CHAPTER 7

COMPILING C FOR SPATIAL COMPUTING

Timothy J. Callahan
School of Computer Science
Carnegie Mellon University

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

This chapter describes techniques for compiling from C or similar languages to reconfigurable architectures. We will first briefly describe the benefits of this approach and the contexts where it is most useful. Then we will describe in detail the algorithms and their technical limitations and challenges.

For the discussion in this chapter, we assume the presence of a microprocessor coupled with the reconfigurable fabric (RF). This eases adaptation in several ways and is particularly useful when supporting a mix of irregular control tasks (best suited to the microprocessor) and compute-intensive, high-throughput tasks (best suited to the RF), as described in the Processor subsection of Section 5.2.2.

The original C code can be partitioned between the central processing unit (CPU) and the RF at several granularities, including procedures, compound loops, inner loops, and blocks. The algorithms described in this chapter apply to any of these cases. The appropriate granularity for a particular system will depend on the hardware available and the particular costs involved in communication between the CPU and the RF and will not be treated in this chapter.

For most of this section we will assume that the source code, both before and after the designer's target-specific efforts to improve performance via hints in comments or pragmas, will be legal C code as defined by the ISO standard [9]. However, at the end we will overview some methods for integrating blocks designed via HDL or schematic capture into a C program.

The benefits to having a full, pushbutton path that starts from C and that can put at least some of the application on the reconfigurable hardware follow.

- There are many more C programmers than hardware designers, and writing an algorithm in C is typically faster than in an HDL.
- There is a large existing code base even for embedded applications, with at least the reference version written in C.
- Working with a single description of the entire program makes it easy for the designer or compiler to quickly explore the tradeoffs of different hardware/software partitionings. Also, it allows both hardware (HW) and

software (SW) versions to be created so that the operating system can choose at runtime which is better (see Chapter 11).
- Designers can start with automatic compilation, and then focus their efforts on improving a few loops while benefiting from the compiler's speedup on the remainder. Furthermore, with the compiler's support the designer's required effort is reduced in many cases to simply restructuring the code or embedding simple compiler directives in the form of comments or `#pragma` syntax.
- The code can be easily tested on a conventional microprocessor for correctness.

This chapter will be of direct value to those interested in compilation for spatial computing from a sequential language. More generally, it will give an application writer an understanding of the power and limitations of the state of the art of such compilers—and thereby how to write high-performance code quickly.

7.1 OVERVIEW OF HOW C CODE RUNS ON SPATIAL HARDWARE

This section provides a quick overview of how C code can be implemented on a reconfigurable fabric. It assumes basic familiarity with C. The approaches used are simple and far from optimal, but easy to understand. The detailed algorithms of how a compiler does this construction will follow.

In the figures that follow (e.g., Figure 7.1), the gray rectangles represent registers. For simplicity, the global clock is not shown. An arrow from the side toward the register indicates a load enable signal. The hardware appears at the operator level, not at the gate/CLB level.

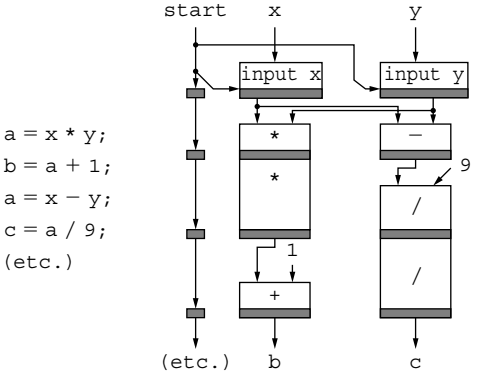

FIGURE 7.1 ■ Straight-line code.

7.1.1 Data Connections between Operations

The simplest components of C code to start with are sequences of straight-line arithmetic and logical statements. A sequence effectively tells us the set of primitive operations that make up the computation and how those operations are linked together—that is, they tell us how the outputs of one operation become inputs to other operations.

In a C program, the statements execute in order. A statement can define a variable, and subsequent statements using that variable get its last defined value. This is how value definitions are connected to their use(s)—the most recent assignment to a variable is the one that is used by a subsequent statement.

With spatial computation, each operation is implemented as a function unit (or *module*) and a producer is connected to its consumer(s) by a direct physical connection. Even if two different C statements assign to the same program variable, they are treated as different variables internally. In the example in Figure 7.1, the two definitions of variable *a*, while sequential in the C program, are actually independent and can be performed in parallel spatially. This is one step in the direction of exploiting the unlimited parallelism of spatial hardware, where we wish to reduce unnecessary ordering of operations as much as possible and keep only the necessary ordering.

Because we are implementing the computation spatially and in parallel, the actual compute datapaths are always instantiated, ready to perform their operations. It is sometimes necessary to inform the modules when their inputs are available and when they should actually perform their actions. The chain of registers on the left of Figure 7.1 acts as a very simple sequencer. In this particular example, the registers simply count off how many cycles are required to compute all of the results. A '1' bit is fed from the `start` signal, kicking off the sequencer and latching values into the input modules. The input modules hold the input values constant during execution of this unit of computation. When a 1 bit appears at `finish`, the final values are ready to pass on.

Mixed operations of different complexity (e.g., adders and multipliers) may take different amounts of time to complete. For efficient operation, rather than slowing all operators down to the latency of the slowest one, it is often worthwhile to decompose slower operators into multiple cycles, potentially pipelining them internally. In this example, multiply and divide are split into two stages requiring two cycles, while add and subtract require just one cycle each.

Throughout this section, we employ a timing discipline where values are held constant until the end of their block schedule. If a module's output register is shown at level P in the schedule, and the overall schedule length is SL, then the output of that module is guaranteed to be correct and stable from cycles P through SL of that specific block execution (where cycle 0 is when the `start` signal is raised).

