To appear in The 28th Annual Intl. Symp. on Microarchitecture (MICRO-28), Ann Arbor, Nov 1995

Efficient Instruction Scheduling Using Finite State Automata*

Vasanth Bala
Hewlett Packard Labs
vas@hpl.hp.com

Norman Rubin
Digital Equipment Corp.
rubin@sousa.enet.dec.com

Abstract

Modern compilers employ sophisticated instruction scheduling techniques to shorten the number of cycles taken to execute the instruction stream. In addition to correctness, the instruction scheduler must also ensure that hardware resources are not oversubscribed in any cycle. For a contemporary processor implementation with multiple pipelines and complex resource usage restrictions, this is not an easy task. The complexity involved in reasoning about such resource hazards is one of the primary factors that constrain the instruction scheduler from performing many aggressive transformations. For example, the ability to do code motion or instruction replacement in the middle of an already scheduled block would be a very powerful transformation if it could be performed efficiently. We extend a technique for detecting pipeline resource hazards based on finite state automata, to support the efficient implementation of such transformations that are essential for aggressive instruction scheduling beyond basic blocks. Although similar code transformations can be supported by other schemes such as reservation tables, our scheme is superior in terms of space and time. A global instruction scheduler that used these techniques was implemented in the KSR compiler.

1 Introduction

A legal instruction schedule must be free of data, control and structural hazards. Structural hazards refer to the contention for hardware resources by instructions executing in the pipelines. Some hardware implementations cause a pipeline stall when structural hazards are detected, while others require the compiler to insert explicit NOP instructions in the stream to prevent such hazards from occuring. Both approaches are equivalent in the sense that a pipeline stall causes a bubble in the pipeline flow, which has the same effect as a statically introduced NOP instruction. The techniques discussed in this paper deal primarily with the detection and avoidance of structural hazards in the compiler's instruction scheduling phase.

Traditionally, one of two alternatives have been employed to check for and avoid structural hazards during the instruction scheduling phase of the compiler. The first method looks backward over already scheduled code each time an instruction is considered, while the second method looks forward over the committed resources.

In the first method, the compiler maintains a list of scheduled instructions in the pipe, and tracks the cycle at which each was issued. Given a new instruction to schedule, the compiler looks back over this list and checks for a hazard between every previous instruction and the one currently under consideration.

The problem with this method is that checking each candidate instruction involves performing various checks to see if all the resource usage rules are being satisfied. Depending on the complexity of the processor implementation, the code to do this can get quite complicated and slow. Another problem is that the idiosyncracies of a particular processor implementation tend to creep into the scheduler code, making it difficult to change the compiler to target different implementations of the same processor architecture.

In the second method, the compiler explicitly models the pipeline of the processor by simulating the instruction timing. A list of committed resources is maintained in each future cycle, usually represented in the form of a bit matrix called the *resource reservation table*. Each time an instruction is considered, its resource requirements are checked against the resources already committed in the current and future cycles. The instruction is legal to issue only if there are no resource conflicts. This method is used in many production compilers today, though only occasional references can be found in the literature (for example, see [4, 2, 11]).

The problem with using resource reservation tables is that the size of the bit matrix used to represent the table is *#resources* $\times$ *length of the longest pipeline*, and every test for a resource hazard requires an OR operation on bit matrices. In global instruction scheduling schemes where the scheduling window can span several basic blocks, it is often desirable to have the ability to do code motion or instruction replacement in the middle of a scheduling window without having to reschedule the entire window. In order to support such a feature, the resource reservation table after the issue of each instruction will have to be recorded, which could result in significant space overhead.

An alternative method to using resource reservation tables was suggested by Davidson, Shar, Thomas and Patel [3]. In their method, the processor pipeline is modeled using a finite state automaton, which can be built automatically from the instruction resource

*This work was begun while the authors were at Kendall Square Research Corporation (KSR).

usage vectors for a given processor implementation. Müller used the table of states and transitions comprising such a finite state automaton in the compiler, effectively reducing the problem of structural hazard detection to fast table lookups [9]. Müller reports an 18-fold speedup compared to the resource reservation table method when the structural hazard detection in the instruction scheduler was done using the automaton. Proebsting and Fraser [10] improved on Müller's technique for constructing the finite state machine by directly computing a minimal state machine from the instruction resource usage vectors.

These methods have the advantage that the finite state automaton in built only *once* for a given processor implementation. Thereafter, the compiler simply uses the automaton transition table during instruction scheduling. Instead of storing a reservation table with each instruction in the current scheduling window, the compiler only needs to save an integer that keeps track of the automaton state at that point. Detecting a resource hazard is accomplished by a single lookup into the automaton transition table to check if the current automaton state has a legal transition on the candidate instruction.

In addition to the convenience and faster compile-time provided by such an automaton technique, we show that it can also be used to drive a sophisticated global instruction scheduler. For instance, a suitably modified automaton can give the instruction scheduler the ability to insert an instruction into the middle of an already scheduled block. This feature allows more efficient placement of compensation code that gets generated when an instruction is speculatively scheduled above a branch upon which it is control dependent.

2 Building the Automaton

Our method for constructing the finite state automaton is based on the techniques described in earlier work by Müller [9], and by Proebsting and Fraser [10]. Both methods derive the automaton from a description of the hardware resource usage vectors for the instruction set for a particular processor implementation. For example, the following are the resource usage vectors we used for the 21064 implementation of the DEC Alpha architecture [1]:

instruction class	*pipeline cycle* 0	1	2	3	4
`int/nop0`	`ib0`				
`fp/nop1`	`ib1`				
`flt-st`	`ib0+l`				
`int-st`	`ib1+l`				
`loads`	`ibx+l`				
`imull`	`ib0+m`	`m^15`	`m+r`	`m^2`	
`imulq`	`ib0+m`	`m^17`	`m+r`	`m^2`	
`stx_C`	`ib1+l`	`l`	`l`		
`fdivs`	`ib1+f`	`f^24`	`f+x`	`f^4`	
`fdivq`	`ib1+f`	`f^52`	`f+x`	`f+x`	`f^4`

Each row is a resource usage vector corresponding to the class of Alpha instruction specified in the first column. The hardware resources used by the instruction in each cycle of the processor pipeline is indicated in the table. The entry `ib0+m` in the row for `imull` specifies that this class of instruction uses the resources `ib0` and `m` in its launch cycle. The entry `m^15` in the same row specifies that the `imull` class of instruction uses the `m` resource for 15 consecutive cycles starting at cycle 1.

