A Reduced Multipipeline Machine Description that Preserves Scheduling Constraints

Alexandre E. Eichenberger and Edward S. Davidson

Advanced Computer Architecture Laboratory
EECS Department, University of Michigan
1301 Beal Ave, Ann Arbor, MI 48109-2122
alexe,davidson@eecs.umich.edu

Abstract

High performance compilers increasingly rely on accurate modeling of the machine resources to efficiently exploit the instruction level parallelism of an application. In this paper, we propose a reduced machine description that results in faster detection of resource contentions while preserving the scheduling constraints present in the original machine description. The proposed approach reduces a machine description in an automated, error-free, and efficient fashion. Moreover, it fully supports schedulers that backtrack and process operations in arbitrary order. Reduced descriptions for the DEC Alpha 21064, MIPS R3000/R3010, and Cydra 5 result in 4 to 7 times faster detection of resource contentions and require 22 to 90% of the memory storage used by the original machine descriptions. Precise measurement for the Cydra 5 indicates that reducing the machine description results in a 2.9 times faster contention query module.

1 Introduction

Current compilers for VLIW and superscalar machines focus on exploiting more of the inherent parallelism in an application in order to obtain higher performance. Fine grain schedulers are a critical element in efficiently exploiting instruction level parallelism and a significant body of research has sought more effective scheduling algorithms. Several new directions have been explored: schedulers may not schedule operations in cycle order, focusing initially on operations along critical paths [1][2][3][4][5][6], they may backtrack to reverse poor scheduling decisions [1][2][3][4][7], and they may hide long latencies by speculating operations across branches and basic blocks [1][6][7][8][9][10].

High performance compilers have also used precisely detailed machine models [1][3][7][11][12][13] to better utilize the machine resources of current processors with increasingly wider issue mechanisms, deeper pipelines, and more heterogeneous functional units. Precise modeling of machine resources is critical to avoid resource contentions that may stall some of the pipelines or, in the absence of hardware interlocks, corrupt some of the results. Resource modeling has to cope with rapidly changing processor models while controlling development cost by reusing existing compiler technology.

To meet these challenges, compilers have increasingly relied on a resource modeling utility, separated from the rest of the compiler, that can quickly answer the following query: "Given a target machine and a partial schedule, can I place this additional operation in this cycle without resource contention?" Typically, this functionality has been provided by a *contention query module* that processes the machine description of a target machine, generates an internal representation of the resource requirements, and provides for a querying mechanism [1][3][7][11][12][13]. The IMPACT compiler, for example, implemented such a module [12] to produce high performance schedules for a wide range of machines, from existing architectures such as X86, PA-RISC, and SPARC to research architectures such as PlayDoh [14].

With the recent emphasis on exploiting instruction level parallelism, compile time is increasingly spent in the contention query module as several cycles of a schedule, possibly in several basic blocks [9][10], are queried per operation in order to achieve good schedules. Optimizing contention query modules therefore has a significant impact on the overall performance of a compiler, as queries are issued in the innermost loop of the scheduler. This optimizing issue has recently been addressed in several papers [15][16][17], but these either overrestrict the manner in which operations are placed or approximate the actual resource requirements of a schedule.

In this paper, we propose a reduced machine description that results in significantly faster detection of resource contentions while exactly preserving the scheduling constraints present in the original machine description. The reduced machine description is expressed using reservation tables that determine the resource usage for each operation. We demonstrate how to derive a reduced machine description for a given target machine and present several examples illustrating the effectiveness of our approach.

The proposed approach fully supports *unrestricted scheduling models* where operations can be scheduled in arbitrary order and prior scheduling decisions can be reversed. Unrestricted scheduling is essential to accommodate the elaborate scheduling techniques used by today's high performance compilers. The Cydra 5 compiler, for example, uses an operation-driven scheduler that reduces the schedule length of a basic block by scheduling operations along the critical path first [1]. Operation-driven schedulers consider operations in topological order, not in order of monotonically increasing (or decreasing) schedule time. Also, the Cydra 5 and IMPACT compilers, as well as others, use software pipelining techniques to achieve loop schedules with high throughput [1][2][10]. Software pipelining schedulers do not consider operations in topological order as, in general, no topological order is defined in dependence graphs with loop-carried dependences. Moreover, experimental results indicate that software pipelined loops can achieve higher throughput in less compilation time when some limited number of scheduling decisions can be reversed, as shown by Rau [3], and used in numerous compilers [1][2][3][4][18]. The Multiflow

To appear in *Proceedings of the Conference on Programming Language Design and Implementation*, May 96

compiler also uses a backtracking mechanism to improve scalar code schedules [7].

The proposed approach also precisely handles *basic block boundary conditions*, i.e. the dangling resource requirements from predecessor basic blocks. In general, the resource requirements at the beginning of a basic block consist of the union of all the resource requirements dangling from predecessor basic blocks. Handling boundary conditions is even more important for high performance compilers that hide operation latencies by (speculatively) moving operations across branches and basic blocks [1][6][7][8][9][10]. Both the Cydra 5 and the Multiflow compilers, for example, use scheduling algorithms that handle dangling resource requirements [1][7].

Currently, most compilers rely on machine descriptions that have been manually reduced using error prone ad-hoc methods. To avoid errors or to reduce the machine description more easily or further, conservative assumptions may be employed. Thus, the reduced machine description may prohibit certain operation sequences that cause no contentions on the target machine. Furthermore, high performance compilers are often developed in parallel with micro-architecture development during which resource requirements often change. Manually reducing the machine description must then be carried out several times, introducing more potential for errors, suboptimal solutions, and increased development and maintenance cost. Using our approach, the resource requirements can be expressed in terms close to the actual hardware structure of the target machine and the reduced machine description used by the compiler is generated in an error-free and automated fashion.

Experiments with the DEC Alpha 21064 [19], MIPS R3000/-R3010 [20], and Cydra 5 [21] machines indicate 4 to 7 times faster contention queries and require 22 to 90% of the memory storage used by the original machine descriptions. These improvements are obtained by using highly reduced machine descriptions instead of the original or manually optimized machine descriptions. When using our reduced machine description during Cydra 5 compilation, precise measurements with a state-of-the-art scheduler and an efficient contention query module for a benchmark of 1327 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels resulted in a 2.9 times faster contention query module.

In this paper, we present related work in Section 2 and an introductory example in Section 3. Algorithms to construct reduced machines are developed in Sections 4 and 5. Reduced machine examples are presented in Section 6. A contention query module is developed in Section 7 and its performance is investigated in Section 8. We present our conclusions in Section 9.

2 Related Work

Resource contention in multipipeline scheduling may be based directly on reservation tables, or on the forbidden latency sets or contention-recognizing state machines derived from them, as introduced by Davidson *et al* [22]. Traditionally, reservation tables contain much redundant information that consumes memory and increases query response time. As a result, recent advances favor finite-state automata approaches. In this paper, however, we propose a reduced reservation table approach that eliminates much of the redundancy and does not suffer from the limitations of the automata approaches, as detailed below.

