementation. In *The J. of Supercomputing*, volume 7, pages 143–180, 1993.

[27] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Optimum modulo schedules for minimum register requirements. *ICS*, pages 31–40, July 1995.