Not all of the resources appearing in the table above correspond to actual hardware resources on the Alpha 21064 processor. The `ibx` resource is actually a pseudo resource that we introduced to model an `ib0-or-ib1` usage constraint; the `loads` class of instruction on the Alpha 21064 can use *either* the `ib0` or the `ib1` decode resource in its launch cycle (cycle 0). Although in this paper we treat the pseudo resource `ibx` as a regular hardware resource, in some cases it may be necessary to treat it as a counted resource, allowing several instructions upto some maximum number to simultaneously use that resource in any given cycle.

Some machines may use a resource for a variable number of cycles; for example, the square root instruction timing is data dependent on the MIPS R3000. The instruction scheduler could schedule for maximum latency or expected latency (provided there is hardware interlock support). Correspondingly, the resource usage vector for such an instruction should reflect the latency assumption that the scheduler expects to make. In some cases, an assumption of maximum latency could lengthen the depth of the longest pipeline considerably, and therefore the size of the automaton (see Section 2.1). In such cases we can apply an automaton factoring technique, discussed in Section 2.4, to isolate the modeling of such long latency instructions in a separate automaton.

2.1 States and Transitions

In the Müller automaton [9], each state corresponds to a *resource reservation table*. The reservation table explicitly models each resource. It is represented by a bit matrix of size *number of resources* $\times$ *length of the longest pipe*. The value at position i, j of state S specifies whether the i'th resource will be used j cycles in the future.

Each instruction I is described by a similarly shaped matrix: the value at position i, j specifies whether instruction I uses the i'th hardware resource j cycles after it is launched. Often, it is the case that several instructions share the same reservation table. For example, the Alpha 21064 processor requires only 10 distinct reservation tables. In the remainder of the paper we will use the term *instruction class* to refer to the set of instructions that share the same bit matrix representation.

Given a state S and an instruction I, it is legal to issue I at this pipeline state only if the reservation table for I does not have any bits set that are also set in the reservation table corresponding to S. If legal to issue, a transition takes place to some state S$'$. The reservation table for S$'$ is computed by OR-ing the reservation tables for S and I.

Sometimes additional information may be required to fully describe the rules of instruction issue. For example on some processors, instructions may be multi-issued only if special alignment rules are followed. Therefore, a state may need an additional bit indicating if the next instruction will be placed at a multi-issue boundary.

The Müller technique may not produce a minimal

automaton. A finite state minimization algorithm [6] has to be applied as a postpass step to produce the minimal automaton. Many modern processor implementations often have very large automata to begin with, and considerable memory may be required to build a non-minimal automaton. For instance, the theoretical maximum number of states possible for the MIPS R3000 processor is $2^{22\times 37}$ states, whereas the minimal automaton has only 6,175 states.

The Proebsting and Fraser automaton [10] overcomes this problem by representing each state S as a *collision matrix*. The collision matrix is a bit matrix of size *number of instruction classes* $\times$ *length of the longest pipeline*. Bit i, j in the collision matrix specifies whether the ith instruction class would create a conflict (i.e., cause a structural hazard) with the current pipeline state j cycles into the future.

In this formulation, an instruction I is described by a similar collision matrix: bit i, j specifies whether an instruction (actually an instruction from the instruction class i) would create a conflict with instruction I in cycle j if launched in the current cycle. Proebsting and Fraser's technique directly produces the minimal automaton for

Given a state S, and an instruction I, the new state transition S′ is computed as follows. Instruction I is *legal to issue* in the current cycle from state S if the row corresponding to instruction class I in the collision matrix of S does not have the bit in its first column set. Otherwise, it indicates that launching I will create a structural hazard in the current cycle, and this represents an illegal transtition. If instruction I can be legally issued from state S, the collision matrices of S and I are OR-ed together to get the collision matrix representing state S′. S′ is refered to as the state produced as a result of "an I transition from state S", and the transition is denoted as $S \xrightarrow{I} S'$.

All states in the automaton are identified by unique integers. When the automaton is written out as a transition table, these integers serve as indices into the transition table array.

2.2 Cycle advancing states

Certain states in the automaton correspond to situations in which the hardware pipeline clock advances to the next cycle. On a single-issue machine such as the MIPS R3000, this happens after the issue of every instruction. On a multi-issue machine this happens when either all instruction decode units are in use (reservation table method), or when no more instructions can be issued from the current state (collision matrix method).

When such a state is encountered, we mark it as *cycle-advancing*, and left shift its collision matrix (or reservation table) by one column. Thus a cycle-advancing state represents the start of a new clock tick in all the pipelines.

Cycle-advancing states are the *accepting states* of the automaton. The start state (corresponding to the empty collision matrix) is also considered to be a cycle-advancing state.

In the multi-issue processor implementations that we have looked at, the number of cycle-advancing states was typically between 20% - 30% of the total number of states. For instance, out of the 13,524 states for the Alpha 21064 automaton, 3,674 were cycle-advancing states.

2.3 Algorithm to Build the Automaton

The following algorithm outlines the construction of the finite state automaton:

ALGORITHM BuildForwardFSA

```
transition_table = NULL;
state_hashtable = NULL;
new_states_list = F0; /* empty start state */

while (new_states_list is not empty) {
  Fa = remove first item in new_states_list;
  for (each instruction class i) {
    if (i is legal to issue from state Fa) {
      Fb = state created on an i transition from Fa;
      if (Fb is not already in state_hashtable) {
        Add Fb to state_hashtable;
        Append Fb to new_states_list;
      }
      /* only legal transitions are recorded */
      Add Fa -i-> Fb to transition_table;
      issued_an_inst = true;
    }
  }
  if (! issued_an_inst) {
    Remove Fa from state_hashtable;
    Left shift its collision matrix by 1 column;
    if (Fa exists in state_hashtable as Fc) {
      Move all incoming transitions into Fa
        so they now go into Fc;
      Mark state Fc as cycle-advancing;
    }
    else {
      Mark state Fa as cycle-advancing;
      Add Fa to state_hashtable;
      Append Fa to new_states_list;
    }
  }
}
```

Consider a simple dual-issue machine, in which the first issue slot can be either an integer (e.g., `iadd`, `imul`), a load or a store instruction, and the second issue slot can be a floating point instruction (e.g., `fadd`). If an instruction cannot be found for an issue slot, one of the two "NOP" instructions `inop` or `fnop` will be used to fill the slot. `inop` and `fnop` instructions thus represent empty issue slots in the schedule.