7.1.2 Memory

Memory loads and stores pose additional complications beyond simple arithmetic and logical operations, in that their effects are not just local. In particular,

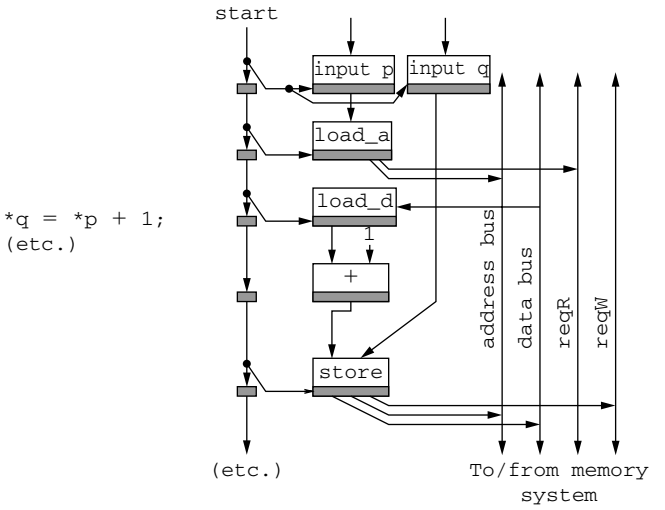

FIGURE 7.2 ■ Implementation of memory accesses.

memory can be used to perform dynamic interconnect between operations, and we must be careful to preserve the original communication semantics of the C program. A "memory" function unit has local input and output connections to other function units as normal, but also has connections to global shared address, data, and control buses. These connect each memory node to the same shared memory system.

Memory access operations must be scheduled on a particular cycle both to allow sharing among memory operations and to preserve sequential C semantics. Without scheduled coordination, two modules can attempt to drive the address or data bus simultaneously. The simple controller triggers each memory access at the correct time so that no clashes arise on either the address or the data buses. Memory access must be scheduled after its input values are ready. The compiler is also responsible for scheduling memory accesses in a way that ensures that each pair that might access the same memory location is performed in the correct relative program order.

The example in Figure 7.2 shows how a load node is split into a `load_a`, which sends the address and load request, and a `load_d` (or *load continuation*), which grabs the data when it comes back. The example assumes a load latency of just one cycle. If the memory system takes extra time to return the load data, as in the case of a cache miss, there must also be a `stall` signal factored into the sequencer to freeze execution of the subcircuit; this is *not* shown in the figure.

7.1.3 If-then-else Using Multiplexers

Simple if-then-else statements can be merged into a single subcircuit by performing the operations along both branches and then using multiplexers to

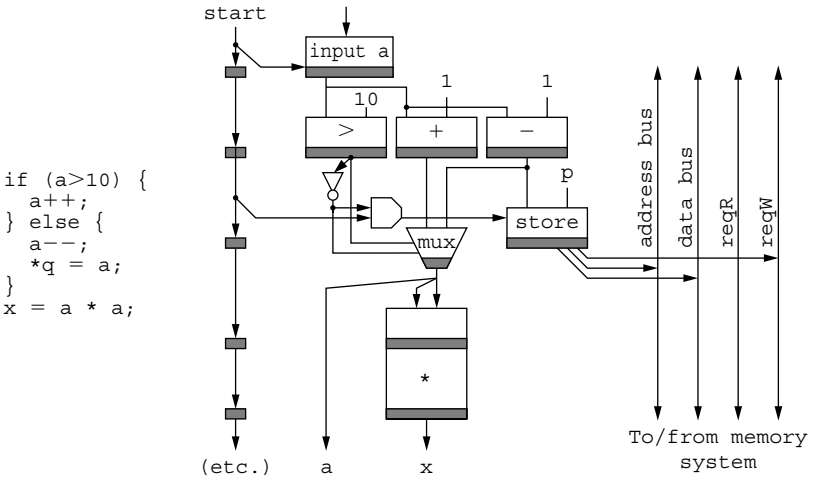

FIGURE 7.3 ■ If-conversion: Combining if-then-else using predicates and multiplexers.

select the correct version of each variable for use in subsequent computation. This removes the branch; instead, the comparison result is used as a *predicate* to choose the correct variable for later use, as with variable `a` in the example in Figure 7.3. In the figure the predicates are the result of the comparison `a > 10` and its inverse, which say whether the `then` or the `else` branch is taken. In general, a predicate is always a Boolean value—the result of a comparison, or a Boolean function of multiple comparisons, as occurs when nested if-then-else statements are reduced. `switch` statements and even forward `goto` statements can be implemented using similar techniques.

If the `then` or `else` contains a side-effect-causing operation, such as the `store` in Figure 7.3, that operation's cycle trigger must be ANDed with the predicate under which it should execute.

7.1.4 Actual Control Flow

To map C code containing more than just simple if-then-else control flow to the reconfigurable fabric, some real control flow is needed. Control flow means that there may be multiple subcircuits on the RF; only one is active at a time; and the transition from one to another subcircuit is guided by the values that are computed by the ongoing computation. This is spatial computation's implementation of a conditional branch.

The control flow is implemented with the control bit: When it reaches the end of a subcircuit, it is directed to the start of the next subcircuit to execute. When a subcircuit has multiple successors, a predicate controls which one receives the control bit. In Figure 7.4, we see the explicit branch either to a subcircuit performing the `then` computation or to the one performing the `else` computation. Subcircuit SC1 computes the condition `a > 10`, and the result determines

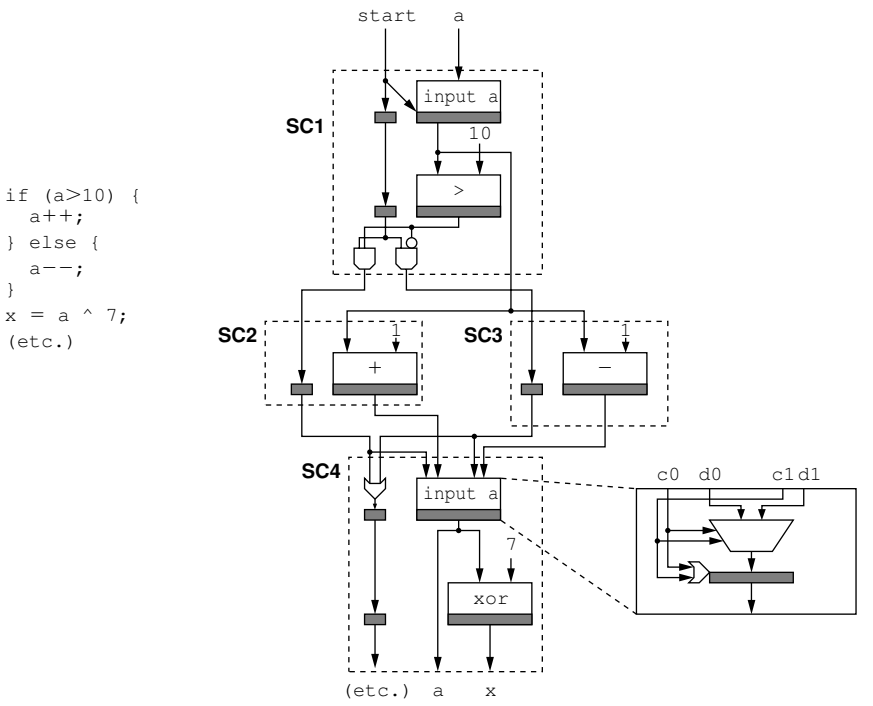

FIGURE 7.4 ■ Actual control flow.