Proebsting and Fraser [15] as well as Müller [16] proposed a contention query module using a finite-state automaton that recognizes all contention-free schedules. The technique proposed by Proebsting and Fraser directly results in minimal finite-state automata [15]. This approach was recently extended for unrestricted scheduling models by Bala and Rubin using a forward and reverse pair of automata [17]. In their approach, operations considered in order of monotonically increasing (or decreasing) schedule time are quickly scheduled using a forward automaton. Additional operations are then inserted in the schedule in cycles recognized as contention-free by the forward and reverse automata. Because an inserted operation introduces additional resource requirements, these additional requirements must be propagated in adjacent cycles, i.e. the state of scheduled operations in adjacent cycles must be updated in both the forward and reverse automata. Their approach also addresses the handling of basic block boundary conditions at the cost of potentially introducing up to $O(s^2)$ new states in the automata, where s is the number of cycle-advancing states in the original automata [17].

The principal advantage of automaton-based approaches is that a single table lookup can determine the next contention-free cycle. A potential problem of this approach, however, is the size of these automata. This issue is addressed in the literature in three ways. First, operations of a target machine can be combined into classes of operations that have compatible resource contentions [15]. Second, large automata can be factored into sets of smaller ones [16][17], reducing the size of the automata, but increasing the number of table lookups necessary to process a contention query. Third, the number of additional states introduced in the automata to handle boundary conditions can be reduced [17], at the cost of making conservative approximations. However, new experimental evidence that was gathered by Bala and Rubin [23] for the Alpha, PA_RISC, and MIPS families indicates that, in practice, no additional state is introduced to precisely handle boundary conditions, when minimal finite-state automata are constructed.

Another problem arises when supporting unrestricted scheduling models, since the state of the forward and reverse automata must be saved for each scheduled operation. In addition to storing the descriptions of the two automata, two states per operation must be stored, which may result in a large memory overhead, especially for wide-issue machines. Supporting unrestricted scheduling models also requires the consistency of the stored state to be maintained when scheduling additional operations [17], as inserted operations introduce additional resource requirements. Thus, handling unrestricted scheduling models introduces both memory and computation overhead that is similar to, or may exceed, the overhead incurred by the reservation table approach. A more detailed comparison is provided in Sections 6 and 8.

Finally, some backtracking schedulers may schedule an operation even though it may result in resource contentions, in which case the earlier scheduled operations that conflict are then unscheduled. For example, this mechanism is a key component of the scheduling algorithm proposed by Rau to generate high performance software-pipelined loop schedules at a low level of computational complexity [3]. Using reservation tables, this mechanism can be easily implemented by keeping a mapping from each reserved resource to the scheduled operation that consumes it. Implementing this mechanism with finite-state automata appears to be more difficult, as it corresponds to modifying the path in the forward and backward automata so that the new operation is accepted by the automata in the desired cycle.

3 Reducing a Machine Description

In this section, we illustrate the three-step process of constructing a synthesized machine, resulting in reduced numbers of resources and resource usages while exactly preserving the scheduling constraints due to resource contentions in the target machine.

The machines handled here may have arbitrary resource requirements, including alternative resource usages, but must have latencies that are known at compile time. Alternative resource usages, e.g. operation X using either resource 0 or 1 interchangeably, require some preprocessing. In this case, we would replace operation X in the original machine description with two new operations, $X0$

a) Machine description (reservation tables)

Operation A (resources × cycles):

resources	0	1	2
0	A		
1		A	
2			A
3			
4			

Operation B (resources × cycles):

resources	0	1	2	3	4	5	6	7
0								
1	B							
2		B						
3			B	B	B	B		
4							B	B

(usage sets)

A0={0} B0=∅

A1={1} B1={0}

A2={2} B2={1}

A3= ∅ B3={2,3,4,5}

A4= ∅ B4={6,7}

b) Forbidden latency set matrix

ops. \ operations	A	B
A	{0}	{-1}
B	{1}	{-3,-2,-1, 0,1,2,3}

d) Reduced machine description (reservation tables)

res.	0	1
0′		A
1′		

res.	0	1	2	3
0′	B			
1′	B	B		B

(usage sets)

A0′={1} B0′={0}

A1′=∅ B1′={0,1,3}

c) Generating set of maximal resources

res.	0	1	2	3
0′	B	A		
1′	B	B	B	B

Figure 1: Reducing a machine description.

and $X1$, identical to X but with operation $X0$ exclusively using resource 0 and operation $X1$ exclusively using resource 1. In general, new operations are created until all alternative resource usages present in the original machine description are removed. The operations generated from a unique operation in the original description are subsequently referred to as *alternative* operations, e.g. $X0$ and $X1$ are the alternative operations of X.

We begin with a given machine description consisting of a set of *reservation tables*, one per operation, that expresses the resource requirements of each operation in terms close to the actual hardware structure of the target machine. The rows of a reservation table correspond to distinct resources of the target machine and its columns correspond to the cycles in which resources are used relative to the issue time of the corresponding operation. An X entry in row i/column j is made in the reservation table associated with operation X if there is a *usage* of resource i in cycle j by operation X, i.e. if resource i is reserved for exclusive use during cycle j by operation X.

Figure 1a shows the reservation tables of a hypothetical machine with 2 operations (A and B) and 5 distinct resources $(0, \ldots, 4)$. Operation A is representative of the resource requirements of a fully pipelined functional unit. Operation B is representative of the resource requirements of a partially-pipelined functional unit, where resource 3 may correspond to a multiply stage used for 4 consecutive cycles and resource 4 may correspond to a rounding mode stage used for 2 consecutive cycles. Although this hypothetical machine was constructed to concisely illustrate our methodology, it is representative of some of the resource usage patterns found in our benchmark examples (see Figure 4 for example).

Step 1. For each pair of operations, we extract from the corresponding pair of reservation tables of the target machine the set of *forbidden latencies*, i.e. the set of initiation intervals for which a resource contention occurs between the two operations. Visually, the set of forbidden latencies between operations X and Y is obtained by overlapping their reservation tables, and searching for all initiation intervals that result in simultaneous use of one or more shared resources.

To formalize the definition of forbidden latencies, we define the *usage set* X_i as the set of cycles in which operation X reserves resource i for exclusive use. Figure 1a illustrates the usage sets of our example machine. For example, usage set B_3 is equal to $\{2, 3, 4, 5\}$, as operation B uses resource 3 in cycle 2, 3, 4, and 5. Two operations, X and Y scheduled at times t_X and t_Y, respectively, conflict if and only if there is some resource, i, and elements $x \in X_i$ and $y \in Y_i$ such that $t_X + x = t_Y + y$, i.e. both operations use resource i simultaneously. When such a conflict occurs, operation X cannot be scheduled $(y \Leftrightarrow x)$ cycles after operation Y. We thus obtain $F_{X,Y} = \{f \mid$ operation X cannot be scheduled f cycles after operation $Y\}$, i.e.

$$F_{X,Y} = \{(y \Leftrightarrow x) \mid \text{for all } i \in Q,\ x \in X_i,\ y \in Y_i\ \} \quad (1)$$

Equation (1) defines a matrix of forbidden latency sets for all pairs of operations, where $F_{X,Y}$ is the set in row X, column Y of the matrix. Figure 1b illustrates this matrix computed for our example machine. While these sets are computed for each operation of the target machine, we need list these sets only for each operation class, as presented by Proebsting and Fraser [15]. In general, two operations belong to the same *operation class* if they have the same sets of forbidden latencies, i.e. operations X and Y belong to the same class if $F_{X,Z} = F_{Y,Z}$ and $F_{Z,X} = F_{Z,Y}$ for each operation Z of the target machine. Note two properties of the forbidden latency matrix. First, operation X necessarily conflicts with itself for an initiation interval of 0 (i.e. $0 \in F_{X,X}$) if it uses any resources. Second, operation X cannot be scheduled f cycles *after* operation Y if and only if Y cannot be scheduled f cycles *before* X (i.e. $f \in F_{X,Y} \Leftrightarrow \Leftrightarrow f \in F_{Y,X}$).