Suppose that the only hardware resources we are interested in modeling are the integer decode resource `id`, the floating point decode resource `fd` and the memory access resource `mem`. Let us assume that the machine has the following resource usage vectors for its instruction set:

instr class	*pipeline cycle* 0	1
`i`	`id`	
`f`	`fd`	
`ls`	`id+mem`	`mem`

The instruction classes denoted `i`, `f` and `ls` stand for the integer/inop, float/fnop and load/store instructions respectively.

The top half of Figure 1 shows the automaton F produced by algorithm BuildForwardFSA for this hypothetical machine. Only legal state transitions (i.e., those that are guaranteed not to introduce structural hazards) are recorded. The collision matrix corresponding to each state is also shown, along with the collision matrices corresponding to each of the instruction classes. For instance, from state F4, either an `f` or `i` class instruction can be launched without creating a structural hazard (since there are no "X"s in the first two rows of the first column in the collision matrix representing state F4), but not an `ls` class instruction. If an `f` class instruction were issued from this state, the collision matrix for F5 is produced by OR-ing the collision matrix of state F4 with that of the `f` class instruction.

The shaded states F0 and F4 are the only cycle-advancing states in the automaton F. Given state F2, a transition on `ls` produces a collision matrix that has all the bits in the first column set. Since nothing else can be issued from such a state, the collision matrix is left shifted by one column to give the collision matrix for F4, and F4 is marked cycle-advancing. Cycle advancing states are denoted by underlining the state identifier in all the figures in this paper.

Algorithm BuildForwardFSA produces an automaton transition table. This table is consulted by the compiler when it schedules inctructions.

2.4 Factoring

Although the Proebsting and Fraser automaton (using collision matrices to represent states) produces a minimal automaton, for many real machines the automata are still very large. The introduction of pseudo hardware resources necessary to model certain complicated pipeline constraints further increases the size of the automaton. For instance, the size of the minimal Alpha 21064 automaton is 13,524 states.

Instead of building one large automaton, a factoring scheme can be used to create multiple smaller automata, the sum total of whose states is less than the number of states in the original (see [5] for a detailed treatment of the theory behind automata factoring).

The factoring scheme is based on the observation that processor implementations typically divide the instruction set into different categories, and each category is executed by a different functional unit on the chip. For instance the integer and floating point units usually have different pipelines and use separate resources on most contemporary processor implementations.

The table of instruction resource usage vectors can be split into multiple tables from which the factored automata can be built using Algorithm BuildForwardFSA. In order to produce n factored automata, the hardware resources are first partitioned into n groups, and then the instruction resource usage tables are derived for each of the n groups.

All cycle-advancing hardware resources are first duplicated into n copies, one copy for each of the n groups. "Cycle-advancing resources" are those that are used by the various flavors of NOP instructions, and represent the minimal set of resources whose usage advances the pipeline clock. In the Alpha 21064 for example, the two decode resources `ib0` and `ib1` and the pseudo resource `ibx` are cycle-advancing resources.

The remaining hardware resources are divided up among the n groups in whatever scheme seems reasonable based on the target processor implementation. One scheme is to divide the remaining resources based on the functional unit they belong to on the chip. For each group of hardware resources, the instruction resource usage table is derived from the original by masking out all resources that do not belong to that group. Algorithm BuildForwardFSA is then applied separately to each of the n resource usage tables to get the n factored automata.

For example, we factored the Alpha 21064 hardware resources into two groups as: {`ib0, ib1, ibx, l, m, r`} and {`ib0, ibx, ib1, x, f`}. This gives the following two instruction resource usage tables (compare these with the original one shown in Section 2):

First factor automaton

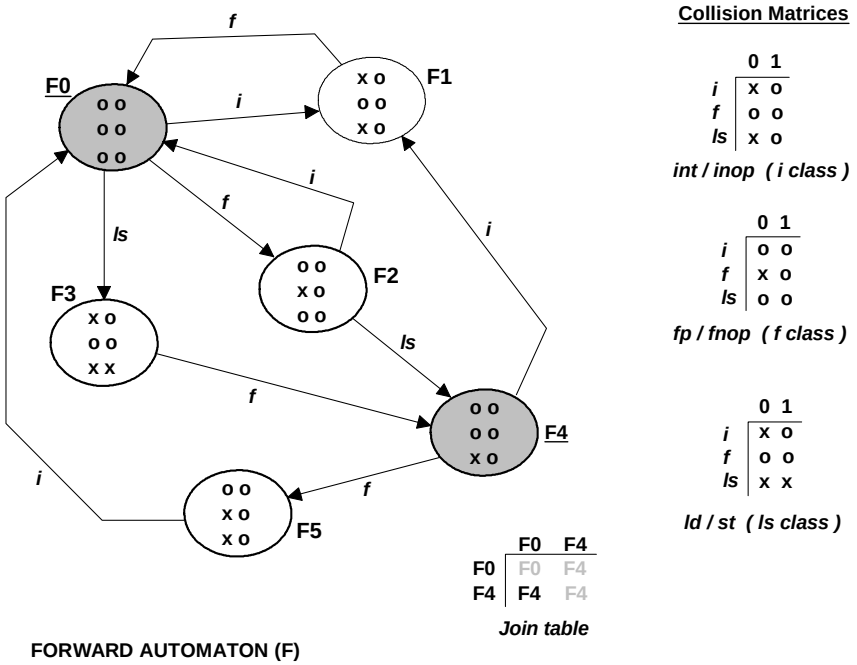
instruction class	*pipeline cycle* 0	1	2	3	4
int/nop0	ib0				
fp/nop1	ib1				
flt-st	ib0+l				
int-st	ib1+l				
loads	ibx+l				
imull	ib0+m	m^15	m+r	m^2	
imulq	ib0+m	m^17	m+r	m^2	
stx_C	ib1+l	l	l		
fdivs	ib1				
fdivq	ib1				

Second factor automaton

instruction class	*pipeline cycle* 0	1	2	3	4
int/nop0	ib0				
fp/nop1	ib1				
flt-st	ib0				
int-st	ib1				
loads	ibx				
imull	ib0				
imulq	ib0				
stx_C	ib1				
fdivs	ib1+f	f^24	f+x	f^4	
fdivq	ib1+f	f^52	f+x	f+x	f^4

The rationale for this method of splitting is that all the factor automata will always reach cycle-advancing states simultaneously since they all share the cycle-advancing resources. The factored resource usage tables can be condensed further by merging together rows that have the same resource vectors. Factored automata take up less space than the original if the partitioning is done appropriately. However, two table lookups are needed to track a transition in the factored automaton, whereas a single table lookup is required in the original automaton.