whether the control bit goes to SC2 or SC3; then one or the other gets the control bit and executes. Control flow paths then merge at SC4, where a control bit from either SC2 or SC3 starts SC4's execution. Note that the source of the control bit entering SC4 also controls whether SC2's or SC3's final version of `a` is latched at the start of SC4 (note in Figure 7.4 the expansion of `input a`).

Subcircuits as small as those shown in Figure 7.4 would not typically be created by the compiler; instead, they would likely be merged as shown earlier. However, if SC2 and SC3 had very different execution lengths, it would be worthwhile to keep them separate like this. If, for example, one had 1-cycle latency and the other 13-cycle, we would only experience the 13-cycle latency when that path was taken. In contrast, when uneven paths are combined into one subcircuit, we pay the worst-case latency every execution.

A subcircuit that has a single predecessor actually does not require input modules, assuming in our implementation that the predecessor subcircuit holds its outputs constant until it is activated again. This simplification is shown in SC2 and SC3 of Figure 7.4.

A loop is implemented simply by control branching back to the top of itself or to some other, earlier subcircuit.

7.1.5 Optimizing the Common Path

We have seen two extremes: (1) combining all the computation in an if-then-else nest and (2) doing no combining and keeping all branches. But the key to getting the best performance from limited spatial hardware is *selectively* merging the computation on the common path(s) (to remove the subcircuit-to-subcircuit latency and to expose operation parallelism) while excluding computation on the rarely taken paths (so that it doesn't get in the way of the common case).

In Figure 7.5 we see the same code as in Figure 7.4, but we have merged the computation along the path with the increment. However, we have excluded the path with the decrement. The compiler chose to merge the computation along the path with the increment (SC1 → SC2 → SC4 from Figure 7.4) into one subcircuit because a test run (or the programmer) told it that that path was more commonly executed. Because reentering the merged increment path is not allowed, we needed to copy the XOR computation for the decrement path.

Merging the common path allowed the compiler to schedule the comparison and the addition in parallel, reducing computation time to three cycles. The schedule for the common case is also better than that for the case where all blocks were merged, as in that case we needed a multiplexer to merge the results from the decrement path, and that would add an extra step between the addition and the XOR. In the general case, the benefit of excluding a rare path could be even greater: Consider if the decrement were instead a multiplication, or even a

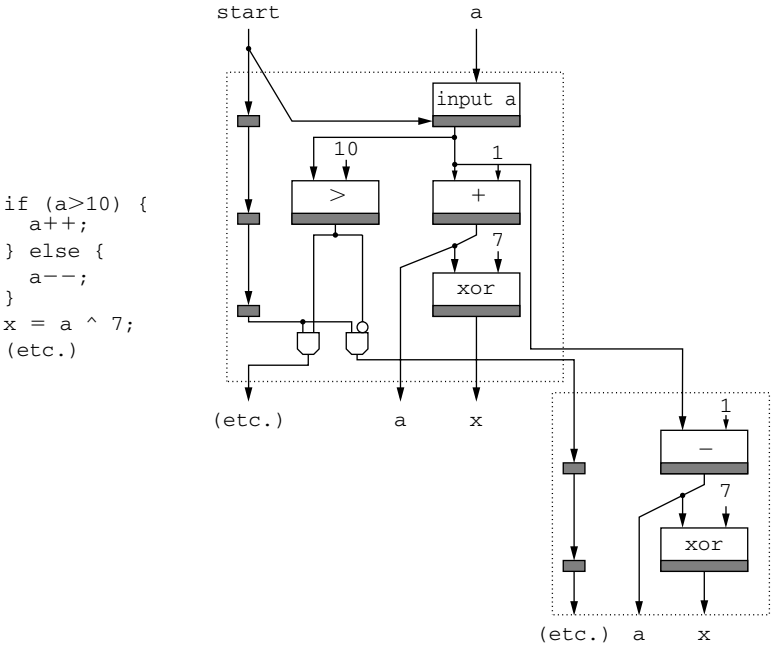

FIGURE 7.5 ■ Optimizing the common path.

long chain of operations. In that case, if that rare path were included, it would force a much longer schedule.

In this case, when the execution flow exits the common path and continues to the excluded path, the total time will be five cycles, longer than the four cycles that would have resulted if decrement had been included. Many 3-cycle executions with a few 5-cycle executions are better than all 4-cycle executions—again, optimizing the common path.

A system might also choose to implement rarely taken paths as normal software on the CPU. This would ease the demand for resources on the reconfigurable fabric and allow implementation of a loop or procedure that otherwise would not fit. This approach is also beneficial when the excluded path includes an operation, such as a library call, that cannot be implemented directly on the RF. However, the cost of transferring control to the CPU for a rare path, when it does happen, must be considered.

7.1.6 Summary and Challenges

In this section we sketched how C can be implemented spatially and began to illustrate optimizations for parallelism that are the key to extracting high performance from spatial hardware, even when the spatial hardware runs at a slower clock rate than the CPU. We also illustrated context-specific optimization, which allows us to highly specialize the computation to the common case execution of the application, further increasing parallelism and reducing the computation required. Nonetheless, these simple techniques leave us with spatial designs that can be inefficient and that underutilize our reconfigurable fabric. These inefficiencies include:

- *Not pipelining:* Sequential paths prevent us from reusing our spatial hardware at its full capacity; spatial operators sit idle for most of the cycles in a block. To fully use the capabilities of the reconfigurable hardware, datapaths should be pipelined for rapid reuse.
- *Memory:* Sequential dependencies among memory access operations limit available parallelism.
- *Operator size and specialization:* The reconfigurable fabric can provide hardware tailored to the compute needs (e.g., just the right datapath width, specialized around compile time constants), but specific information about operator size is often not immediately apparent in the original C program.

