Lazy Data Routing and Greedy Scheduling for Application-Specific Signal Processors (The Long Version)

Ken Rimey and Paul N. Hilfinger

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

Generating efficient horizontal code is difficult when intermediate results must be routed through an irregular network of functional units and pipeline registers. A compiler should do the routing on the fly as the code is scheduled. However, scheduling code in the presence of as-yet unrouted intermediate results is tricky because it is easy to unwittingly close off all feasible routes for one of them. We describe a network-flow algorithm that tests for this situation.

We have built a compiler that uses this technique. It is part of an experimental development system for application-specific signal processing chips.

1 Introduction

This paper describes a way to generate efficient code for certain horizontal-instruction-word architectures that have resisted previous attack. These architectures afford some opportunity for pipelining and for parallel operation of functional units, but rather than provide enough bandwidth to store intermediate results in memory or in a register file, they provide feedbacks in the pipeline and chaining of the functional units. This leaves the compiler with the difficult task of orchestrating the operation of the functional units and choreographing the movement of intermediate results through the network of functional units and pipeline registers. The task is greatly simplified by the assumption that encoding of instruction words is minimal and makes few or no constraints on the operation of the datapath.

Allen [1] provides a good survey of architectural issues in digital signal processing (DSP). The signal processing architectures considered in this paper resemble the microengines used to implement many general-purpose computers, but are one-level, not two-level, machines. The techniques described here might be applied to microengines, but that is beyond the scope of this paper.

1.1 Local Scheduling

The usual approach for generating horizontal code is to first generate a loose sequence of microoperations (vertical code) and then pack these microoperations tightly into a small number of instructions in a *compaction* post-pass. Compaction of one straight-line code segment at a time is *local compaction* [2], while compaction of a program flow graph—moving microoperations across forks and joins—is *global compaction* [4]. Local compaction is now well-understood; the research community has largely shifted its attention to global compaction.

However, for datapaths dominated by feedbacks and chaining, packing microoperations in a separate post-pass works poorly and generating good horizontal code for even straight-line code segments presents a challenge, as Vegdahl has observed [7]. Not only must the code generator choose which functional units to use; it must also choose how to route each intermediate result from the output of one functional unit to the input of another, optionally storing it in a register file in the meantime. This task is called *data routing*. How best to route a particular value depends on the time interval between its definition and use or uses, as well as on the datapath resources that are free during that interval. For this reason we abandon the compaction post-pass, and instead pack or *schedule* microoperations as they are generated. In this paper, we consider only *local scheduling*.

Our local scheduler is similar to the "operation scheduler" developed by Fisher et al. [3] for use in a trace-scheduling compiler for a VLIW supercomputer. Our work differs in that we consider machines in which intermediate results must often reside in *hot spots* such as busses, nodes that store data for only one instruction cycle, and registers that would obstruct computation if tied up. Like Fisher et al., we choose the route for each intermediate result lazily,

i.e., when a use is scheduled (which may be long after the definition was scheduled). The presence of hot spots complicates lazy data routing by introducing the possibility that all feasible routes for some as-yet unrouted intermediate result will be closed off unwittingly by an unfortunate scheduling decision. Rather than backtrack, we watch for and avoid this situation.

1.2 Application-Specific Processors

Digital signal processing systems are often implemented as application-specific integrated circuits (ASIC's) to reduce their cost and size. If the required sample rate is not too high, such ASIC's may incorporate programmable processors. These *application-specific processors* need to be as small as possible, while just fast enough to achieve the sample rate. An easily-modifiable processor design is important for taking advantage of the opportunity to tune each processor to suit the program that it will run.

We have constructed a compiler [6] that translates a subset of C into horizontal code for a family of application-specific processors developed by Brodersen, Rabaey, and others of the *Lager* project at Berkeley [5]. The compiler is being used in two ongoing DSP research projects: a geometry engine for robot-arm inverse kinematics and an adaptive filter for mobile radio.

The compiler must meet conflicting goals. On the one hand, it should be easy to retarget. The user should be able to evaluate a change in processor design by retargeting the compiler and then recompiling the signal processing program. On the other hand, the resulting, irregular processor architectures are difficult ones for which to generate code.