The two automata for the Alpha 21064 derived from the factored resource usage tables shown above have 237 and 232 states respectively, compared to 13,524 states in the original (minimized) automaton, a factor of 28 smaller. In fact, the states of each automaton can



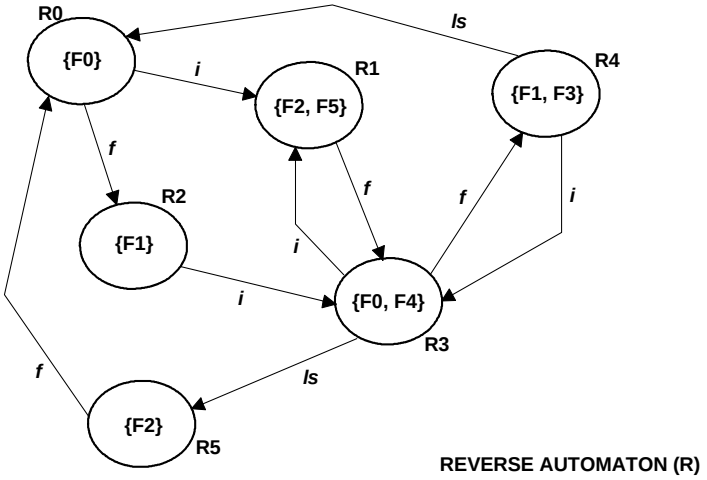


Figure 1: The forward automaton F and its reverse R for a hypothetical dual-issue machine.

be represented by `bytes` as opposed to `ints` required to represent the 13,524 states of the original. This makes the storage taken to represent both factored automata about two orders of magnitude smaller than that of the original automaton!

In the remainder of this paper we use the term "automaton state" to refer to either a state in the original unfactored automaton or the equivalent tuple of states in the factor automata. The choice of which automaton to use is driven by the space/time tradeoff that makes most sense in the context of a particular implementation, and the techniques we discuss here apply for both alternatives.

3 Instruction Scheduling

It is helpful to view the instruction scheduling problem in terms of regular expressions. We can treat the set of all legal (i.e., structural hazard free) instruction schedules for a processor as comprising the sentences of a language whose alphabet is given by the instruction set of the processor.

The Finite State Automaton discussed above recognizes all legal sentences (instruction schedules) in this language.

The cycle-advancing states of the automaton are the "accepting" states (the illegal state is not explicitly respresented in the automaton). The automaton can be started at any cycle-advancing state and used to

recognize legal instruction schedules starting from that state.

Since the automaton produced by algorithm BuildForwardFSA recognizes legal instruction schedules (sentences) in instruction execution order, we call this automaton the "forward" automaton F, whose states are denoted as F0, F1, ...Fn.

Instruction scheduling algorithms can be implemented in one of two basic flavors. The first, called *cycle scheduling*, schedules the current cycle with instructions before moving on to the next one. The second approach, called *operation scheduling* schedules instructions in priority order, placing them in the earliest legal cycle where their resource demand is satisfied.

The applicability of the automaton in the cycle scheduling scheme is straightforward. Given a starting automaton state, the instruction scheduler scans some priority queue of ready instructions (i.e., those whose input operands are available in the current cycle), and schedules the first one that allows a transition from the current state. The automaton state is updated to the new state, and the process is repeated. This will guarantee a schedule that is free of structural hazards.

In order to use the automaton for the operation scheduling scheme, we can model the scheduling process as filling a table of empty instruction slots extending into the future. This initial "table of empty instruction slots" corresponds to a sentence consisting only of NOPs. Operation scheduling is then the repeated transformation of this initial sentence by replacing specific NOPs with instructions, such that no structural hazards are created at any step.

The techniques discussed in this paper for instruction scheduling using finite state automata apply equally well for either implementation scheme. In the remainder of this paper we will assume that the compiler implements the cycle scheduling scheme since it is easier to discuss.

We also assume that as the instruction scheduler schedules a block, it records the automaton state transitions for later use. Figure 3 shows an example that illustrates this. Blocks B1 and B2 show the scheduled instruction sequences, with the forward automaton transitions listed on their left. Notice that a basic block always ends in a cycle-advancing state of F. This is because the instruction scheduler cannot break an instruction cycle (or a VLIW instruction word) across a basic block boundary.

If the traditional reservation table bit matrices were recorded in the schedule instead of these automaton states, the space requirement would be significantly increased. For instance, the reservation table for the Alpha 21064 would be a bit matrix of size 6×59 (6 resources, and a maximum pipeline length of 59 cycles for the `fdivq` class of instructions). That means about 48 bytes of storage to record a single reservation table, compared to the automaton which requires only 4 bytes (for the integer representing the automaton state) to record the equivalent amount of information. Thus, in the example in Figure 3, recording the forward automaton states for the schedule in Block B1 requires $11 \times 4 = 44$ bytes, while recording the reservation table bit matrices would require $11 \times 48 = 528$ bytes.

The key insight here is that the instruction scheduler really wants a mechanism to reason about *sentences* (schedules), and not about hardware resource usage. The automaton is merely a schedule recognition machine that efficiently answers questions about what suffixes can follow a legal schedule without introducing structural hazards. It can *detect* a structural hazard, but cannot determine what hardware resource(s) are responsible for its occurance, or at which cycle the actual hazard will occur.

3.1 Basic Block Boundaries

Before scheduling a basic block, the instruction scheduler needs to determine the automaton state at which to start. If the block has no scheduled predecessors, the state F0 is used as the starting state. State F0 is the least restrictive of the forward automaton states, since any instruction can be issued from this state.

The tricky part is deciding the starting state for a block that has more than one predecessor (i.e., a *join block*), that has already been scheduled. A complimentary problem is determining a legal ending automaton state for the block if one or more of its successors has already been scheduled earlier.

The solution to these problems are related, since we cannot allow structural hazards to occur when execution crosses a basic block boundary.