Formal Problem Definition. Generate a reduced machine description, i.e. one with a reduced number of resources and resource usages in its reservation tables, which for each operation pair produces exactly the same forbidden latencies as the target machine. One of several objective functions (e.g. the number of resources or resource usages) may be minimized, depending on the desired internal representation. Querying for resource contentions using either the original or reduced machine descriptions yields the same answer, as both descriptions enforce the same forbidden latencies. Note that to schedule a given machine, we need know only whether, not where, resource conflicts occur. Thus we are free to represent the machine with any set of synthetic resources and usages that preserves the forbidden latencies.

Step 2. We build the *generating set of maximal resources* which is defined as a set of resources that contains all maximal resources

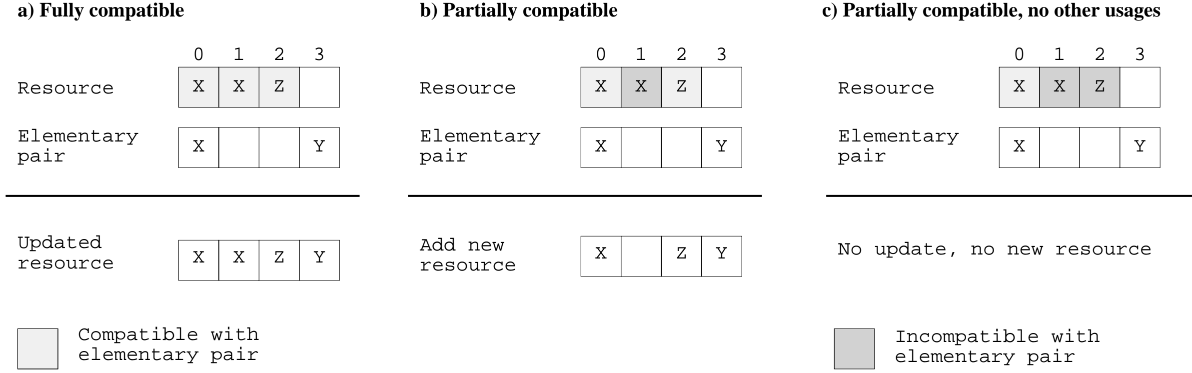


Figure 2: Three situations when adding an elementary pair to a resource.

associated with the target machine [24]. A *maximal resource* is defined as a synthesized resource such that (a) every forbidden latency generated by this resource is forbidden in the target machine and (b) no additional usage by any operation can be added to this resource without generating a forbidden latency that is not forbidden in the target machine. Finally since shifting all usages in a resource by some constant in time has no effect on the forbidden latencies, we only consider maximal resources that have their earliest usage in cycle 0.

From the construction in Theorem 1 below, it follows that there are only two maximal resources in our example machine, as shown Figure 1c. The first resource, resource $0'$, is a maximal resource that generates $1 \in F_{B,A}$; it also includes forbidden latencies: $0 \in F_{A,A}$, $0 \in F_{B,B}$, and $\Leftrightarrow 1 \in F_{A,B}$. Note that no other usages of A or B can be added to resource $0'$, as they would necessarily introduce forbidden latencies not present in the forbidden latency matrix of our example machine. Similarly the second maximal resource, resource $1'$, generates $F_{B,B}$ which includes all the remaining forbidden latencies and no other usages can be added. Note that the forbidden latency sets of the maximal resources need not be disjoint, e.g. $0 \in F_{B,B}$ is generated by both maximal rows here.

The maximal resources are interesting because any reservation table that generates the same forbidden latency matrix can be constructed from subsets of maximal resources, where the selected usage set in each resource may be translated by some number of cycles. As a result, we can use a (possibly empty) subset of the usages of each maximal resource to cover all the forbidden latencies of a target machine with the fewest number of (nonempty) synthesized resources.

Step 3. We select a subset of the maximal resources and their resource usages which covers all the forbidden latencies in the forbidden latency matrix. The selection heuristic minimizes an objective function that varies as a function of the desired internal representation. For our example machine, if the objective is to minimize the number of synthesized resources, we must select both resources $0'$ and $1'$, since resource $0'$ is the only resource covering $0 \in F_{A,A}$ and resource $1'$ is the only resource covering $3 \in F_{B,B}$. However, if the objective function is to minimize the number of resource usages, we may also remove the second or third usage of B in resource $1'$, shown Figure 1c, since the three remaining usages of B are sufficient to generate all the forbidden latencies in $F_{B,B}$.

Comparing Figure 1a to Figure 1d, we can appreciate the benefit of reducing the reservation tables of a target machine. First, the reduced machine description reduces the number of resources from 5 to 2, thus potentially decreasing the memory requirements needed to store the reserved resources of a schedule. Second, the number of resource usages decreases from 3 to 1 for operation A, and from 8 to 4 for operation B. If detecting resource contentions is linear in the number of usages, the reduced machine description results in significantly faster queries. We refine this view in Section 7.

4 Building Generating Sets of Maximal Resources

In this section, we present an algorithm that constructs the generating set of maximal resources, a set that contains all the maximal resources of a target machine. The algorithm builds the maximal resources incrementally, adding usages to current resources and creating new resources when appropriate. It is an efficient algorithm that does not backtrack; however, it may produce some submaximal resources in addition to all the maximal resources. A mechanism to remove submaximal resources as well as redundant maximal resources is discussed in Section 5.

We do not consider here the forbidden latencies directly but employ elementary pairs of usages that generate them. We define the *elementary pair* associated with forbidden latencies $f \in F_{X,Y}$ as a usage by operation X in cycle 0 and a usage by operation Y in cycle f. We also define a compatibility relation between an elementary pair and the usage of a resource. Elementary pair, p, with usages u_0 and u_1 is *compatible* with a usage, u, in resource q if the (nonnegative) forbidden latencies generated by u, u_0 and u, u_1 are both in the forbidden latency matrix.

Note that any resource with n usages can be constructed from $n \Leftrightarrow 1$ elementary pairs, namely by (a) shifting all its usages by some constant so that its first usage occurs in column 0, (b) choosing one usage in column 0 and constructing a set of elementary pairs consisting of this usage together with each other usage in the resource, and (c) placing all these pairs, which are known to be compatible since they exist together in the given resource, together in one resource which is then the same as the given resource.