1.3 Organization of this Paper

The purpose of this paper is to describe the design that we have chosen for the compiler's back end in seeking to meet these goals. Section 2 illustrates the operation of the scheduler with the aid of a simple example. Section 3 describes the search algorithm for routing intermediate results. Section 4 describes the method for avoiding decisions that make data routing impossible. Finally, Section 5 introduces some practical facilities needed to cope with real-life architectures.

2 Greedy Scheduling

Greedy scheduling is scheduling without backtracking. This section illustrates the procedure with a simple example: using the toy datapath of Figure 1 to compute a simple recurrence.

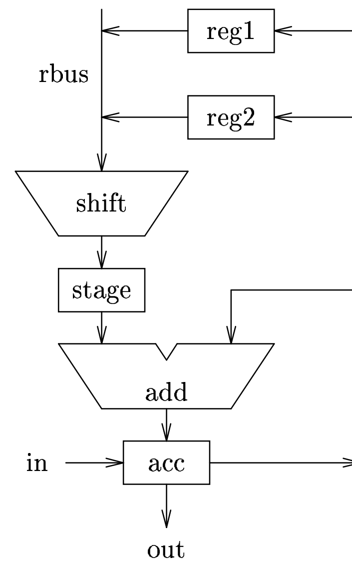

Figure 1: A toy datapath.

The datapath has two functional units: an adder, and a barrel shifter that performs a signed right-shift by zero, one, or two bits. I/O is provided to and from the accumulator `acc`. There are four registers in all: `acc`, `reg1`, `reg2`, and `stage`. Data written into any of these cannot be read until the next instruction; they effectively divide the datapath into pipeline stages. The `stage` register cannot retain data because each successive instruction overwrites its contents. With the understanding that instructions are horizontal and minimally encoded, we have now completely defined the architecture.

The recurrence to be computed implements a low-pass filter:

$$y_n = \frac{3}{4}y_{n-1} + \frac{1}{4}x_n$$

Successive inputs and outputs are denoted x_n and y_n respectively ($n = 0, 1, 2, \ldots$). The corresponding program is an infinite loop whose body, written in C, is

```
out(y = (y>>1) + (y>>2) + (in()>>2));
```

Let us assume that y will be stored in `reg1`.

2.1 Inputs to the Scheduler

The scheduler's task is to take

- a DAG (directed acyclic graph) representing a straight-line code segment and
- a machine description for the datapath

and build a schedule, a fragment of code.

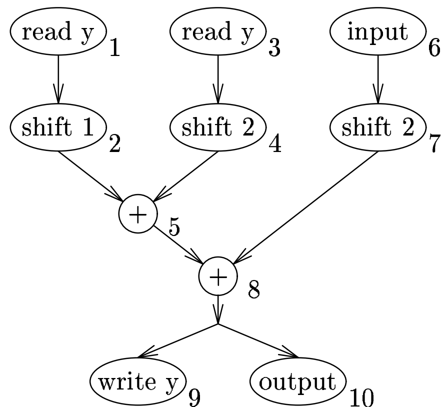

Figure 2: The sample DAG.

2.1.1 The DAG

Figure 2 shows the DAG for the body of the loop. The nodes of the DAG represent *operations*, while the edges and forks represent *values*. The edges define data dependencies, which constrain the sequence in which operations may be scheduled. Operations that interact with state may additionally be subject to side-effect dependencies; for example, there are read-write dependencies here between node 1 and node 9 and between node 3 and node 9. These read-write dependencies happen to be redundant with the data dependencies and are not shown.

The numbering of the nodes in the figure defines a total order consistent with all dependencies. We will use this as a priority order to guide the scheduler. This particular ordering was derived from the source program, which is not a bad choice in practice, although there are other possibilities.

2.1.2 The Machine Description

A machine description consists mainly of register declarations and a list of supported microoperations, written in register transfer notation.

Here is a microoperation supported by the toy datapath:

```
acc = acc + stage
```

The effect of executing this in instruction n would be to take the values in `acc` and `stage` at time n, add them together, and leave the result in `acc` at time $n+1$.