One simple solution is to never allow resources to be in use across a basic block boundary. The automaton state at the top of every block could be the empty start state, and at the end of the block enough NOP instructions can be issued until all pipes are quiesced (i.e., we wind up at the empty start state of the automaton F). The problem with this solution is that sparsely filled cycles will be created at the end of basic blocks, resulting in inefficient code.

A better solution is to derive the start state at the top of a join block (called the *join state*) by OR-ing together the automaton states (i.e., their collision matrix representations) at the end of its predecessor blocks. The problem here is that the join state produced as a result may not even belong to the forward automaton F produced by algorithm BuildForwardFSA. The reason is that such a join state represents the *confluence* of several prefix sentences in the language recognized by the automaton F, and this may not be a sentence in the language recognized by F. This can be rectified by making a suitable modification to Algorithm BuildForwardFSA as discussed below.

3.2 The Join Table

We noted earlier that all scheduled blocks must end in a cycle-advancing state of F. Consider the set of all pairs of cycle-advancing states in the forward automaton F. If we OR-ed together the collision matrices of each pair, we would get a new set of states that correspond to all possible pairwise combinition of states of F that the scheduler could encounter at the start of a join block.

There are totally $(\#\textit{cycle advancing states of } F)^2/2$ such join states (the table is symmetric about the principal diagonal due to the commutativity of the OR operation on collision matrices). Algorithm BuildForwardFSA can be modified to compute these join states on the fly. Each time a new cycle-advancing state is created, its collision matrix is OR-ed with that of all

the other cycle-advancing states already created earlier. Each such join state is denoted as Join(Fa,Fb) for the pair of cycle-advancing states Fa and Fb. This join state is also added to the `state_hashtable` used by algorithm BuildForwardFSA, and to the `new_states_list` if it was not already in the hashtable.

Note that some of the join states created in this manner may wind up being the same as some state in our original automaton F. An example is Join(F0,F4) in Figure 1; the collision matrix for Join(F0,F4) is the same as that of F4. On the other hand, we may also introduce several join states that would not have belonged to the original automaton. The latter ones can be distinguished from the other regular states of the forward automaton F by the fact that they have no incoming transitions. This is so because only a *confluence* of several prefixes that leads to such a join state, and the automaton is not built to recognize such sentences. Because the join states synthesized by the modified algorithm are also added to the `new_states_list`, there will be outgoing transitions from such join states. So once we get to a join state, we can continue to recognize hazard free instruction schedules in the usual manner.

But how do we get to the appropriate join state? One way to do this is to maintain a square table with number of rows (and columns) equal to the number of cycle advancing states of F, and use its lower triangular half to point to the appropriate join state that a given pair of F states corresponds to. We call this table the *join table* of F. In our example, the join state for Join(F0,F4) is F4, and this is shown in the join table for F in Figure 1. Incidentally, note that state F0 is the identity for the Join relation. Any unscheduled predecessor block is assumed to end in the state F0.

The join table thus allows the instruction scheduler to select an appropriate join state from which to start scheduling a join block, given the ending automaton states at the bottom of its predecessors. This works because the join of two states is a *more restrictive* state than either of them, in the sense that everything that is an illegal suffix of either state is also an illegal suffix of the join state. This is guaranteed by the construction of the join state in the modified Algorithm BuildForwardFSA. Once we have a join state in F, the usual scheduling algorithm of "issue a NOP if nothing else fits" will generate a schedule that is compatible with the incoming states from the predecessor blocks. If a join block has more than two predecessors, the join state is determined in a pairwise manner using the join table, with each subsequent lookup of the join table yielding a more restrictive join state.

If information about block execution frequencies is available to the compiler, and one of the predecessors of a join block is not frequently executed, the compiler can opt to issue NOPs at the end of that predecessor block until it reaches the F0 state.

Sometimes it is the case that both incoming states to a join block represent the situation where there is a very long instruction currently in flight (such as a divide or square root). This may cause the scheduler to follow a large number of NOP transitions starting from the join state before it actually gets to issue a real instruction. The number of additional states created by the modified Algorithm BuildForwardFSA may also increase significantly. However, if we assume that this scenario represents a somewhat rare event, we can optimize our representation of join tables by zeroing out the collision matrix of every join state beyond some cycle k, as soon as it is synthesized by the modified Algorithm BuildForwardFSA. In other words, we only model at most k NOP transitions going out of any join state, and beyond that point we will rely on the hardware to stall the pipeline to prevent any structural hazards from occurring. This will not reduce the number of join states, but it will lower the number of additional states created by algorithm BuildForwardFSA corresponding to collision matrices derived by transitions starting from the join states. Obviously this scheme works only on processors that have hardware interlock support to stall the pipeline in case of a structural hazard.

For example, the following table gives a measure of how the total number of states grows in the two factor automata for the Alpha 21064 processor, as the value of k is increased (the modified Algorithm BuildForwardFSA discussed above was used to generate each of the automata):

k	*#states in the factor automata*
0	237 + 232 (no join states)
1	240 + 238
2	250 + 267
3	274 + 332
4	326 + 473
5	434 + 766
6	654 + 1363
7	1098 + 2568

4 Scheduling Beyond Basic Blocks

One of the problems in designing a global instruction scheduler (i.e., one that speculates instructions updwards past basic block boundaries), is figuring out how to handle compensation code. Global instruction schedulers typically select a path in the control flow graph (called a "trace") based on various heuristics, and then move one or more instructions in the trace up toward the top in order to shorten the number of execution cycles incurred if the dynamic flow of control happens to go along that trace. When an instruction is moved up past an "off-trace" edge that connects a side trace into the current one, a copy of that instruction needs to be inserted on the off-trace edge. This copy is refered to as *compensation code*.

Naive insertion of compensation code penalizes the off-trace path in order to speed up the current trace, because it produces sparsely scheduled cycles at the trace join point [8]. The ideal solution is to somehow absorb the compensation code into the side trace, without lenghening the side trace.

One way to do this is to schedule the side trace (if it isn't already) and then replace any NOP instructions in its schedule with the compensation instructions. A variation is to allow the side trace schedule to be lengthened only up to some threshold.

The problem is that replacing a NOP with a compensation instruction could introduce a structural hazard with something that was either issued earlier in the schedule or something issued later in the schedule. The first part, i.e., the detection of a structural

hazard with something earlier in the schedule can be detected using the forward automaton F, whose states were recorded with the schedule. To answer the second part efficiently, we need a reverse automaton, that models resource usage in future cycles.