The following sections show how we can address many of the simple translation scheme's limitations.

7.2 AUTOMATIC COMPILATION

A particular compiler flow is largely determined by the system architecture. Here we will assume that fairly large pieces of code will be migrated

to the reconfigurable fabric—a loop or perhaps even a complete procedure. There is little difference in the algorithms between granularities at this level.

We assume a standard C compiler frontend that parses the source files (see Figure 7.6(a)) and performs further processing until the intermediate representation consists of a *control flow graph* (CFG) for each procedure. A CFG consists of *basic blocks*, each containing an ordered list of simple instructions and connected by control edges indicating a possible branch from the end of one basic block to the start of another, as shown in Figure 7.6(b). By definition, entry to a basic block occurs only at the beginning, exits occur only at the end, and all instructions inside the basic block execute once the block is entered.

Within each basic block, complex expressions are broken up by introducing compiler temporary variables so that each simple instruction contains just one operation. This list of simple instructions in each basic block resembles assembly code to some degree, but is of a higher level: variables (including compiler temporaries) are used instead of explicit registers, and all type information is still available. Many optimizations are performed on this representation to reduce the number of instructions by, for example, constant propagation, constant folding, and common subexpression elimination. (See Aho et al. [1] or Muchnick [13] for related background.)

The frontend also provides some standard analyses. Of particular interest here is live variable analysis, which indicates whether or not the current contents of a variable need to be preserved for a possible future use.

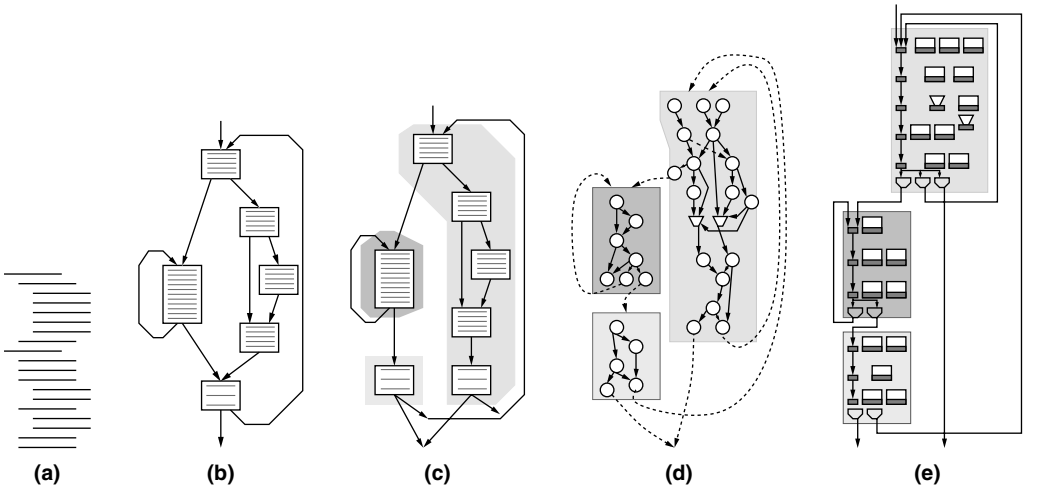

FIGURE 7.6 ■ Overall compiler flow: (a) original C source code, (b) CFG basic blocks, (c) clustering of basic blocks into hyperblocks, (d) construction of the DFG, and (e) circuit generation from the DFG.

After frontend processing produces an optimized CFG for each procedure, we start compilation steps specific to reconfigurable computing:

- *HW/SW partitioning:* This is very system dependent, and its discussion is deferred until section 7.3.1.
- *HW/HW clustering of the CFG basic blocks into hyperblocks:* illustrated in Figure 7.6(c) and discussed in section 7.2.1.
- *Building the dataflow graph (DFG) for each hyperblock:* illustrated in Figure 7.6(d) and discussed in section 7.2.2.
- *DFG optimization:* discussed in section 7.2.3.
- *Generating the circuit from the DFGs:* This involves module mapping (packing one or more DFG nodes into a single-cycle macro function unit), scheduling, connecting hyperblock subcircuits, and other related tasks; illustrated in Figure 7.6(e), which leaves out data connections, and discussed in section 7.2.4.

After we go over these steps, we will describe some uses and variations in Section 7.3.

7.2.1 Hyperblocks

Because basic blocks are limited to straight-line control flow between branches, they are often quite small and limit our opportunities for parallelism. As we saw in the previous section, we can often convert if-then-else constructs into dataflow using multiplexers. These composite blocks, or *hyperblocks*, have a single entry point at the top and one or more exits. All branches within the hyperblock are eliminated by using predicates and multiplexers. Each hyperblock becomes a subcircuit, as shown earlier.

To form hyperblocks, the compiler starts with the basic block CFG. It then combines blocks along commonly taken paths—for example, the right group in Figure 7.6(c), excluding rarely taken paths. A single basic block can always be a hyperblock. To respect the single top-entry requirement, *tail duplication* is required to eliminate an edge that otherwise would reenter the hyperblock; that edge is redirected to a copy of its original target. For example, the bottom basic block in Figure 7.6(b) is duplicated in Figure 7.6(c).

The hyperblock was originally constructed by Mahlke and colleagues [12] for compiling to VLIW (very long instruction word) processors (see VLIW datapath control subsection of Section 5.2.2), although the clustering heuristics they developed are not necessarily effective here. In particular, VLIW processors have a fixed instruction issue width; once this is saturated, adding additional parallel paths may extend the schedule and hurt the performance of the common case. With spatial computing, we have no limit on per-cycle operation parallelism, so it is often beneficial to make "fatter" hyperblocks by including more parallel paths from the CFG.

7.2.2 Building a Dataflow Graph for a Hyperblock

Here we focus on constructing a DFG (dataflow graph) from the set of basic blocks in a hyperblock. The DFG is a "stepping stone" between the original

software specification and the final spatial hardware implementation. The compiler performs many important tasks in building it:

- Control dependence within the hyperblock is converted to data dependence: Internal conditional branches are eliminated through the introduction of predicates (Boolean values indicating the "taken" path through the computation). The only remaining conditional branches are exits out of the hyperblock.
- Data producer–consumer relationships are made explicit via data edges in the graph; also, because a new DFG node is created for each definition, variable renaming is effectively performed, which eliminates false dependencies.
- Any remaining ordering constraints between individual operations, particularly memory operations, are also made explicit through ordering edges.