Our convention is that a result computed in instruction n appears in its destination at time $n+d$, where d is the *delay* associated with the destination. In addition to the four unit-delay registers mentioned earlier, one other destination is declared in the machine description for the toy datapath. This is `rbus`,

name	delay	storage?
reg1	1	yes
reg2	1	yes
rbus	0	no
stage	1	no
acc	1	yes

Table 1: The registers and busses of the toy datapath.

which has zero delay, and like `stage`, does not provide storage. The declarations are summarized in Table 1.

Microoperations can be divided into two groups: *function microoperations* and *transfer microoperations*. Function microoperations implement the operations associated with nodes of the DAG. Transfer microoperations just copy data from place to place and are associated with data routing.

The machine description for the toy datapath lists five register transfers representing function microoperations:

```
acc = acc + stage
acc = stage + acc
stage = rbus >> N  (N = 1, 2)
acc = in()
out(acc)
```

The first two represent two different ways to use the same microoperation to implement "+."

The machine description also lists six transfer microoperations:

```
reg1 = acc
reg2 = acc
rbus = reg1
rbus = reg2
stage = rbus
acc = stage
```

These are supplemented by three transfer microoperations implicitly defined by Table 1:

```
reg1 = reg1
reg2 = reg2
acc = acc
```

Actually, because the variable y has been assigned to `reg1`, the microoperation "`reg1=acc`" will be treated as a function microoperation in generating code for this DAG. It will be used to implement the "write y" operation. The microoperation "`rbus=reg1`," on the other hand, will be treated as an ordinary transfer microoperation. This is because the "read y" operation will not itself generate code.

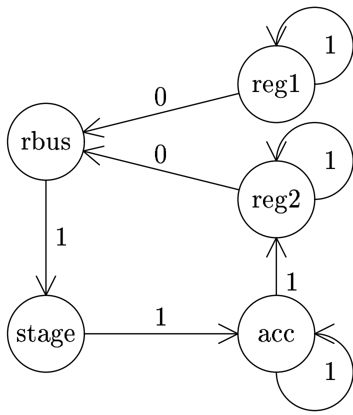

Figure 3: The *place graph* summary of transfer microoperations.

The transfer microoperations are summarized in the directed graph of Figure 3. For lack of a better term, we call the nodes of this graph *places*. Each edge is labeled with the time needed by the corresponding transfer microoperation. The edge from `acc` to `reg1` has been deleted to reflect the special treatment of "`reg1=acc`."

2.2 The Scheduling Algorithm

Horizontal code does not have to be generated one instruction at a time. Abstractly, the scheduler begins with an infinite sequence of empty instructions, inserts microoperations into these instructions, and when it is finished, drops trailing empty instructions.

There are two levels to the algorithm. On the top level, the nodes of the DAG are scheduled by a greedy algorithm, with no backtracking. Under that, the details of scheduling a node are handled by a procedure that does use backtracking.

2.2.1 Greedily scheduling the DAG

For each node N of the DAG, the scheduler must find some time T at which N can be scheduled. The basic scheduling transaction is to try a pair $[N, T]$. A transaction may succeed or fail. The idea of *greedy* scheduling is never to retract a successful one.

Thus greedy scheduling comes down to choosing what pair to try next. The choice is from among untried pairs, excluding all $[N, T]$ such that either N has been scheduled or some node on which it is dependent has not. Say $T_1 \leq T_2$ and $N_1 \leq N_2$ (i.e., N_1 is the same as N_2 or precedes it in the priority order of Section 2.1.1). Then between $[N_1, T_1]$ and $[N_2, T_2]$, $[N_1, T_1]$ is the natural one to try first. Between $[N_1, T_2]$ and $[N_2, T_1]$, however, both choices are workable. There are two basic variants of greedy scheduling:

- *Operation scheduling* chooses $[N_1, T_2]$.
- *List scheduling* chooses $[N_2, T_1]$.