4.1 The Reverse Automaton

The forward automaton F can be thought of as a *suffix distinguisher*. Given a state in F, the set of paths that lead out of that state represents the set of all legal suffixes that can follow the unique prefix sentence that got us to that state.

If we reverse the transitions of the forward automaton, we will get a complimentary automaton R that is a *prefix distinguisher*. Given a state in this reverse automaton R, the set of paths that lead out of that state correspond to the set of all legal prefixes for the unique suffix sentence that got us to that state.

The following algorithm outlines the construction of a reverse automaton R, given a forward automaton F:

ALGORITHM BuildReverseFSA

```
transition_table = NULL;
state_hashtable = NULL;
new_states_list = R0; /* start state */
R0.F_states = F0;

while (new_states_list is not empty) {
  Ra = remove first state in new_states_list;
  Create a new state Rb, with Rb.F_states = φ;
  for (each instruction class i) {
    for (each element Fk ∈ Ra.F_states) {
      for (each state Fi such that Fi -i-> Fk)
        Rb.F_states = Rb.F_states ∪ Fi;
    }
    if (Rb ∉ state_hashtable) {
      Add Rb to state_hashtable;
      Add Rb to new_states_list;
    }
    Add edge Ra -i-> Rb to transition_table;
  }
}
```

It can be shown that the reverse automaton recognizes the reverse of all sentences in the language recognized by the forward automaton. The proof can be found in most standard automata theory texts [6] and is not given here.

Consider the forward automaton F shown in figure 1. Note that it is deterministic by construction: each state has at most one output transition with a given label. Simply reversing the transitions may produce a non-deterministic reverse automaton, where a state may have multiple output transitions on a given label. This is because the forward automaton may have states that have multiple incoming transitions with the same label (such as state F1 in the figure).

The non-deterministic automaton can be made deterministic by grouping together the states to which multiple output transitions occur on the same label. Each state of the deterministic automaton thus produced corresponds to a set of states of the original. Algorithm BuildReverseFSA produces a determinstic reverse automaton R directly from F, in which each state corresponds to a set of states of F. The second half of Figure 1 shows the reverse automaton R derived from the forward automaton F.

The two reverse automata corresponding to the factored forward automata for the Alpha 21064 processor had 237 and 231 states respectively, about the same size as the forward automata.

After a block of instructions has been scheduled, the instructions in the block can be scanned bottom-up to record the reverse automaton state transitions. Given any point in a scheduled block, we can now reason about alternative legal prefixes and legal suffixes at that point.

In order to record the reverse automaton states after scheduling a block, the starting reverse state at the bottom of the block must be determined. If Fe is the (cycle-advancing) forward state at the end of the block, we choose the reverse state Re that satisfies the conditions (a) Re.F_states contains Fe, and (b) among all the R states satisfying (a), Re has the smallest set of F_states. Condition (b) ensures that we pick the R state that is the *most restrictive* in terms of the choice of alternative prefix sentences that can legally reach that state (an alternative view is that condition (b) will pick the R state that allows the greatest freedom for choosing suffix sentences that can follow state Fe). Algorithm BuildReverseFSA can be easily modified to identify the most restrictive such reverse state that corresponds to each cycle-advancing F state.

As an example, consider Block B1 in Figure 3. It ends in the cycle-advancing state F0. The reverse automaton states R0 and R3 satisfy condition (a), but only R0 satisfies condition (b). If we go bottom-up along Block B1 and follow the reverse automaton transitions, we get the sequence of transitions in R corresponding to the reverse of the sentence that was recognized by F. Note that at every point in the schedule, it is always the case that the forward state Fi is contained in the set of F states corresponding to the reverse automaton state Rj at that point.

We have seen how the join table is a general solution for determining the forward state at which to begin scheduling a block. But we have not yet resolved the issue of what forward state we should end a block in. The non-trivial case is when the block being scheduled has two or more already scheduled successor blocks (unscheduled successor blocks don't matter because their start state will be determined based on their predecessors' ending states). Let F_bot be the set of ending forward states that are compatible with the start states of the scheduled successor blocks. F_bot can be derived by looking at all the reverse states Ri at the top of these scheduled successors:

$$\text{F_bot} = \text{F0} \ \cup \bigcap_i \text{R}i.\text{F_states}$$

The strategy is to schedule the block top-down in the normal manner, and if we wind up with an ending forward state Fe that belongs to the set F_bot we are done. Otherwise, we can continue issuing NOP cycles until a state is reached that is contained in F_bot. If we reach the state F0, we are guaranteed to have compatibility with the successor blocks' start states, becuase

```
    PROCEDURE Replace(Ik, Ir)

0   Let Fa = recorded forward state above Ik;
1   Let Rb = recorded reverse state below Ik;
2   if (∃ a legal transition Fa --Ir--> Fb') {
      /* Ir will not create a structural hazard with
       anything issued before Ik in the schedule */
3     if (Fb' ∉ Rb.F_states) {
        /* Ir will create a structural hazard with
         something following Ik in the schedule */
4       return FALSE;
      }
5     Replace Ik with Ir;
      /* recompute forward states */
6     for (i = Ir; i; i = next_instr(i)) {
7       Let Fa --i--> Fb';
8       Let Fb = recorded forward state below i;
9       if (Fb' == Fb)
10        break;
11      Record state Fb' in place of Fb;
      }
      /* recompute reverse states */
12    for (i = Ir; i; i = prev_instr(i)) {
13      Let Ra = recorded reverse state above i;
14      Let Rb --i--> Ra';
15      if (Ra' == Ra)
16        break;
17      Record state Ra' in place of Ra;
      }
18    return TRUE;
    }
19  return FALSE;
```

Figure 2: Code to replace an instruction Ik in a schedule with another instruction Ir.

state F0 is a completely unrestrictive state: anything can be issued starting at state F0.