These actions convert the sequential ordering of instructions to a partial order of DFG nodes, exposing parallelism. In addition, maximal control speculation is employed so that all safe operations execute every iteration, removing dependencies between predicate calculations and those operations, breaking critical paths, and further increasing operation parallelism. Finally, the DFG is an ideal representation with which to perform many additional optimizations, described next.

The DFG is composed of nodes and edges:

- *Nodes:* These include constants, inputs to the hyperblock, simple computational operations having no side effects (such as addition), memory accesses, and exit nodes. Exit nodes are associated with an outgoing control edge from one hyperblock to another; when an exit node's predicate input is true, it causes a control transfer to the target hyperblock recorded on the node. The exit node also defines which live data values should be transferred to the successor hyperblock, as indicated by liveness edges.
- *Edges:* These are directed edges between the nodes and are of three types: data edges, indicating producer–consumer relationships; ordering edges, indicating an ordering constraint between two nodes such as memory operations; and liveness edges. Liveness edges go only to exit nodes. They indicate the set of values that are live-out at that hyperblock exit and thus must be copied out—that is, transferred to the successor hyperblock or back to the CPU. Each liveness edge is annotated with the name of the variable because, in general, the variable cannot be deduced from the source DFG node (a single node may be the source for different variables at different exits). These edges are necessary because the set of live variables to be transferred typically differs at each exit. Also, the source DFG node for a given variable can be different at different exits.

Top-level build algorithms

We build the DFG from the basic block CFG for each hyperblock. The algorithm for building the DFG performs a single forward pass, visiting each basic

block in the hyperblock in an order such that each basic block is visited only after all its predecessors have been visited. Then, when visiting each basic block, the simple instructions are visited in sequence. This forward pass builds all of the DFG nodes, including nodes directly translated from instructions as well as predicate calculation nodes and mux (multiplexer) nodes inserted to implement predicated execution. The forward pass also builds all data and ordering edges.

Building data edges

When a node is constructed, the compiler creates data edges to its inputs using the `lastDefs` data structure. Throughout the forward pass, this table is kept up to date regarding which node produced the last definition of each variable; there is at most one such definition at any point. We show an example in Figure 7.7.

At the start of processing a hyperblock's entry basic block, the `lastDefs` list is initialized with an input node associated with each live variable, as with `y:n1` in the example.

Whenever an instruction assigns to a variable, `lastDefs` is updated. In our example, `y++` in BB1 uses the current value of `y`, `n1` as the source for the incoming edge to the new add node, `n4`; then the `lastDefs` list is updated so that the new value of `y` is available from `n4`.

A copy—an assignment from one variable to another—requires no action other than updating the `lastDefs` list (see for example `x=y` in BB1 in Figure 7.7). A new entry for `x` is made in the `lastDefs` list, `x:n1`, just using the current entry for `y`. Similar for `z=y`, although at that point the entry for `y` is different so a different source node is given to `z`. This has the effect of performing

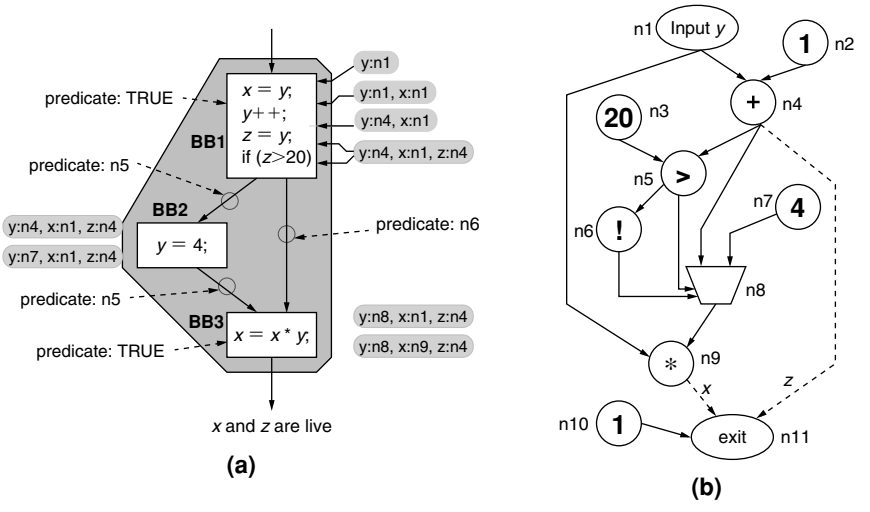

FIGURE 7.7 ■ Basic blocks selection for the hyperblock: (a) the state of the `lastDefs` list at various points in the process; (b) the resulting DFG.

copy propagation and constant propagation for free while building the DFG. At the end of processing each basic block, the final `lastDefs` list is recorded.

For a nonentry basic block B with a single predecessor in the hyperblock, the predecessor's final `lastDefs` list is used as the starting `lastDefs` list for processing B. This occurs from the end of BB1 to the start of BB2.

Building muxes

At a basic block with $N > 1$ incoming CFG edges, a given variable may have differing definitions arriving via the edges as indicated by the predecessors' respective final `lastDefs` lists. In such cases, an unencoded mux is constructed in the DFG to route the appropriate definition to subsequent consumers. An unencoded mux has N data inputs and N Boolean select inputs—only one of the select inputs can be true—and the corresponding data input is routed to the output. The N data inputs to the mux are from the data source nodes from the arriving `lastDefs` lists; the select input corresponding to each of the N data inputs is the predicate for that arriving edge. The data output of the mux structure becomes the definition of the variable entered in the `lastDefs` list for the start of processing that basic block. This occurs for `y` entering BB3, where the compiler inserts mux `n8` to select between sources `n4` and `n7`, and then makes `n8` the new entry for `y`. Because the entries for `x` and `z` are the same, however, no mux is built for either of them.

Predicates

At the beginning of processing each basic block, a node calculating that block's predicate is built if necessary and the predicate source is recorded to be used as input for nodes that cannot be executed speculatively (e.g., stores). The predicate for the hyperblock entry block is TRUE. For each other basic block, the predicate is built as the OR of the predicate sources of all incoming edges. When there is just one incoming edge, the calculation degenerates to just using that edge's predicate.