Algorithm 1 (Building Generating Sets of Maximal Resources) The first step assigns the initial generating set to the empty set and builds a list of elementary pairs associated with the forbidden latencies of the target machine. We exclude here the elementary pairs associated with the negative forbidden latencies ($f < 0$) since they are redundant (i.e. $f \in F_{X,Y} \Leftrightarrow \Leftrightarrow f \in F_{Y,X}$). We also exclude the elementary pairs associated with the 0 self-contention latencies ($0 \in F_{X,X}$), which are processed as a special case at the end of the algorithm.

The second step attempts to add the first elementary pair on the list to each of the resources of the current generating set in turn. One of two cases will occur when attempting to add elementary pair p to resource q:

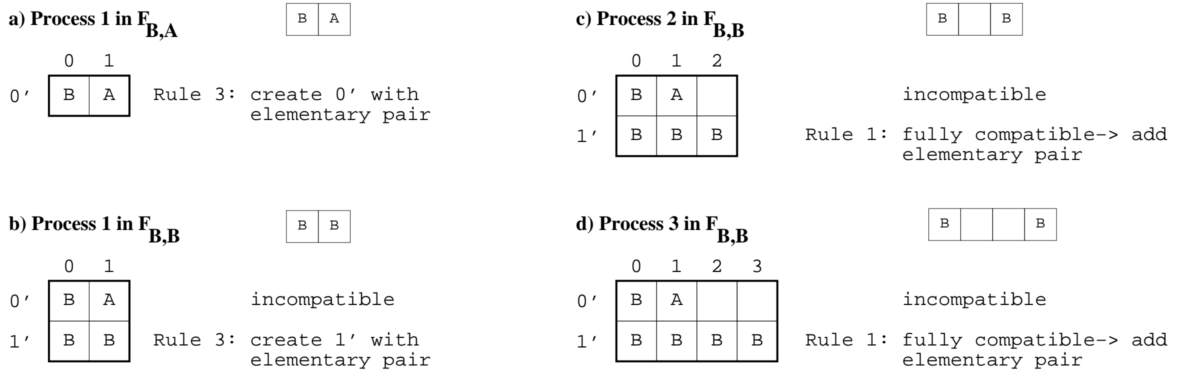


Figure 3: Building the generating set for our example machine.

- Elementary pair p is *fully compatible* with resource q, i.e. with all usages in q. In this case, we add the usages of p to resource q. This process is referred to as *Rule 1*, and is illustrated in Figure 2a for the elementary pair associated with the forbidden latency $3 \in F_{X,Y}$.

- Elementary pair p is *partially compatible* with resource q, i.e. it is incompatible with at least one usage in q. We may not simply add the usages of p to resource q, as this would generate some forbidden latencies not present in the forbidden latency matrix. Instead, we leave resource q in the the current generating set unchanged and consider adding a new resource, consisting of the usages of p and each of the usages of resource q that are compatible with p. If this new resource is not simply p itself with no other usages, then it is added to the current generating set; otherwise it is discarded. This process is referred to as *Rule 2*, and is depicted in Figure 2b and 2c.

After applying Rule 1 or Rule 2 for p with each resource in the current set, and placing the corresponding updated and new resources in the current set, we add elementary pair p itself as a new resource if the two usages of p are not yet found together in any single resource in the set. This is referred to as *Rule 3*.

Elementary pair p is then removed from the list of elementary pairs and the second step of the algorithm is executed repeatedly until the pair list is empty.

For each operation, X, (if any) that has $0 \in F_{X,X}$ as its only forbidden latency, the third step adds a new maximal resource that consists of one usage, by operation X in cycle 0. This is referred to as *Rule 4*. All other operations, Y, must be part of at least one elementary pair, and $0 \in F_{Y,Y}$ was forbidden automatically when the first elementary pair with a Y usage was processed.

Theorem 1 *Building the generating set of maximal resources as described in Algorithm 1 produces resources that forbid only those latencies that are forbidden in the target machine. Furthermore, the final generating set includes all maximal resources of the target machine.*

Proof. Rules 1, 2, 3, and 4 never place a usage in a resource unless it is compatible with each other usage in that resource, i.e. no resource in the current (and hence final) generating set forbids any latency not forbidden in the target machine.

We prove the second part of Theorem 1 by contradiction. Suppose there is a maximal resource not in the generating set. We shift its resource usages so that its earliest usage occurs in cycle 0 and call that maximum resource q. If q has a single usage, by operation X, either Rule 4 applies and thus q is present in the final generating set or Rule 4 does not apply and thus q is not a maximal resource since at least one resource (with two or more usages) in the final generating set has a usage associated with operation X.

Otherwise, let q have n usages, u_1 to u_n, with u_1 in cycle 0. We refer to the elementary pairs containing usage u_1 and each of the other $n \Leftrightarrow 1$ usages as p_2 to p_n. Without loss of generality, we may assume that the elementary pairs p_2 to p_n are numbered in the order in which they are processed by Algorithm 1.

After p_2 is processed (by Rules 1, 2, and 3), its corresponding usages u_1 and u_2 are present in at least one resource, q_{12}, of the current generating set. Other elementary pairs are then processed, possibly adding usages to q_{12}, but never removing any.

Eventually, the algorithm will process p_3. From resource q, we know that the usages of p_2 are both compatible with p_3. Hence Rule 1 or 2 will result in a resource q_{123} containing usages u_1, u_2, u_3, and possibly others; Rule 3 will not apply. Repeating this process with all the remaining elementary pairs, including p_4 through p_n, we obtain resource $q_{12\ldots n}$ containing all usages u_1 through u_n, and possibly others. Thus the set of usages in resource q is a subset of the usages in $q_{12\ldots n}$ which is in the final generating set. Hence q is either not maximal or is present in the final generating set, which contradicts the initial assumption. □

Figure 3 illustrates the algorithm, step by step, for our example machine. The algorithm processes the four nonnegative forbidden latencies (excluding the 0 self-contention latencies) $1 \in F_{B,A}$, $1 \in F_{B,B}$, $2 \in F_{B,B}$, and $3 \in F_{B,B}$, in that order. The 0 self-contention latencies are included automatically without special single usage resources in this example. The generating sets are shown at each step, in Figures 3a, 3b, 3c, and 3d, respectively. The rule applied to each resource is also indicated to the right of each resource.

5 Selecting Synthesized Resources and Resources Usages

Once the generating set of maximal resources has been computed, we select a subset of these resources and their usages that covers all the forbidden latencies in the forbidden latency matrix. The selection heuristic attempts to minimize an objective function that varies as a function of the desired internal representation for partial schedules. In this paper, we consider the two following internal representations.

Discrete-Representation. This representation uses a *reserved table* with one row per resource and one column per schedule cycle. Each entry contains a flag indicating whether the corresponding resource has been reserved by an operation in the current partial schedule. Entries may contain additional fields, such as a field identifying the operation that consumes the corresponding resource,

Representation:	original	discrete	bitvectors	(32 bits)	(64 bits)
Objective function minimizing:	–	*res-uses*	*1-cycle-word uses*	*2-cycle-word uses*	*4-cycle-word uses*
number of resources	56	15	15	15	15
average resource usages / operation	18.2	8.3	8.8	10.1	11.4
average word usages / operation	13.2	6.7	6.2	4.7	3.3

Table 1: Results for the Cydra 5: 52 operation classes, 10223 forbidden latencies (all ≤ 41).

