Lazy Data Routing and Greedy Scheduling for Application-Specific Signal Processors*

Ken Rimey and Paul N. Hilfinger
Computer Science Division
University of California
Berkeley, CA 94720

1 Introduction

This paper concerns code generation for a troublesome class of horizontal-instruction-word architectures (whose machine language resembles horizontal microcode). These are application-specific processors, minimalistic programmable processors to be incorporated into application-specific signal processing chips. The processors of interest afford some opportunity for pipelined and for parallel operation of functional units, but do not provide enough bandwidth to store intermediate results in memory or in a register file. Instead, a typical datapath provides direct connections between functional units (often through pipeline registers), forming an irregular network.

The usual way to generate horizontal code is to first generate a loose sequence of microoperations (vertical code) and then pack these tightly into instructions in a compaction post-pass. Local compaction, which packs one straight-line code segment at a time, is now well-understood; the research community has largely shifted its attention to global compaction. For our application-specific processors, however, packing microoperations in a separate pass works poorly and generating good horizontal code for even straight-line code segments presents a challenge. 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. 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. We consider only local scheduling in this paper.

Our local scheduler is similar to the "operation scheduler" developed by Fisher et al. [1] for use in a trace-scheduling compiler for a VLIW supercomputer. However, we consider machines in which intermediate results must often reside in hot spots such as busses and latches as well as 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). Hot spots complicate 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 using a network-flow algorithm.

We have used this approach in a compiler [3] that translates a subset of C into horizontal code for a family of application-specific processors developed by Brodersen and others of the Lager project at Berkeley [2]. It is designed to be easily retargetable to aid in tuning each processor to suit the program that it will run. We expect the user to evaluate changes in processor design by retargeting the compiler and then recompiling the signal processing program. The compiler is currently being used in developing two experimental chips: a geometry engine for robot-arm inverse kinematics and an adaptive filter for mobile radio.

2 Greedy Scheduling

We illustrate greedy scheduling with a simple example: using the datapath of Figure 1 to compute a simple recurrence.

The datapath has two functional units: an adder that can pass either input unchanged, and a shifter that performs a signed right-shift by zero, one, or two bits. I/O is to and from the accumulator acc . There are four registers: acc , reg1 , reg2 , and stage . Data written into one of these cannot be read until the next instruction. The stage register is overwritten by each successive instruction. With the understanding that instructions are horizontal and minimally encoded, we have now completely defined the architecture.

The recurrence implements a low-pass filter:

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

The 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 .

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

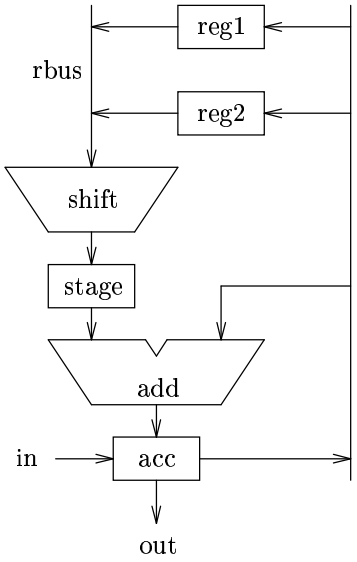

Figure 1: The example datapath.

2.1 Inputs to the Scheduler

The scheduler's task is to build a fragment of code (a schedule) from

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

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 constraining 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 nodes 1 and 9 and between nodes 3 and 9. The numbering of the nodes in the figure is a priority ordering for the scheduler, consistent with all dependencies.

The 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 example datapath:

acc=acc+stage

Executing this in instruction n sums the values in acc and stage and leaves the result in acc at time $n+1$. By convention, 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 delay-one registers mentioned earlier, one other destination is declared in the machine description for the example datapath. This is rbus, which has zero delay. The declarations are summarized in Table 1.

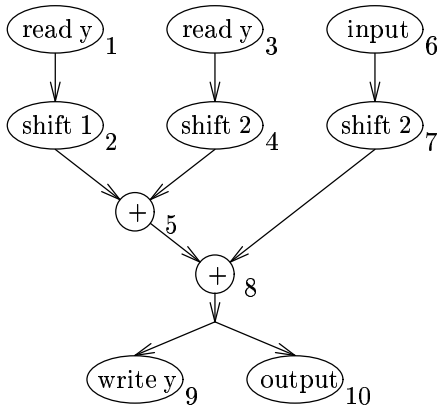

Figure 2: The sample DAG.

Microoperations fall 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 example datapath lists five register transfers for function microoperations:

acc=acc+stage	acc=stage+acc
acc=in()	out(acc)
stage=rbus¿¿N [N = 1, 2]	

The first two represent different ways to implement "+" with the same microoperation. Also listed are six transfer microoperations:

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

Because the variable y will be 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. Because "read y" will not itself generate code, rbus=reg1 will serve as an ordinary transfer microoperation.