Of course, there is no need to try a pair $[N, T]$ if a node N' on which N is dependent has been scheduled at time $T' > T$. With this understanding we see that, in list scheduling, the sequence of scheduled pairs is ordered by time. In some applications this means that list scheduling builds one instruction at a time. This bonus is not obtained, however, when list scheduling is used with lazy data routing. We use operation scheduling in the example of Section 2.3.

2.2.2 Scheduling a DAG node

There may be several different function microoperations that could be generated for a given DAG node. Even in our toy datapath where there is only one of each kind of functional unit, there are two ways to add numbers: The first and second arguments could be put in `stage` and `acc` respectively, or vice versa. To schedule a node N at a time T, we just try all applicable register transfers in the order in which they appear in the machine description.

We have thus reduced the problem to that of scheduling, for a given time T, a register transfer of the form

```
dest = f(src1, src2, ..., srcn)
```

This register transfer takes n argument values $v_1, v_2, \ldots, v_n$ and produces a result value u. Scheduling it involves the following steps:

1. Choose an input value v_i, and *deliver* v_i to `src`i; that is, find a sequence of transfer microoperations that propagate v_i to `src`i at time T from any place and time where it may be found.
2. Repeat Step 1 until all argument values have been delivered.
3. Schedule the function microoperation itself at time T. This involves checking for consistency with side-effect dependencies and resource usage.
4. Record the presence of a new value u in `dest` at time $T + d$, where d is the delay associated with `dest`.

If any step is inconsistent with the current schedule, the scheduling attempt *fails*; some other register transfer or some other time must be tried.

Actually, a little more persistence is a good idea. An unfortunate choice among ways of delivering one

argument may preclude the delivery of another. We find that, if delivery of an argument other than the first fails, an effective heuristic is to start over, delivering that argument first.

2.3 Example

Here we show how code is generated for the DAG of Figure 2. The first step is to note that the initial value of the variable y is available in `reg1`. For reference, this will be value 1. We will give each value the same number as the node that produces it, except for nodes that (like nodes 1 and 3) produce preexisting values.

Scheduling node 1 of the DAG, "read y," is easy; we just look up the current value of y, value 1. This becomes the output of node 1 and the input of node 2. Node 2 is the operation "shift 1," for which there is the function microoperation "`stage=rbus>>1`." We successfully schedule this in the first instruction; delivering value 1 to `rbus` involves only the transfer microoperation "`rbus=reg1`." The output of node 2 appears in `stage` in time for the second instruction (i.e., at time 2).

After scheduling nodes 1 and 2, only the first instruction is non-empty:

```
rbus=reg1, stage=rbus>>1;
```

Value 2 will be used by node 5, but other nodes are to be scheduled first. Because `stage` will lose value 2 by the third instruction, it may prove necessary to put "`acc=stage`" in instruction 2. The idea of lazy data routing is to postpone deciding whether to resort to that. Scheduling will proceed with the understanding that value 2 could, in principle, be copied into `acc`, and from there into `reg2`. Such a hypothetical sequence of transfer microoperations is called a *spill path*, and is the topic of Section 4. No microoperation that would obstruct a spill path is scheduled unless an alternative path is available.

The second "read y" node yields value 1, as the first did. A "shift 2" node computes value 4 from this. Then a "+" node sums values 2 and 4 to produce value 5. Here is the schedule at this point:

```
rbus=reg1, stage=rbus>>1;
rbus=reg1, stage=rbus>>2, acc=stage;
acc=acc+stage;
```

Only value 5 still has a spill path, since that is the only value with remaining uses.