Representation:	original	discrete	bitvectors	(32 bits)	(64 bits)
Objective function minimizing:	–	*res-uses*	*1-cycle-word uses*	*3-cycle-word uses*	*7-cycle-word uses*
number of resources	39	9	9	9	9
average resource usages / operation	9.4	2.9	2.9	3.6	4.2
average word usages / operation	7.5	2.6	2.6	2.0	1.5

Table 2: Results for a subset of the Cydra 5: 12 operation classes, 166 forbidden latencies (all ≤ 21).

as used in the Iterative Modulo Scheduler algorithm [3], or a field identifying the predicate under which the resource is reserved, as proposed in the Enhanced Modulo Scheduling scheme [2]. Because the number of entries tested to detect resource contentions is proportional to the number of resource usages over all reduced reservation tables, the primary objective of the selection heuristic is to *minimize* the number of resource usages in the reduced machine description. This objective function is referred to as *res-uses*.

Bitvector-representation. This representation extracts the flag bits of the discrete representation and packs them into one bitvector per schedule cycle (and reduced reservation tables are represented likewise). If k bitvectors can be packed per memory word, the number of words tested to detect resource contentions is reduced, as are the memory requirements for storing the reserved table. The primary objective of the selection heuristic is now to *minimize* the number of words that need to be tested, i.e. the number of nonempty groups of k consecutive cycles in the reduced reservation tables. A secondary objective is to *maximize* the numbers of resource usages in these nonempty words, as more resource usages per word permit faster (early out) detection of resource contentions. This objective function is referred to as *k-cycle-word uses*, where *k* is the number of bitvectors packed in a single memory word.

Selection Heuristic. Although integer programming can solve these minimum cover problems, we have found a fast and effective heuristic. First, we prune the resources of the generating set by successively removing each resource that produces a set of forbidden latencies that is generated or covered by a remaining resource. At this point all nonmaximal resources, if any, and others such as mirror-images of maximal resource, are eliminated from the generating set. Second, for each nonnegative forbidden latency, we build a list containing all usage pairs that generate the forbidden latency in the pruned generating set.

Third, we choose one of the forbidden latencies with the shortest list of usage pairs. The heuristic selects from the list the usage pair that covers the largest number of forbidden latencies not yet covered by currently selected resource usages. In case of ties, the heuristic selects the usage pair whose newly covered forbidden latencies have a larger sum. Once a usage pair is selected, the pair of usages and the corresponding resource are marked as selected. When using the bitvector-representation, the heuristic also marks every other usage of marked resources within the same word. The selection heuristic then proceeds with the next nonnegative forbidden latency (third step) until the marked resource usages cover every forbidden latency in the target machine.

6 Reduced Machine Examples

In this section, we present experimental results for three machines, the DEC Alpha 21064, the MIPS R3000/R3010, and the Cydra 5. Each reduced machine description generates exactly the same forbidden latency matrix as the original machine description. For the Cydra 5 machine descriptions, we also verified that precisely the same schedules were produced regardless of the machine description used by the compiler when scheduling a benchmark suite of 1327 loops obtained from the Perfect Club [25], SPEC-89 [26], and the Livermore Fortran Kernels [27].

For each machine and internal representation, we present three data points, including the total number of resources in the machine description, the average number of resource usages per operation class in the machine description, and the average number of words of bitvectors that need to be tested to answer a query, i.e. the number of nonempty groups of k consecutive cycles in the reservation tables. The third metric, referred to as *word usage*, is averaged over all operation classes and possible alignments between the bitvectors encoding the reserved and reservation tables. In this section we assume that each operation class has the same frequency, which yields pessimistic average usages since complex operations are usually less frequent than simple operations. We also assume that the performance of the contention query module is proportional to the average resource usage or word usage per operation, depending on the internal representation. A more detailed performance analysis of the contention query module is presented in Section 8.

As a proof of concept, we investigated our technique on the Cydra 5 machine [21] which has the most complex resource requirements of the three machines. The machine configuration investigated here has 7 functional units: 2 memory port, 2 address generation, 1 FP adder, 1 FP multiplier, and 1 branch unit. The original machine description used by the Cydra 5 Fortran77 compiler [1] was manually optimized, i.e. some physical resources were eliminated from the machine description as they did not introduce any new forbidden latencies [28]. This description models 56 resources and 152 distinct patterns of resource usages, resulting in 52 distinct operation classes with 10223 forbidden latencies. It is significantly larger than the machine descriptions used in previous studies, e.g. it has 3.5 times more operation classes and 2.4 times more forbidden latencies than the MIPS R3000/R3010 machine description used in [15]. Our algorithm reduced this original Cydra 5 machine description in less than 11 minutes on a SPARC-20.

Table 1 presents data for four reduced machine descriptions and the original description of the Cydra 5. The second col-

Representation:	original	discrete	bitvectors	(32 bits)	(64 bits)
Objective function minimizing:	–	*res-uses*	*1-cycle-word uses*	*4-cycle-word uses*	*9-cycle-word uses*
number of resources	8	7	7	7	7
average resource usages / operation	12.8	5.8	5.9	8.1	10.9
average word usages / operation	11.6	5.0	4.8	2.9	2.0

Table 3: Results for the DEC Alpha 21064: 12 operation classes, 293 forbidden latencies (all $\leq$ 58).

Representation:	original	discrete	bitvectors	(32 bits)	(64 bits)
Objective function minimizing:	–	*res-uses*	*1-cycle-word uses*	*4-cycle-word uses*	*9-cycle-word uses*
number of resources	22	7	7	7	7
average resource usages / operation	17.3	7.3	8.1	8.3	8.5
average word usages / operation	11.0	5.6	5.6	2.4	1.6

Table 4: Results for the MIPS R3000/R3010: 15 operation classes, 428 forbidden latencies (all $\leq$ 34).

umn corresponds to the machine description reduced for discrete-representation, i.e. it attempts to minimize *res-uses*. The three remaining columns correspond to machine descriptions for reduced bitvector-representation, i.e. they attempt to minimize *k-cycle-word uses* and secondarily maximize *res-uses*. Underlined numbers correspond to the entries minimized by the respective objective functions.

Compared to the original machine description, the reduced machine descriptions reduce the number of modeled resources by a factor of 3.7 (from 56 to 15). For discrete representation, the average resource usage is reduced by a factor of 2.2 (from 18.2 to 8.3). Similarly, for the 64 bit word bitvector representation, the average word usage is decreased by a factor of 4.0 (from 13.2 to 3.3). Only 25% of the data storage used by the original machine description is required to store the reserved tables (4 cycles of 15 bits each vs. 1 cycle of 56 bits per word). Note the successive increases in the number of resource usages when reducing for 1, 2, and 4 cycles per word. These increases permit faster detection of resource contentions and do not increase memory space for state storage.

Table 2 presents similar data for the subset of operations actually used in the 1327 loop benchmark compiled for the Cydra 5. Comparing the original description to the reduction for a 64 bit word bitvector representation, the reduced machine description decreases the average word usage by a factor of 5 (from 7.5 to 1.5). The reservation tables associated with the machine descriptions of the original model, the discrete reduction, and the 64 bit word bitvector reduction are shown, respectively, in Figures 4a, 4b, and 4c.