At the end of processing a basic block, a predicate is built if necessary and recorded for each outgoing edge. For a basic block ending in a conditional branch, an edge's predicate is built as its source block's predicate, ANDed with the branch condition under which that edge is taken. For a basic block ending in an unconditional branch, the edge predicate on the single outgoing edge is just the same as the block's predicate. After forming predicates for a nested if-then-else, it may be possible to simplify them; for example, a block may be (p1 AND p2) OR (p1 AND not p2), which can be reduced to just (p1) by rules of Boolean logic.

Ordering edges

To help build ordering edges, the compiler maintains lists of all loads and stores seen along any path from the entry of the hyperblock to the current point. At the start of processing the hyperblock, the lists are initialized as empty. At the end of processing each basic block, the state of the lists at that point is recorded. At the start of any nonentry basic block, the starting lists are

simply calculated: For a basic block with a single predecessor, the predecessor's lists are copied; when there are multiple predecessors, the respective lists are unioned.

When building a new load, construct an ordering edge from each upstream store to the new load, and then the load is added to the `seen_loads` list. When a new store is built, an ordering edge is constructed from each node on both the `seen_loads` and `seen_stores` lists to the new store and the store is added to the `seen_stores` list. This step is very conservative; for example, it adds an ordering edge from a store to each subsequent load even if the load is from a different array. Later phases use dependency information to remove ordering edges that are not necessary—that is, when it is guaranteed that the two accesses cannot refer to the same memory location.

Live variables at exits

This phase determines, for each exit, which values must be copied out to the next hyperblock or CPU when that exit is taken. For each such variable, a liveness edge is constructed from the node responsible for the last definition, as found in the `lastDefs` list, to the DFG exit node.

If the variable is live at that exit, there will be an entry for it in `lastDefs` at the point of exit. The indicated DFG node is the one providing the value for the variable, so the edge is constructed from that node to the exit DFG node.

Figure 7.8 shows an example of a swap. There are two exits from the first hyperblock, at one of which `a` and `b` are swapped—this results purely from

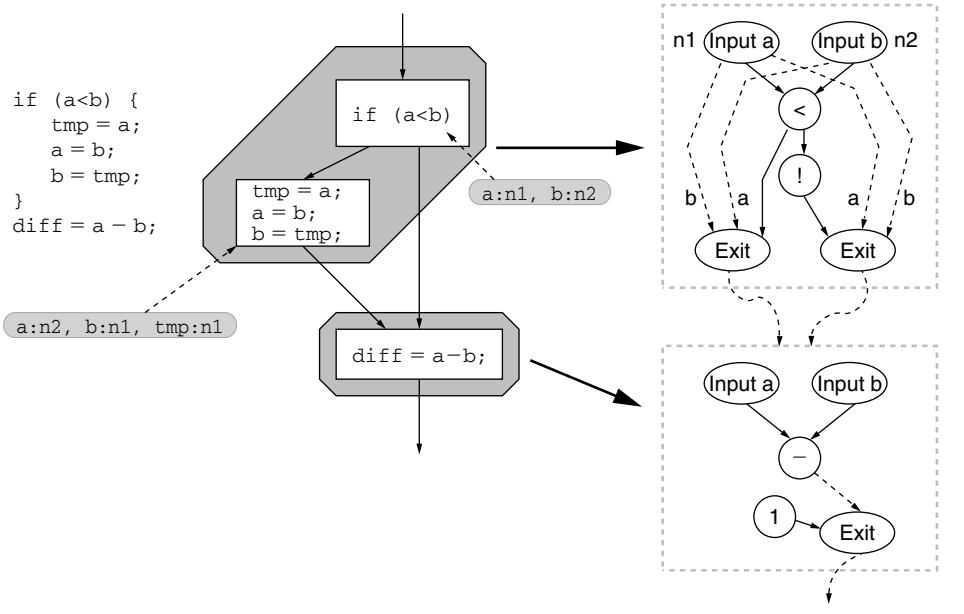

FIGURE 7.8 ■ Code, hyperblock formation, and resulting DFGs.

lastDefs list processing. The figure shows the differing contents of the lastDefs lists at the different exits. In one case, a's source is n1 (input a); in the other, its source is n2 (input b). Later, when the compiler translates the DFGs to subcircuit implementations, it will also form connections from the appropriate liveness edge sources in the first hyperbock to the input nodes in the second hyperblock.

Scalar variables in memory
If the address of a scalar variable is taken at some point by the C language & operator, it may be written or read through a pointer access. In this case, in general the variable must reside in memory. When direct accesses to the variable are interspersed with pointer accesses, we can't be sure when the pointer access might be accessing that variable without further analysis. Thus, we must keep the memory version of the variable up to date. When this situation occurs, each use of the variable requires an explicit load from memory, and each definition requires a store. Going to memory for each variable access is obviously detrimental to performance, especially on a reconfigurable fabric, so later optimizations attempt to eliminate or reduce the number of such accesses.

7.2.3 DFG Optimization

Optimizations have been performed by the compiler frontend before DFG construction even starts. More optimizations are performed during construction, some of them coming automatically in the construction process, such as constant and copy propagation. Finally, after the DFG is completed, the compiler performs many optimizations, often performing the same ones multiple times, and sometimes iterating a set of different optimizations until no further improvement occurs. We will review a few of these optimizations in the following subsections. (More detail can be found in other references; see the work of Budiu [4] and Callahan [5].) These optimizations consider the scope of the DFG (i.e., each hyperblock), which is larger than each basic block but smaller than the entire procedure.

Constant folding
Constant folding is simply the reduction of expressions of compile time constants to the equivalent constants. Its most obvious benefit is that it removes operations from the DFG and ultimately reduces area and latency in the subcircuit. A second benefit is that constant folding can enable operator specialization for other operations. (See Chapter 22.)

Figure 7.9 shows a simple example of constant folding. The important part of this example is observing how this opportunity for optimization occurs only after hyperblock formation, because the definition of x in B3 no longer interferes with constant propagation and constant folding in B1-B2-B4. This effect is not limited to constant folding, but has the potential to improve all optimizations described here.