The transfer microoperations are summarized in the directed graph of Figure 3. We call the nodes of this graph

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 datapath.

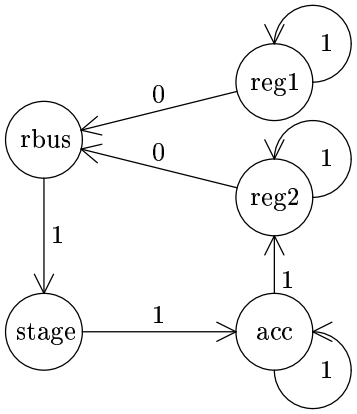

Figure 3: The place graph.

places. Each edge is labeled with the time used 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. Conceptually, the scheduler inserts microoperations into an infinite sequence of initially-empty instructions. There are two levels to the algorithm. The top level schedules the nodes of the DAG using a greedy algorithm. The lower level handles the details of scheduling each node, using some backtracking.

For each node N of the DAG, the top-level of the scheduler must find some time T at which N can be scheduled. The basic scheduling transaction is to try a pair $[N, T]$. The idea of greedy scheduling is never to retract a successful transaction. Thus greedy scheduling comes down to choosing what pair to try next. An earlier time T is preferred, as is a node N situated earlier in the priority order. Either T or N can be given more weight. We adopt the latter alternative here.

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 thus reduce the problem to scheduling a register transfer of the form

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

This takes n argument values $v1, v2, \ldots, v_n$ and produces the result value u. Scheduling it involves the following steps.

1. For each v_i, deliver v_i to src i; that is, find a sequence of transfer microoperations that propagates v_i to src i at time T from some place and time where it already exists.

2. Put the function microoperation in instruction T. Check side-effect dependencies and resource usage.

3. Record the presence of the new value u in dest at time $T + d$, where d is the delay for dest .

If any step is inconsistent with the current schedule, some other register transfer or time must be tried. (If delivery of an argument other than the first fails, we find that it is a good idea to try again, delivering that argument first.)

2.3 Example

We begin generating code for the sample DAG by designating the initial value of y in reg1 as value 1. We will give each value the same number as the node that produces it, except for nodes that produce preexisting values (nodes 1 and 3).

Scheduling node 1, "read y," is trivial; we just look up the current value of y, value 1. For node 2, "shift 1," we schedule the function microoperation stage=rbus¿¿1 in the first instruction after delivering value 1 to rbus using the transfer microoperation rbus=reg1.

As yet, 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 not retain value 2, it may prove necessary to put acc=stage in the second instruction. The idea of lazy data routing is to postpone resorting to that. Scheduling proceeds with the understanding that value 2 could, in principle, be copied into acc and from there into reg2 . Such a 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.

Continuing, the second "read y" yields value 1 as the first did. A "shift 2" 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 only it has remaining uses.