Table 3 shows the results of our technique for the DEC Alpha 21064 [19] using the machine description presented by Bala and Rubin [17]. Comparing the original description to the specific reduction for a 64 bit word bitvector representation, the average word usage is decreased by a factor of 5.8. This reduced representation may detect all resource contentions by testing on average two 64 bit words even though the largest forbidden latency is 58 cycles. Bala and Rubin have presented factored finite-state automata for this processor with (237 + 232) states in the forward automata and (237 + 231) states in the reverse automata. By encoding each factored state in 8 bits, and assuming that each state is stored as opposed to recomputed, 64 bits of memory per schedule cycle are needed to cache the 8 states per cycle of the factored forward and reverse automata for this dual issue microprocessor, compared to 7 bits per schedule cycle for the bitvector specific reductions.

Table 4 shows the results of our technique for the MIPS R3000/-R3010 [20] using the machine description presented by Proebsting and Fraser [15]. Comparing the original description to the specific 64 bit word bitvector reduction, the reduced machine description decreases the average word usage by a factor of 6.9. Proebsting and Fraser reported a (forward-only) finite-state automaton for this processor with 6175 states [15].

7 Contention Query Module

To evaluate the impact of reduced machine descriptions on the performance of contention queries, we implemented a contention query module for the two representations described in Section 5. The module supports four basic functions: `check`, `assign`, `assign&free`, and `free` for a given operation X and cycle j.

Check. This function determines whether operation X can be scheduled in cycle j in the current partial schedule without generating resource contentions. For the discrete representation, this query checks each usage in the reservation table for operation X (offset by j cycles) against the corresponding entry in the reserved table. When a resource contention is detected, the function aborts its search; otherwise, every usage is checked before the query completes successfully. The number of usages tested by the function is thus bounded by the total number of usages in the reservation table for operation X.

For the bitvector representation, this function simply amounts to "anding" each nonempty bitvector in the reservation table with the corresponding column bitvector in the reserved table and checking for a 0 result. With k bitvectors packed per memory word, contentions for k consecutive cycles are effectively detected by performing a one word "and" and test. The function aborts as soon as contentions are detected; otherwise, every non-empty word in the reservation table is checked before the query completes successfully. The number of words tested by the function is thus bounded by the total number of non-empty words in the reservation table for operation X.

Assign. This function reserves each of the resources consumed by operation X scheduled in cycle j. When a discrete representation is used, this function simply sets the flag of each reserved table entry corresponding to a usage in the reservation table for operation X (offset by j cycles). When a bitvector representation is used, the words encoding the bitvectors of the reservation table for operation X (offset by j cycles) are "ored" to the corresponding words encoding the bitvectors of the reserved table.

Assign&free. Unlike the previous function, the `assign&free` function first ensures that the resources consumed by operation X

are available. If any of the resources are already reserved by other operations, these operations (and possibly others [3]) will be unscheduled and their resources released.

The function is implemented by adding a new field to each reserved table entry, identifying the operation that reserves the corresponding resource. The new field is used to determine which operation to unschedule, and is updated each time a resource is reserved.

For the discrete representation, the function iterates over each usage in the reservation table for operation X, detecting resource contentions, unscheduling operations, and updating the corresponding reserved table entry fields (flag and new field). For the bitvector representation, we use an optimistic strategy which initially does not update the new fields (*optimistic mode*). Thus the resources are checked and reserved, respectively, by simply "anding" and "oring" the words encoding the bitvectors. However, when an operation must be unscheduled because of resource contentions, the entire list of scheduled operations is scanned to reconstruct the new field entries, which will be kept up-to-date thereafter (*update mode*). In update mode, the function iterates over each usage in the reservation table for operation X since the additional fields must be updated for each resource usage anyway.

Free. This function releases the resources consumed by operation X scheduled in cycle j. When a discrete representation is used, this function simply resets the flag of each reserved table entry corresponding to a usage in the reservation table for operation X (offset by j cycles). When a bitvector representation is used, this function simply "ands" the complement of the words encoding the bitvectors of the reservation table for operation X (offset by j cycles) to the words encoding the bitvectors of the reserved table.

Note that the four basic functions iterate over the resource usages for the discrete representation and over the words encoding the bitvectors for the bitvector representation, except for the `assign&free` function in update mode which always iterates over the resource usages. Note also that the work performed to handle a resource usage is approximately comparable to the work required to handle a word encoding the bitvectors. As defined in this section, either the `assign` or `assign&free` function, but not both, may be used within a partial schedule as the later one relies on the additional field in the reserved table.

The contention query module provides an additional function that facilitates the finding of a contention-free operation at a given cycle in presence of alternatives. Alternative operations were introduced in Section 3 as related operations performing an identical task but using different resources, e.g. `add1` and `add2` where both operations perform an addition, but use distinct functional units. Alternative operations may contain more operations than those caused by replicated hardware structures, e.g. a move operation may also be implemented as `add 0` or `mult 1`. The additional function is defined as follows.

Check-with-alt. This function determines if operation X, or any of its alternative operations, can be scheduled in cycle j without resource contentions. If so, the function returns one of the contention-free operations; otherwise, it returns an error value. In this paper, we implemented this function by repetitively calling the `check` function for each of the alternative operations until it succeeds. Other more efficient techniques could be implemented.

8 Performance of the Contention Query Module

To evaluate the impact of the contention query module and the reduced machine representation, we have selected a state-of-the-art scheduler that results in high performance code at low computational complexity, and is thus likely to be representative of the scheduling algorithms used in future high-performance compilers.

We implemented a scheduler for software pipelined loops using the algorithm developed and described by Rau in [3]. This algorithm, referred to as the *Iterative Modulo Scheduler*, exploits the instruction level parallelism present in loop iterations by overlapping the execution of consecutive iterations. The key characteristic of the scheduling algorithm is its iterative nature: it schedules operations using a priority function that gives precedence to operations along critical paths and allows prior scheduling decisions to be reversed, unscheduling operations when data dependences are violated or resource contentions occur. The algorithm satisfies the definition of the unrestricted scheduling model since it schedules operations in arbitrary order and may reverse scheduling decisions.

We used a benchmark of loops obtained from the Perfect Club [25], SPEC-89 [26], and the Livermore Fortran Kernels [27] which consists exclusively of innermost loops with no early exits, no procedure calls, and fewer than 30 basic blocks, as compiled by the Cydra 5 Fortran77 compiler [1]. The input to the scheduling algorithms consists of the Fortran77 compiler intermediate representation after load-store elimination, recurrence back-substitution, and IF-conversion. The benchmark suite consists of the 1327 loops successfully modulo scheduled by the Cydra 5 Fortran77 compiler.

The characteristics of the generated schedules for the 1327 loop benchmark are summarized in Table 5. The first two rows indicate the number of operations per loop iteration and the initiation interval of the software pipelined loops. The third row shows to the ratio of the initiation interval (II) to the minimum initiation interval (MII), and is a good indication of the quality of the produced schedules. We see that in 95.6% of the loops, our implementation of the Iterative Modulo Scheduler produces a schedule with minimum initiation interval, i.e. achieving the maximum feasible steady-state throughput. This ratio is within 0.5% of that obtained in [3].