If the ending state Fe is not equal to F0, then that successor block's join (forward) state at the top has to be recomputed using the join table. Looking up the join table may now result in a different starting forward state for the successor block than the one currently recorded in its schedule. In this case, the recorded forward states in the successor may need to be updated. This update is similar to the lines 6 through 11 of Procedure Replace in Figure 2 (with Fa = the newly computed join state, and Ir = the instruction at the top of the successor block). Note that the newly computed join (forward) state will also belong to the set of F states corresponding to the reverse state recorded at the top of the successor block. Thus the schedule in the successor block will remain unaltered- only the recorded states will change to reflect the different prefix schedule possibilities that can precede the schedule in the successor block. Alternatively, NOPs can be issued at the end of the current block until the ending state Fe = F0. Since F0 is the identity in the join relation, the join state at the top of the successor block will remain the same.

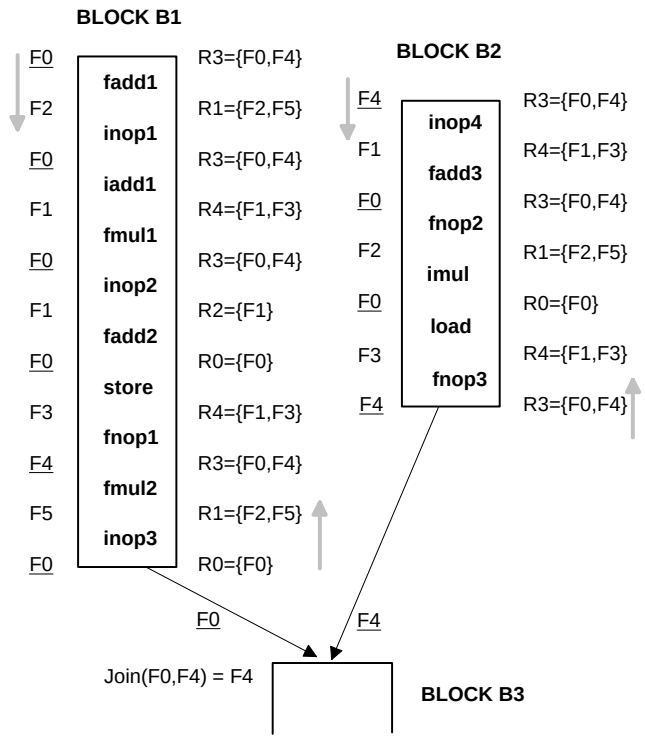


Figure 3: Scheduled basic blocks and their recorded automata state transitions.

5 Schedule Transformations

The combination of the two complimentary automata F and R make a powerful tool for global instruction scheduling. Procedure Replace in Figure 2 illustrates one example of the kind of transformation that is supported efficiently by the dual automata. Given a scheduled sequence of instructions containing a particular instruction I_k, Replace(I_k, I_r) replaces I_k with I_r if doing so does not create a structural hazard or lengthen the schedule (it is assumed that doing this replacement will not violate any data dependencies).

Suppose we want to speculate a `load` instruction (an `ls` class instruction) into Block B1. In order to avoid lengthening the schedule, this `load` must replace some `inop` instruction that already exists in the schedule, such that no structural (or data) hazards are created. `inop` instructions appear in three places in the schedule. Assuming that the data dependencies permit the `load` instruction to be placed in any of these positions in the instruction sequence, we get the following results from Procedure Replace:

- Replace(`inop3, load`) returns FALSE, becuase the test on line 2 fails. State F5 (the recorded forward state above `inop3`) does not have a legal transition on an `ls` class instruction. This is because issuing the `load` in place of the `inop3` will cause a conflict with the `store` in the previous cycle (due to a structural hazard on the "`m`" resource in the current cycle).

- Replace(`inop2, load`) also returns FALSE, because the test on line 3 fails. Even though F0 has a

legal transition on an `ls` class instruction to state F3, state F3 is not contained in the set of F states comprising R2, the recorded reverse state below `inop2`. The reason for this is that the `store` in the next cycle uses the "`m`" resource, which would be unavailable for the `load` in its next cycle, were it to be issued in place of the `inop2`.

- Replace(`inop1, load`) returns TRUE. Issuing the `load` in place of the `inop1` creates no structural hazards with instructions currently in flight or with the sequence of instructions that follow it. After the replacement is done, lines 6 through 17 update the recorded forward and reverse states in the schedule. The tests at line 9 and line 15 stop the update process when the newly synthesized state is the same as the one recorded earlier. This works because every forward state recognizes a unique prefix sentence in the language and similarly each reverse state recognizes a unique suffix.

The number of forward states updated upon a successful Replace operation is bounded by the latency of the inserted instruction. In practice, these updates can be done lazily by keeping a pair of markers with each scheduled block or trace. The top marker specifies the point up to which the reverse automaton states may be out of date, and the bottom marker specifies the point up to which the forward states may be out of date. A later Replace operation on the same scheduled sequence of instructions can first perform just the necessary updates required in order for the tests on lines 2 and 3 to work correctly.

If traditional resource reservation table bit matrices were recorded in the schedule instead of automaton states, the test for feasability of replacement (lines 2 and 3) would take time proportional to the latency of the replacement instruction Ir. The reservation tables for the sequence of instructions starting at the candidate replacement point, and extending upto the latency of the replacing instruction will have to be checked to make sure that no structural hazards can be introduced in the future. In our case, the reverse automaton states "record the future" in a compact way allowing a constant time feasability test, independent of the latency of the candidate replacement instruction Ir. Of course, one could argue that a "reverse equivalent" of reservation tables (a somewhat fuzzy concept to define in concrete terms) could also be recorded in the schedule to make this a constant time test, but that would entail an even greater space overhead since every forward *and* every reverse automaton state now has to be replaced by the equivalent resource reservation table.

The following are a few examples of other transformations similar to that implemented by Procedure Replace, that can be supported efficiently by the dual automata:

1. Replace an instruction in an already scheduled block by a new one. In other words, given a sentence $I_0 I_1 .. I_k .. I_n$, and an instruction I_r, can we replace I_k with I_r without introducing a structural hazard or lengthening the schedule? Such a transformation was illustrated by Procedure Replace, where I_r is an instruction we want to speculate up, and I_k is a NOP.

2. Interchange two consecutive instructions in an already scheduled block. In other words, given a sentence $I_0 I_1 .. I_k I_{k+1} .. I_n$, can we interchange I_k with I_{k+1} without introducing a structural hazard or lengthening the schedule? This situation arises when I_k produces a value that is read by I_{k+1}, but only a few cycles after I_{k+1} is launched. For example, suppose instruction I_{k+1} is a call and instruction I_k establishes an argument. By interchanging the two instructions, we make I_k compute the argument in the shadow of the branch to subroutine instruction I_{k+1}. Repeated use of this transformation can also be used to "bubble up" an instruction in the trace.