Node 6, "input," can be inserted into the first instruction; acc is free then, and a spill path is available. Nodes 7–10 take us to the end result:

```
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

For describing and implementing data routing, the place-time graph is helpful. This is a directed graph whose nodes

are of the form $[P, T]$ where P is a place and $T \in \{1, 2, \ldots\}$ is a time. The place-time nodes $[P_1, T_1]$ and $[P_2, T_2]$ are connected by an edge if P_1 and P_2 are connected in the place graph by an edge labeled $T_2 - T_1$. 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 its edges.

Various circumstances can prevent the labeling of a node. It may already have a different label. It may be mutually exclusive with a labeled node. Finally, it may lie on a spill path for which there is no other route.

A delivery path for a value v and a destination $[P, T]$ is a path consisting of 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. $Q_n = [P, T]$.

Data routing is the task of finding delivery paths.

As mentioned above, whether a place-time node can be labeled by a value v may depend on the labels of other nodes. This means that a search that visits nodes only once may fail to find an existing delivery path. Nevertheless, we look for paths with 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 when a node already labeled by v is found.

The order in which the 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.

4 Spill Paths

A spill path for the value v is a delivery path to any node of the form $[P, T_{max}]$, where T_{max} is a time in the distant future. (In practice, we let T_{max} grow dynamically.) A live value is one that labels one or more place-time nodes and is an input to an as-yet unscheduled operation. As scheduling proceeds, a set of disjoint spill paths is maintained, one for each live value.

The spill path for a value facilitates its delivery 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. Actually, an open but tortuous path might not be discovered by the depth-first search described in Section 3, but this problem can be solved with a backup delivery algorithm that makes explicit use of the spill path of the value to be delivered.

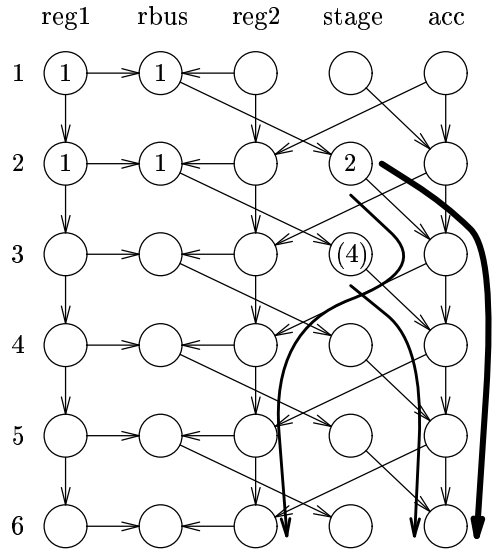

Figure 4: The place-time graph, truncated at $T = 6$. The curved arrows are the old (heavy) and new (lighter) spill paths on introduction of value 4.

We will use the first "shift 2" node in the sample DAG to illustrate the updating of spill paths. Figure 4 shows the state of the place-time graph after the argument, value 1, has been delivered to [rbus , 2]. Its spill path has just been erased. The spill path of value 2, a bystander, is shown as a heavy, curved arrow. We now label [stage , 3] with the result, value 4. Creating a spill path for this requires adjusting the spill path of value 2. The resulting spill paths are shown in the figure as lighter, curved arrows.

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_{max}). 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 (a network whose edge capacities are all one).

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 5. 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 with edges to the out-nodes of all place-time nodes labeled by the value. Finally, we create a sink node to which we link all T_{max} out-

Figure 5: Constructing the zero-one network from the place-time graph.

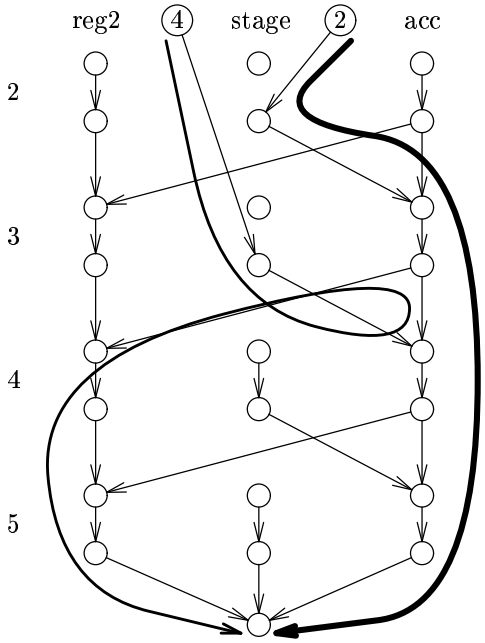

Figure 6: An augmenting path (lighter arrow).

nodes.

A set of spill paths is a set of edge-disjoint paths through the network, 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 a unit of flow to appear at the head of the path and disappear at the tail.

Figure 6, a fragment of the network for the place-time graph of Figure 4, illustrates the use of augmentation to create a new spill path. There is initially one, shown as a heavy, curved arrow emerging from the source node for value 2. A source node has been created for value 4 and a second spill path is to be introduced. Augmenting the flow along the path shown as a lighter, looping arrow will result in the desired pair of spill paths.

A similar procedure is invoked to label a place-time node that lies on a spill path. We delete the corresponding internal edge in the network even though it carries flow. Then we look for an augmenting path from the in-node to the out-node.

5 Concluding Remarks

To accommodate architectures that are more complex than Figure 1, our compiler supports register and memory banks, as well as arbitrary resource constraints between microoperations. In implementing these features, it uses a doubly-generalized place-time graph. First, certain place-time nodes can have more than one label. Second, there can be inter-node labeling constraints. The algorithms in this paper generalize naturally for the first case, but there is not much to be done for the second; when there are inter-node constraints, updating spill paths becomes an NP-complete problem and our rigorous algorithm becomes a heuristic. Nevertheless, at least when resource constraints are few, the spill path updating algorithm works quite well. It is key in making lazy data routing possible.

Lazy data routing is helpful for generating horizontal code for architectures with unusual pipeline topologies, especially when it is impractical to customize the compiler to each particular topology. It can find seemingly suboptimal ways of chaining functional units that, in the context of the surrounding program, lead to better code. It can even be used to choose among ways to generate constants; it is only necessary to place each value in one or more fictitious registers from which the data router can fetch it as it sees fit.

References

[1] 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.

[2] 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.

[3] 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.