Node 6, an "input" operation, can be inserted into the first instruction; `acc` is free then, and a spill path is available. Scheduling node 7 involves finding a mult-step delivery path for value 6. Skipping ahead, we reach the end result:

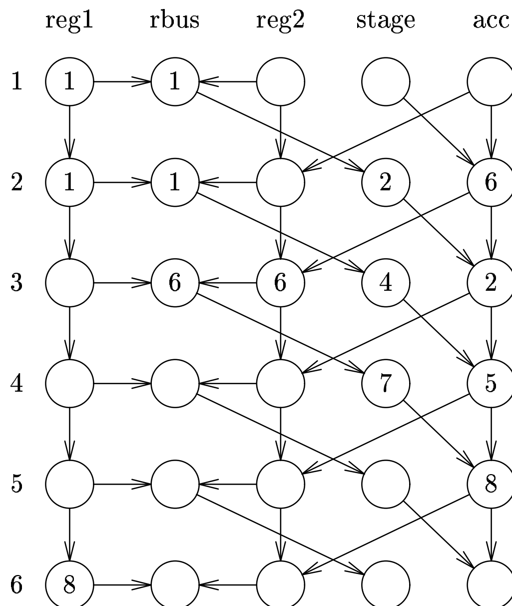

Figure 4: The *place-time graph*, truncated at $T = 6$.

```
rbus=reg1, stage=rbus>>1, acc=in();
rbus=reg1, stage=rbus>>2, acc=stage, reg2=acc;
acc=acc+stage, rbus=reg2, stage=rbus>>2;
acc=acc+stage;
reg1=acc, out(acc);
```

3 Lazy Data Routing

3.1 The Place-Time Graph

To facilitate the description and implementation of data routing, we introduce the *place-time graph*. This is a directed graph whose nodes are of the form $[P, T]$ where P is a *place*, a node of the place graph of Figure 3, and $T \in \{1, 2, \ldots\}$ is a *time*. Two place-time nodes $[P_1, T_1]$ and $[P_2, T_2]$ are connected by an edge in the place-time graph if the place nodes P_1 and P_2 are connected by an edge labeled $T_2 - T_1$ in the place graph. Figure 4 shows the place-time graph derived from the place graph of Figure 3.

The place-time node $[P, T]$ is labeled by the value v if v is available in P at time T. Labels are attached to nodes by two different mechanisms. First, the scheduling of function microoperations introduces newly computed values into the graph. Second, data routing copies values from node to node along the edges of the place-time graph. The labels shown in Figure 4 are from the example.

Various causes may prevent the labeling of a given node by a given value. The node may already have a different label. The node may be mutually exclusive

with a labeled node. (See Section 4.2.3.) Finally, the node may lie on a spill path for which there is no alternative route. We find it convenient perform these tests as steps in labeling a node. Thus we say that an attempt to label a node may *fail*.

3.2 The Data Routing Algorithm

Consider an attempt to deliver the value v to the place-time node $[P, T]$. If $[P, T]$ is already labeled by v, there is nothing to be done. If it is labeled by some other value, the delivery attempt fails. In the general case, we seek a *delivery path* through the place-time graph, and we label its nodes with v, implicitly scheduling transfer microoperations corresponding to its edges. A delivery path consists of a sequence of distinct nodes $Q_1, Q_2, \ldots, Q_n$ $[n \geq 1]$ such that

1. Q_1 is labeled by v,
2. $Q_2, Q_3, \ldots, Q_n$ are unlabeled, and
3. $Q_n = [P, T]$.

As mentioned in Section 3.1, whether an attempt to label a place-time node fails can depend on the labels of other nodes. This means that a search that visits nodes only once may fail to find an existing path. Nevertheless, we look for paths by using a depth-first search rooted at $[P, T]$. Nodes are labeled with v as the search goes forwards (following edges of the place-time graph *backwards*) and unlabeled as it backs up. The search terminates as soon as a node already labeled by v is found. The penalty for using this heuristic algorithm is occasional, unnecessary failure that may cause an operation to be scheduled later than necessary.

The order in which the depth-first search considers the parents of a node is carefully chosen according to estimates of the *delivery cost* of v. The estimate for a given node is the cost of the cheapest unlabeled path to that node from some node labeled by v, where the cost of a path is the sum of heuristic costs assigned to its nodes. A straightforward data-flow algorithm can be used to calculate these estimates. (We find it helpful to reuse still-valid estimates calculated in previous delivery attempts.)

4 Spill Paths

A *spill path* for the value v is a delivery path to any node of the form $[P, T_\infty]$, where T_∞ is a time in the distant future. A *live* value is one that labels one or more place-time nodes and is an input to an as-yet unscheduled operation. We now consider the task of maintaining a set of disjoint spill paths, one for each live value.