3. Interchange two non-consecutive instructions in an already scheduled block. In other words, given a sentence $I_0 I_1 .. I_i .. I_k .. I_n$, can we interchange I_i and I_k without introducing a structural hazard or lengthening the schedule? This may be useful in cases where consecutive instructions cannot be interchanged. For example, on a VLIW machine, operations grouped together in the same VLIW instruction bundle may not be interchangeable because certain issue slots must address particular functional units.

6 Other Uses of the Automata

The forward automaton encodes all possible legal instruction schedules for a particular implementation of the processor's instruction set architecture. This suggests that the forward automaton might be a useful tool for CPU testing. Executing the sequence of instructions corresponding to every distinct path beginning and ending at F0 in the forward automaton will make the processor go through all the possible pipeline state transitions and therefore exhaustively test the chip for all possible code sequences that the compiler can ever produce. Such a method may be less time consuming than random testing of the pipeline hardware.

Another example of the usefulness of the automaton representation was demonstrated when the KSR2 processor, a revised faster-clocked version of the KSR1 processor was delivered to the KSR compiler group. Although the two chips had the same architecture and hardware resource usage vectors for the instruction set, a hardware bug in the KSR2 required a compiler workaround. There were two specific instructions I and J that should never be scheduled one cycle apart (i.e., they could be scheduled in the same cycle, or 2 or more cycles apart, but never 1 cycle apart).

This is an arbitrary scheduling restriction, but one that can be modeled in terms of resource usages. Our solution was to introduce a dummy hardware resource `x`. We modified the resource usage vectors for instructions I and J so that instruction I used resource `x` two cycles after its launch cycle, and instruction J used resource `x` during its launch cycle. The resulting automaton only accepted schedules in which I and J were not separated by one cycle. What could potentially

have been a software hack in the instruction scheduler was handled elegantly, without touching any of the compiler code.

The dual automata are generally useful in most scenarios where scheduled code needs to be manipulated, such as in post-scheduling instruction distribution [7].

7 Conclusions

We have presented algorithms for building a forward and reverse automaton from a table of instruction resource usage vectors for a processor implementation. This is done *once* at compiler development time, and the generated transition tables are used by the compiler's instruction scheduler each time it compiles a program for that processor. A factoring scheme gives the compiler architect a spectrum of space/time tradeoffs for optimizing the size of the automaton transition tables. The key insight here is that the instruction scheduler really wants to reason about schedules, and not about what hardware resources are used where. The dual automata lets us treat the instruction scheduling process as the recognition of strings in a language.

Additionally, we also presented techniques for using these dual automata during global instruction scheduling, to perform many powerful transformations involving already scheduled blocks. These can be implemented efficiently (constant time feasability test, linear time updates), and incur substantially less space overhead than traditional methods.

Dual forward and reverse automata have been used for scanning regular languages [12], but we are not aware of other work that discusses their use in global instruction scheduling. We implemented an early version of the techniques presented in this paper in a large (over 150,000 lines of C) production compiler at Kendall Square Research, to generate code for the KSR1 and KSR2 processors used in the KSR/Series supercomputer products. In the KSR compiler implementation, the automaton table lookup replaced about 2000 lines of C code (out of a total of 6000 lines of instruction scheduler code) that was being used to model all the complex restrictions of the KSR processor pipeline.

The goal of our global scheduling strategy was to minimize the overhead due to speculative code motion even if we gambled incorrectly on the direction of certain branches. Each block was first locally scheduled, and thereafter instructions from other blocks along a likely trace were speculated up to replace NOPs in the current block. These speculated instructions left NOP holes in their original home blocks, which in turn were filled in by further speculations from below. The dual forward and reverse automata made such a strategy efficiently implementable. In fact our new improved global instruction scheduler that did the speculative code motions often ran *faster* than the old local instruction scheduler (that did not have the benefit of using the automaton tables), and the generated code was at least as efficient.

As processor implementations continue to get more complex, the software engineering issues in designing efficient instruction schedulers become significant because it could delay the time to market for a compiler version that is capable of fully exploiting all the advanced features of the processor. We believe that techniques such as the ones discussed here, that allow complicated hardware details to be separated from the scheduler code without compromising the efficiency of generated code, will become ever more important in the future.

Finally, we would like to thank Mike Smith, and Chris Reeve and other members of the KSR compiler group for participating in several discussions.

References

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

[2] J.C. Dehnert, and R.A. Towle. Compiling for the Cydra-5. In *The Journal of Supercomputing*, 7, pp. 181-227, May 1993.

[3] E.S. Davidson, L.E. Shar, A.T. Thomas, and J.H. Patel. Effective Control for Pipelined Computers. In *Spring COMPCON-75 digest of papers*. IEEE Computer Society, Feb 1975.

[4] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. In *IEEE Transaction on Computers*, C-30(7), pp. 478-490, July 1981.

[5] J. Hartmanis and R. Stearns. Algebraic Structure Theory of Sequential Machines. See Chapter 2 "Partitions and the Substitution Property". Prentice-Hall, 1966.

[6] J.E. Hopcroft, and J.D. Ullman. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979.

[7] R. Lo, S. Chan, F. Chow, and S-M. Liu. Improving Resource Utilization of the MIPS R8000 via Post-Scheduling Global Instruction Distribution. In *The 27th Annual Symposium on Microarchitecture*, San Jose, Dec 1994.

[8] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnel, and J.C. Ruttenberg. The Multiflow Trace Scheduling Compiler. In *The Journal of Supercomputing*, 7, pp. 51-142, 1993.

[9] T. Müller. Employing Finite Automata for Resource Scheduling. In *The 26th Annual International Symposium on Microarchitecture*, Austin, Dec 1993.

[10] T.A. Proebsting, and C.W. Fraser. Detecting Pipeline Hazards Quickly. In *The 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan 1994.

[11] B.R. Rau, M.S. Schlansker, and P.P. Tirumalai. Code Generation Schemas for Modulo Scheduled Loops. In *The 25th Annual International Symposium on Microarchitecture*, Dec 1992.

[12] P-C Wu, F-J Wang, and K-R Young. Scanning Regular Languages by Dual Finite Automata. In *ACM Sigplan Notices*, 27(4), pp. 12-16, April 1992.