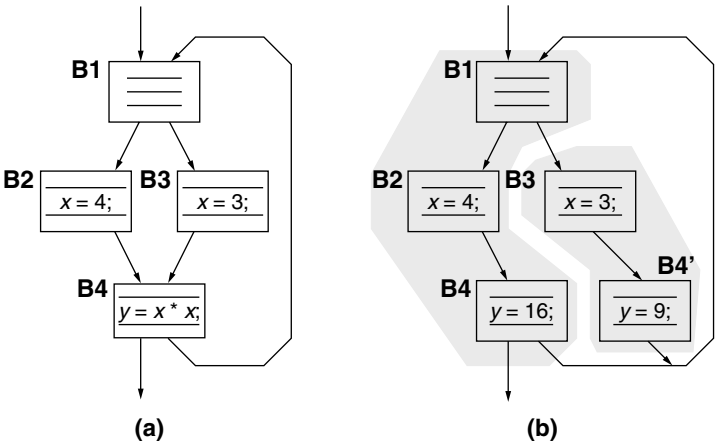

FIGURE 7.9 ■ The commonly taken path in the loop is B1-B2-B4 (a). Hyperblock formation (b)—tail duplication occurs with basic block B4. This enables constant propagation and then constant folding for the expressions `x*x`, although this is actually done after conversion to the DFG.

Identity simplification

This can be considered a special case of constant folding, that is, finding cases where the operator can be eliminated because one of the inputs is a *specific* constant. Integer operations that add or subtract zero, shift by zero, or multiply by one are eliminated. Similar optimizations exist for Boolean predicate operations: If either an OR or an AND has a constant input, it can be eliminated by replacing it either with a constant or with a pass-through from the other input.

Strength reduction

This replaces one operator with another operator (or operators) having less overall latency/area. For example, replace `x*2` with `x+x` or `x<<1`. Again, this is often based on having a specific constant input. Sometimes, equivalent implementations occur whether we do operator-level strength reduction or bit-level specialization, but it does not hurt to have multiple attacks. Multiplication by a constant is an important example because it occurs so often and because a general multiplication function unit can be expensive on a reconfigurable fabric. The expression `x*7` can be expressed as `(x<<2)+(x<<1)+x`, but even better as `(x<<3)-x`.

Dead node elimination

A cleanup pass eliminates nodes that are "dead"—that is, those that are not "live." A node is live when it is required for proper execution if (1) it has side effects (i.e., it is a store or an exit), or (2) its data output is used by another "live" node, including the case where the node supplies a live-out value to an exit node. The algorithm starts by marking as live all nodes with side effects: stores and exits. Then it marks as live any node whose data output is used by any other "live" node, and so on. Only data and liveness edges need to be traversed.

Once no more nodes can be marked as live, any remaining nodes not marked as such are known to be dead and can be safely removed.

Common subexpression elimination
Common subexpression elimination (CSE) is a well-known optimization for identifying and removing redundant computation—that is, the same operation is performed on the same operands. When a node has the same operands as another, it is immediately obvious from the structure of the graph. All simple operator nodes are subject to elimination, as are all nodes introduced to support predicated execution (Boolean calculations and muxes). Store and exit node types are not considered for elimination. Loads can be considered if additional analysis is done (see Memory access optimization subsection later).

Boolean value identification
The C language defines signed and unsigned integer data types of various sizes, but ISO C does not contain a Boolean data type [9]. Although the result of a comparison is defined to be either 0 or 1, the type of the result is a signed integer—typically 32 bits. However, no information is lost if only a single bit is used to carry the result. This can be exploited to advantage in hardware. Therefore, it is useful to identify as "Boolean" those operations guaranteed to produce only 0 or 1. When necessary for non-Boolean uses, Boolean values can be converted back to standard C type by zero-padding.

The algorithm identifies "base case" Boolean-producing nodes: comparisons, constant 0, and constant 1. Then it forward-propagates the Boolean property to nodes that have an opcode that preserves the Boolean property and that also have all inputs already flagged as Boolean. Opcodes that preserve the Boolean property include bitwise AND, OR, and XOR, as well as muxes. Opcodes that do not preserve the Boolean property include bitwise NOT and addition. However, all predicate calculations are marked as Boolean when they are constructed, including NOT operators.

For a compilation flow that eventually goes through commercial logic synthesis tools, many of the excess bits being trimmed would be trimmed eventually anyway. However, if the compiler needs to make decisions based on hardware area estimates—for example, for hardware/software partitioning—it is useful to have more accurate information about required bus and function unit width earlier in the compiler flow. This is also a motivation for the next two analyses.

Type-based operator size reduction
ISO C semantics [9] dictate that arithmetic and logical operations involving type `char` and/or `short` operands must be performed at the precision of type `int`. Figure 7.10 shows the implicit type conversions.

During initial DFG construction, all three casts are faithfully translated to DFG nodes. But since the destination's representation size of `short` (say 16 bits) is less than that of `int` (say 32 bits), the upper bits of the addition are discarded. Thus, a 16-bit adder will give the same result as a 32-bit adder in all cases, so in the intermediate representation we can signify that just a 16-bit adder

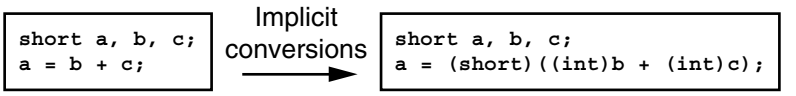

FIGURE 7.10 ■ Implicit type conversions.

is required. This in turn means that the addition uses just the lower 16 bits of each operand; thus, reducing the size of one operator may enable the size reduction of others. There also may be type conversions on the operands that can be eliminated, as shown in the figure. Besides the obvious savings in area and operator size for the addition, there are additional savings in this example: eliminating the two sign-extending type conversions on `b` and `c`.

Dataflow analysis-based operator size reduction
More detailed dataflow analyses can be performed to find the number of bits actually required by variables and operators. They may be based on range—for example, `i` within the loop `for (i = 0; i < 100; i++)`. They may also be bit level: propagating forward information about bits fixed at `0` or `1` and propagating backward information about bits not used (e.g., Budiu et al. [3]).

Memory access optimization
The handling of memory access ordering occurs in three phases:

1. The compiler conservatively adds ordering edges between pairs of memory accesses during DFG construction.
2. After DFG construction, the compiler tries to find and remove false ordering edges. Considering each pair of memory accesses connected by an ordering edge, it applies a series of tests. If any test can prove that the two operations can never access the same location during the same iteration, that ordering edge is removed. These various tests are based on array index analysis, pointer analysis, and simple testing of fixed locations (e.g., `&a` and `&b`).
3. Although removing false ordering edges is useful in itself because it exposes more parallelism and typically results in a shorter schedule, there are also many optimizations based on ordering edges that will see improved results.