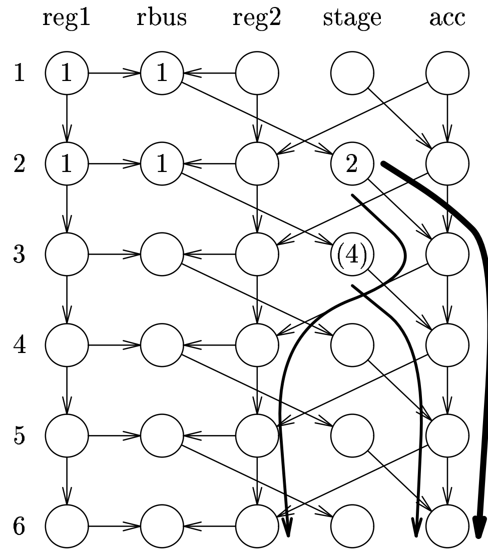

Figure 5: Old (heavy) and new (lighter) spill paths on introduction of value 4.

To illustrate, Figure 5 shows how the place-time graph is updated in scheduling node 4 of the sample DAG. This is a “shift 2” node that will generate the function microoperation:

```
stage = rbus >> 2
```

The argument, value 1, has already been delivered to `rbus`. Its spill path has been erased. Value 2, a bystander, has a spill path, shown as a heavy, curved arrow. We now label [`stage`, 3] with the result, value 4. If that node had lain on a spill path, we would have tried to reroute it. Finally, we give life to value 4 by creating a spill path for it. The final spill paths are shown in the figure as lighter, curved arrows. Notice that the one for value 2 has moved.

4.1 The Spill Path Updating Algorithm

To introduce a spill path for a new live value, or to label a place-time node that lies on a spill path, it might be necessary to adjust all existing spill paths. If a way to do this exists, it can be found very quickly, in a worst-case time linear in the size of the place-time graph (truncated at T_∞). This is accomplished by transforming the problem of rearranging node-disjoint paths in the place-time graph into a problem of rearranging edge-disjoint paths in a different graph, a zero-one network (one whose edge capacities are all one).

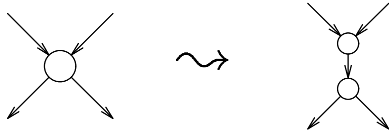

Figure 6: Constructing a network from the place-time graph.

The first step in the construction of the network is to replace every node of the place-time graph by a pair of nodes as shown in Figure 6. The pair consists of an *in-node* and an *out-node* connected by an *internal edge*. Next we delete internal edges that correspond to labeled place-time nodes. Then, for each live value, we create a source node, as well as edges from this to the out-node of every place-time node labeled by the value. Finally, we create a *sink node* to which we link all T_∞ out-nodes.

A set of spill paths is a set of edge-disjoint paths through the network, one leading from each source node to the sink node. It is also a *network flow* from the sources to the sink, conserved at each node and limited to one unit along each edge. Define an *augmenting path* to be an undirected path that traverses free edges forwards and occupied edges backwards. To *augment* a network flow along an undirected path means to change the flow along each edge from zero to one or vice versa. Augmenting a network flow along an augmenting path preserves its correctness, except that it causes one unit of flow to appear at the head of the path and disappear at the tail.

This idea is used to update the network flow to reflect the labeling of a node or the introduction of a new live value. To illustrate the latter, Figure 7 shows a fragment of the network corresponding to the place-time graph of Figure 5. The place-time node [`stage`, 2] is labeled with value 2, whose spill path is shown as a heavy, curved arrow. The place-time node [`stage`, 3] is labeled with value 4, for which a spill path is to be introduced. A source node for value 4 has been created and attached to the appropriate out-node. An augmenting path from this source node to the sink has been found and is shown as a lighter, looping arrow. Augmenting the flow along this path will result in the desired pair of spill paths.