Measurements:	min	freq	avg	max
number of operations	2.00	0.4%	17.54	161.00
initiation interval (II)	1.00	28.7%	11.52	165.00
II/MII	1.00	95.6%	1.01	1.50
sched. decisions / operation	1.00	78.7%	1.52	6.00

Table 5: Characteristics of the 1327 loop benchmark.

A key feature of the Iterative Modulo Scheduler algorithm is that it can reverse a limited number of scheduling decisions. In this paper, the scheduler may perform up to $6N$ scheduling decisions, where N is the number of operations in the loop being scheduled. When the scheduler exceeds the allocated budget of scheduling decisions, the scheduling algorithm makes a new attempt with a larger initiation interval.

The last row of Table 5 indicates that in 78.7% of the loops, no scheduling decision was ever reversed. The actual ratio of schedule decisions to the number of operations is 1.52, when averaged over each loop and each scheduling attempt, including the 9.6% of the attempts for which the $6N$ upper limit was exceeded. The ratio is highly sensitive to the upper limit used by the scheduler; e.g. an upper limit of $2N$ results in an average ratio of 1.14, including the 11.3% of the attempts for which the $2N$ upper limit was exceeded.

The contention query module used in this section closely corresponds to the one described in Section 7, adapted to handle the periodicity of the modulo schedules (i.e. using a Modulo Reservation Table [24][29]). We used here the `assign&free` function instead of the simpler `assign` function because the scheduling algorithm schedules an operation even though it may result in resource contentions, in which case the earlier scheduled operations that conflict are unscheduled. In the 1327 loop benchmark, the `assign&free` function unscheduled one or more operations in 13.0% of the at-

Representation:	original	discrete	bitvectors	(32 bits)	(64 bits)	
Objective function minimizing:	–	*res-uses*	*1-cycle-word uses*	*2-cycle-word uses*	*4-cycle-word uses*	frequency
`check`	2.62	2.06	1.90	1.25	1.11	75.6%
`assign&free`	5.68	2.15	1.75	1.67	1.63	16.0%
`free`	6.48	2.58	2.23	1.58	1.29	8.4%
Weighted sum:	3.46	2.11	1.91	1.35	1.21	100.0%

Table 6: Performance of the basic functions (in work units per call).

tempts, accounting for 14.6% of all reversed scheduling decisions; the other reversed scheduling decisions were due to violated dependence constraints.

The performance of each basic function is quantified by counting the number of *units of work* performed by each function, where one unit of work handles a single resource usage or a single non-empty word in a reservation table. The overhead incurred in the transition from the optimistic mode to the update mode associated with the `assign&free` function is also taken into account. The average number of work units per function call is given for each of the basic functions and machine representations for the complete Cydra 5 description. The machine representations are those in Table 1.

First consider the performance associated with the discrete representations. Although the reduced machine representation in Column 2 of Table 6 eliminates much of the redundancy in Column 1, decreasing the average resource usages per operation by a factor of 2.2 (from 18.2 to 8.3 in Table 1), the average work units performed by the `check` function decreases by only a factor of 1.3, from 2.62 to 2.06 work units. This effect may be attributed to the fact that the redundancy in the original machine description helps in finding resource contention quickly. However, the redundancy in the original machine description significantly affects the average work performed by the `assign&free` and `free` functions, as the reduced description decreases the average work units by a factor of 2.6 and 2.5, respectively.

The performance associated with the bitvector representations and 1, 2, and 4 bitvectors per word, is shown in Columns 3, 4, and 5 of Table 6, respectively. We can see that increasing the number of bitvectors per word significantly decreases the average work units performed by the `check` and `free` functions, each of which iterates exclusively over non-empty words. The work performed by `assign&free` also decreases, but more moderately as it iterates either over words in the optimistic mode or over resource usages in the update mode, and incurs a mode transition overhead.

The overall performance of the contention query module is obtained by multiplying the average work units performed by each function by the relative frequency of scheduler calls to that function and summing these products. These frequencies are shown in the rightmost column of Table 6. The most frequently called function is the `check` function as, on average, the scheduler issues 4.74 `check` queries per scheduling decision. Of all the schedule decisions, the scheduler issued a single `check` query in 49.5% of the cases, two in 15.7%, three in 8.7%, four in 5.6%, five to twenty in 17.0%, and up to 96 queries in the remaining 3.5%. These numbers include the number of additional `check` queries generated by the `check-with-alt` function used to select an advantageous alternative operation; in the benchmark, 79% of the operations have no alternative and 21% of the operations have exactly one alternative.

The overall performance for the five machine descriptions are given in the last row of Table 6. When using a discrete representation, we see that reducing the machine description increases the performance of the contention query module by a factor of 1.6, decreasing the average work units from 3.46 to 2.11. When using a 64 bit word bitvector representation, reducing the machine description increases performance by a factor of 2.9, decreasing the average work units from 3.46 to 1.21. Query module work with a 64-bit machine description is thus only 21% above the absolute minimum since the contention query module must handle at least one unit of work per call to detect, reserve or free the resources modeled in any finite-resource machine model.

9 Conclusions

In this paper, we have presented an efficient contention query module that supports the elaborate scheduling techniques used by today's high performance compilers. In particular, we support unrestricted scheduling models, where the operation currently being scheduled may be placed before some already scheduled operations and backtracking is performed to produce highly optimized software-pipelined and critical-path sensitive schedules. We also support precise boundary conditions where resource requirements may dangle from predecessor basic blocks to permit effective latency-hiding techniques.

Our contention query module is based on a reduced machine description that results in significantly faster detection of resource contentions while exactly preserving the scheduling constraints present in the original machine description. This approach achieves two goals. First, it handles queries significantly faster which is increasingly important as queries for contentions are issued within the innermost loop of a scheduler and their complexity increases with machine complexity. Second, it reduces machine descriptions in an error-free and automated fashion, thus, simplifying the interface between the actual hardware structure of the target machine and the compiler representation of the scheduling constraints due to resource contentions.

Experiments with three machine descriptions indicate that our approach addresses the perceived weakness of resource modeling approaches based on reservation tables. Because the machine descriptions are reduced, all resource contentions of one query are detected by a conservative (unweighted) average of 1.6 (MIPS R3000/R3010), 2.0 (DEC Alpha 21064), and 3.3 (Cydra 5) "and" operations when using the 64 bit word bitvector representation. Precise experiments with a state-of-the-art scheduler and a contention query module indicates that the average number of work units (processing one non-empty word or resource usage in a reservation table) is as low as 1.21 for a benchmark of 1327 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels on the Cydra 5 using the 64-bit word bitvector representation. Moreover, the memory requirements needed to store the reserved resources of a schedule are small, as a 64 bit word may encode the bitvector of 4 (Cydra 5), 9 (MIPS R3000/R3010), or 9 (DEC Alpha 21064) schedule cycles.

Acknowledgments

This work was supported in part by the Office of Naval Research under grant number N00014-93-1-0163 and by Hewlett-Packard;

much of this work was carried out while visiting Hewlett Packard Laboratories in summer 1995. The authors would particularly like to thank B. Ramakrishna Rau for suggesting the problem and providing much helpful guidance in formulating it. The authors would also like to thank Santosh Abraham, Scott Mahlke, B. Ramakrishna Rau, and Michael Schlansker for many useful suggestions and for providing the Cydra 5 machine model and loop benchmark. The help of Todd Proebsting regarding the MIPS R3000/R3010 machine model and Vasanth Bala on the Alpha 21064 machine model is also gratefully acknowledged.