On the other hand, say we want to label a node that lies on a spill path. In other words, we want to delete the corresponding internal edge in the network, but it is occupied. The procedure is to delete it anyway and then to seek an augmenting path from the in-node to the out-node. If in searching for such an augmenting path we stumble across one from the in-node to the sink, we do well to use it and separately rip up the flow from the out-node.

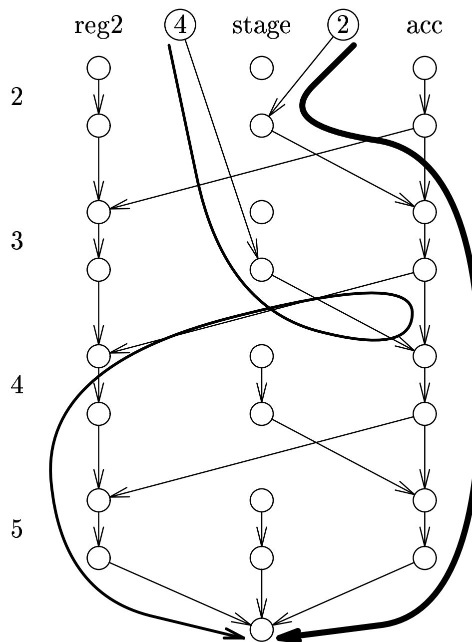

Figure 7: An augmenting path (lighter arrow).

Another useful kind of update changes T_∞. In practice, giving T_∞ a huge value wastes both time and space. A better idea is to increase it dynamically, maintaining a buffer zone between it and labeled nodes.

4.2 Using Spill Paths to Avoid Getting Stuck

Constraining the labeling of nodes as required to maintain a set of spill paths reduces the likelihood of heading down a scheduling dead end. Preventing it altogether requires a few refinements.

4.2.1 Delivery Revisited

A spill path facilitates the delivery of a value by providing a prefix for the delivery path. By keeping open a path through the early, heavily-labeled part of the place-time graph, one hopes to guarantee the existence of a delivery path to $[P, T]$ for all reasonable P and *some* T.

Unfortunately, an open but tortuous path might go undiscovered by the heuristic depth-first search described in Section 3.2. For this reason, we define a second, *reliable* delivery algorithm that makes explicit use of the spill path of the value to be delivered. The reliable algorithm uses the simple algorithm as a subroutine; moreover, the first thing it does is try the simple algorithm, which tends to find a more direct

path when it does find a path. Here is the reliable procedure for delivering the value v to the place-time node $[P, T]$:

1. Try to deliver v to $[P, T]$ using the simple algorithm. If successful, return. If the cost estimation performed by the simple algorithm proves that no path exists, quit.
2. Make a list of the nodes on the spill path of v.
3. Attach the label v to each of these nodes.
4. Try again to deliver v to $[P, T]$ using the simple algorithm. If unsuccessful, quit (after reversing Step 3).
5. Finally, remove useless labels that were attached in Step 3.

4.2.2 Acceptable Place Graphs

Consider a value that has been put in register A and will ultimately need to be delivered to registers B and C. Say it is delivered to C first. Say that, when the time arrives to deliver the value to B, its spill path leads from register C to D. If that spill path is to guarantee an opportunity to deliver the value to B, there had better be a path from D to B in the place graph. This is a constraint imposed on the place graph.

A place graph certainly meets the constraint if there is a path from A to B whenever there is a path from B to A, i.e., if the graph consists of disconnected strongly-connected components. This condition can be loosened slightly. For example, in a graph with two strongly-connected components, an edge from one to the other can be tolerated if the first component contains no place node that appears on the right side of a function microoperation, or if the second contains no place node that appears on the left side of a function microoperation. In the latter case, we delete the links in the network that go to the sink node from out-nodes of the second component.

4.2.3 Mutually Exclusive Edges

If the place-time graph has groups of mutually exclusive nodes, the corresponding network will have groups of mutually exclusive edges. These must be respected by spill paths that are to serve as reliable prefixes for delivery paths. However, finding network flows in this case is an NP-complete problem.

We update spill paths using a linear-time algorithm that reduces, in the case where there are no mutually exclusive edges, to the algorithm described in Section 4.1. When mutually exclusive edges do exist, the updating algorithm may fail to find existing ways to reroute spill paths. In practice, such failure usually does not prevent each operation from being scheduled eventually. The updating algorithm is just a depth-first search for an augmenting path, but one that updates the flow during the search in order to detect constraint violations.

5 Describing More Complex Architectures

Typical architectures for which our compiler is to generate code are more complex than the toy architecture of Figure 1. In this section, we mention a few important issues that did not arise in that example.

Consider replacing the two registers `reg1` and `reg2` of the toy datapath with n registers. Say using $2n$ bits of the instruction to select registers for reading or writing is unacceptable. Encoding the read-select bits has no impact on scheduling; reading several registers simultaneously makes no sense anyway (there being just one `rbus`). However, encoding the write-select bits introduces a resource constraint because writing the contents of `acc` into several registers simultaneously does make sense.

Such a file of n registers can be modeled using our mutual exclusion mechanism. This is a good idea for the $k \leq n$ registers to which variables are assigned, but for the other $n - k$ registers, a different approach suggests itself: using a single place node (a *fat* one) that can hold $n - k$ values. This approach saves compilation time and space when n is large; it also allows the useful possibility $n = \infty$. Moreover, it reduces the use of mutual exclusion, improving the operation of the spill path updating algorithm. Fat place nodes are a very natural generalization for all algorithms of concern.

A memory bank is a register file with an additional feature: dynamically computed addresses. Cells of a memory bank that hold simple variables and temporary results can be treated much like the registers of a register file. Cells whose addresses are dynamically computed, however, have no place in the place-time graph. They raise the issue of aliasing, which is beyond the scope of this paper.

The machine description should not reflect technical details such as these. We supply register files and memory banks as building blocks. We provide convenient notation for associating side-effect dependencies and simple resource constraints with microoperations. We also allow the specification of register word lengths, but this last feature, which impacts

both data routing and spill path management, is only partly implemented.

6 Concluding Remarks

Lazy data routing is helpful for generating horizontal code for architectures with unusual pipeline topologies. It can find seemingly suboptimal ways of chaining functional units that, in the context of the surrounding program, lead to better code. Its implementation is somewhat complex, but the reward for the effort is a reduced need for other optimizations. The technique is particularly useful when it is impractical to customize the compiler to each particular pipeline topology.

Lazy data routing can be used to do lazy constant generation. When two uses of a constant appear close together, a choice presents itself: The constant can be generated once and used twice, or it can be generated twice. In general, the better choice can be identified only during scheduling. To use lazy data routing to make this decision, it is only necessary to place the value in a fictitious register from which the data router can fetch it as it sees fit. The problem of choosing among many ways of generating the same constant is solved in the same stroke.

The implementation of lazy data routing in the presence of hot spots is complicated by the existence of situations in which scheduling cannot continue. Our spill path method efficiently steers the scheduler around these dead ends. We have written a compiler in Common Lisp that uses this technique and compiles roughly one line of code per second. Spill path updating uses a significant fraction of the compilation time, but surprisingly, not as much as the data-flow cost estimation of Section 3.

Acknowledgements

This research is funded by DARPA contract number N00039-87-C-0182.

References

[1] Jonathan Allen. Computer architecture for digital signal processing. *Proceedings of the IEEE*, 73(5):852–873, May 1985.

[2] Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick Wayne Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C-30(7):460–477, July 1981.

[3] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, June 1984.

[4] Joseph A. Fisher, David Landskov, and Bruce D. Shriver. Microcode compaction: Looking backward and looking forward. In *Proceedings of the National Computer Conference*, pages 95–102. AFIPS, 1981.

[5] Jan Rabaey, Stephen Pope, and Robert W. Brodersen. An integrated automatic layout generation system for DSP circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-4(3):285–296, July 1985.

[6] Ken Rimey and Paul N. Hilfinger. A compiler for application-specific signal processors. In *VLSI Signal Processing, III*, pages 341–351. IEEE Press, November 1988.

[7] Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *The 15th Annual Workshop on Microprogramming*, pages 125–133, 1982.