References

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

[2] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus. Enhanced Modulo Scheduling for loops with conditional branches. *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 170–179, Dec. 1992.

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

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

[5] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocs. *Proceedings of the International Conference on Supercomputing*, pages 442–452, 1988.

[6] K. Ebcioglu, R. D. Groves, K.-C. Kim, G. M. Silberman, and I. Ziv. VLIW compilation techniques in a superscalar environment. In *Proc. of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 36–48. 1994.

[7] G. P. Lowney et al. The Multiflow trace scheduling compiler. In *The Journal of Supercomputing*, volume 7, pages 51–142, 1993.

[8] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Transactions on Computers*, 44(4):481–494, April 1995.

[9] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[10] S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 55–71, Sept. 1992.

[11] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the Eighteenth Annual International Symposium on Computer Architecture*, pages 266–275, May 1991.

[12] J. C. Gyllenhaal. A machine description language for compilation. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[13] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[14] 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.

[15] T. A. Proebsting and C. W. Fraser. Detecting pipeline structural hazards quickly. *Twenty-First Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 280–286, Jan. 1994.

[16] T. Müller. Employing finite automata for resource scheduling. *Proc. of the 26th Annual International Symposium on Microarchitecture*, pages 12–20, 1993.

[17] V. Bala and N. Rubin. Efficient instruction scheduling using finite state automata. *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 46–56, Nov. 1995.

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

[19] Digital Equipment Corp., Maynard, MA. *DecChip 21064 Microprocessor Hardware Reference Manual EC-N0079-72*.

[20] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

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

[22] E. S. Davidson, L. E. Shar, A. T. Thomas, and J. H. Patel. Effective control for pipelined computers. *Spring COMPCON-75 digest of papers*, pages 181–184, Feb. 1975.

[23] V. Bala. Personal communication. Feb. 1996.

[24] 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*, pages 159–164, 1976.

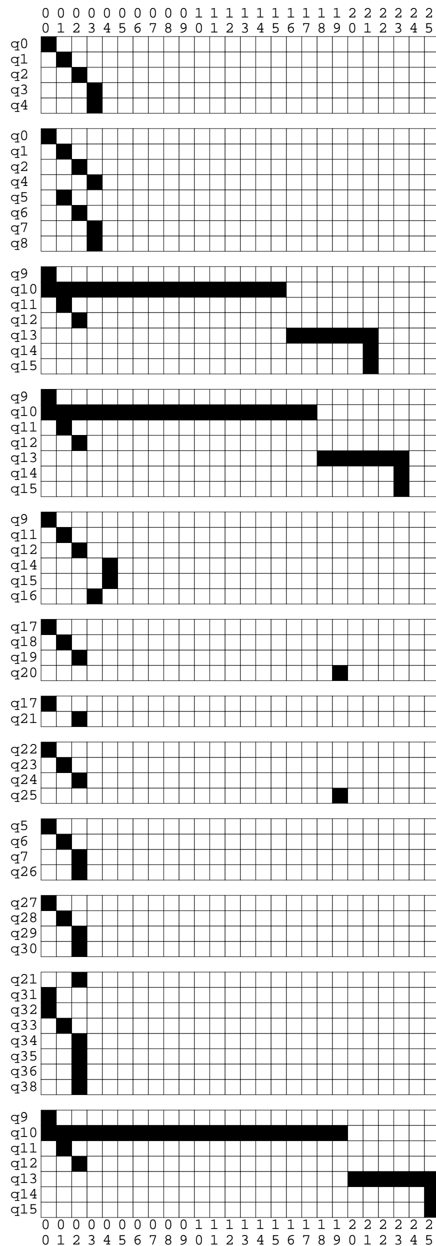
[25] M. Berry et al. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

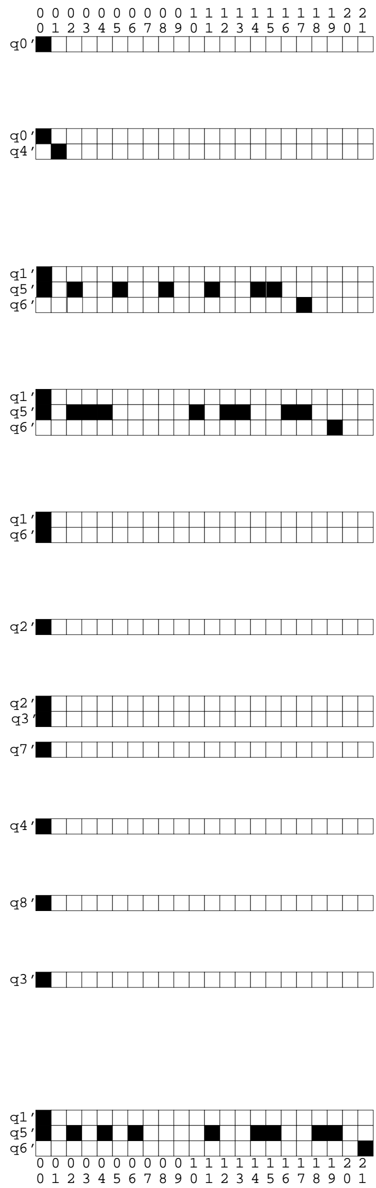
[26] J. Uniejewski. SPEC Benchmark Suite: Designed for today's advanced system. *SPEC Newsletter*, Fall 1989.

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

[28] M. S. Schlansker. Personal communication. June 1995.

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





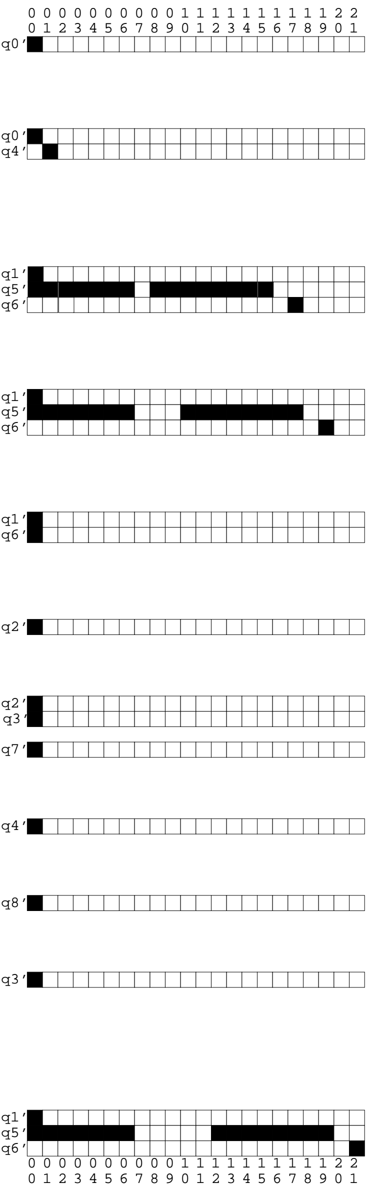


Figure 4: Reservation tables for a subset of the Cydra 5.