Space does not allow the description of all memory optimizations that have been developed (see Callahan [5] or Budiu and Goldstein [2] for more examples), so just one will be presented here as an example.

Removing redundant loads
Consider this simple C code snippet:

```
a = *p;
*q = b;
c = *p;
```

Originally, there will be ordering edges from the first load to the store and from the store to the second load. But if subsequent pointer analysis can guarantee

that `p` and `q` can never point to the same location, those ordering edges will be removed.

The existence or absence of ordering edges is then used in the following optimization. Two loads can be reduced to one if (1) they definitely access the same location, and (2) there is no intervening store that might modify that location. Both of these requirements can be determined directly from the DFG.

To check (1), the compiler checks if the addresses of the two loads come from the same node (this assumes that common subexpression elimination has been run, which would ensure that equivalent addresses come from the same node). To check (2), we need to check for an intervening store. If there is a path from one of the loads to any store, and from that store to the other load, via ordering edges, then that store is intervening and represents a possible modification of that memory location. If both requirements hold—(1) same location and (2) no intervening store—then one of the loads can be eliminated, and its consumers can use the output of the other load. In this example, the store to `*q` was originally intervening, but is no longer after removal of the ordering edges.

7.2.4 From DFG to Reconfigurable Fabric

At this point we have an optimized DFG for each hyperblock. The final translation involves mapping DFG nodes to modules, scheduling each module to a specific timestep, and creating the simple sequencer, resulting in an actual subcircuit (RTL HDL description) for each hyperblock. Then, finally, connections are made among the sequencers and modules from different hyperblock subcircuits to complete the overall circuit.

Packing operations into clock cycles

A CPU cannot exploit the fact that a simple logical AND requires much less latency to complete than an integer addition; both take one cycle. But with spatial computing, we can pack multiple low-latency operations into a clock period (i.e., between registers) [6]. A typical example is predicate calculation, which consists of 1-bit Boolean calculations—a large subgraph of these can be performed in the time it takes to do one 32-bit addition. Another case is two successive ripple-carry adders because the latencies of their carry chains largely overlap. Additional opportunities arise from the context-specific optimization of each operation allowed by spatial computing (Chapter 22), which can greatly reduce the latency of a specific operation. On the other hand, long latency operations, such as multiplication, are typically split into stages across multiple cycles, and these stages are not considered for combining as noted before.

For simplicity it is useful to assume a target clock period from the start to get an even "packing," even if the reconfigurable platform supports a variable clock period. For systems with a fixed clock period, the upper bound is a hard limit. If the final circuit has a combinational path with latency exceeding the clock period, then some portion of the design flow must be rerun, either with more conservative decisions (for example, with operation packing) or with higher priority given to the failing paths. With a variable clock period, mistakes can be accommodated.

After this grouping, rather than a graph of operator nodes, we have a graph of modules, each of which implements one or more original DFG nodes (or a stage of a multi-cycle operation). Each module has a register at its output.

Scheduling

Scheduling a module-mapped DFG is straightforward using list scheduling. The output of list scheduling is, for each module m, an assigned slot $\sigma(m)$ when it *starts* computing. A module m's outputs are available to other modules starting at $\sigma(m) + \text{lat}(m)$, where $\text{lat}(m)$ is the latency (in clock cycles) of m (a multi-cycle operation is scheduled as a unit). In most cases this latency is one clock.

List scheduling maintains three lists of modules, and each module is a member of exactly one list. The three lists are:

- `scheduled`: modules that have already been assigned a slot. This is initialized to the input modules, all scheduled at slot 0.
- `ready`: modules whose sources have all been scheduled.
- `notready`: modules that have one or more sources not yet scheduled.

Then the list-scheduling algorithm iterates as follows until all modules have been scheduled:

1. Choose a module m from the `ready` list based on some priority heuristic.
2. Set S to the earliest cycle on which m can be scheduled, considering only when m's inputs are first all available.
3. If m has a resource conflict at slot S with any already scheduled module, increment S and go to step 3.
4. Schedule m in slot S and put it on `scheduled`.
5. Check m's successors and move them as appropriate from `notready` to `ready`.
6. If any nodes remain on `ready`, go to step 1.

Only memory operations can encounter a resource conflict in step 3, arising from the use of shared address and/or memory data buses. In contrast, any simple (nonmemory) module is scheduled as soon as all its inputs are available. Note that most such simple modules are not "actively" scheduled—they don't have an activation input from the sequencer. These passive modules simply compute a result each cycle whether or not their inputs are valid. After scheduling, the total schedule length is known, so the sequencer can be built to count off the cycles and trigger those modules that need it. The output of the final sequencer stage is ANDed with the predicate values for each exit node to create the appropriate outgoing control bit. Also, the source of each liveness edge to each exit node is translated to the appropriate connection to the input module in the destination subcircuit.

Pipelined scheduling

Here we will briefly give an idea of how pipelined scheduling works. Only hyperblocks branching to themselves to form a self-loop are considered. In the

final implementation, the key difference is that with pipelined scheduling the calculation of the control bit that is fed back to the top of the sequencer is produced not at the end of the schedule but somewhere in the middle. The result is that there are multiple '1' control bits shifting through the sequencer simultaneously, corresponding to the fact that multiple iterations of the loop are executing in an overlapped fashion. The compiler must now watch out for resource conflicts between successive iterations when scheduling the loop. The spacing between successive iterations is limited by either loop-carried data or memory dependencies, or by resource requirements. Further details are available in works by Callahan [5, 8].

Connecting memory nodes to the memory ports
Recall that each load node is split into a `load_a`, for sending the request and address, and a `load_d`, for receiving the data. Our circuit diagrams have implied that shared access to the memory port uses buses driven by tristate buffers, which some FPGAs have. But this approach could run out of tristate buffers or could restrict placement options. An alternative is to use an unencoded mux to drive each input to the shared port. For example, a mux might replace the address bus; when a memory module asserts a request to its control line of the mux, its address is routed to the mux output and to the memory port. The load data bus returning data from memory does not need any active routing; it is driven only by the memory port and fans out to all of the `load_d` modules, one of which will latch the result. However, additional buffering may be required to avoid timing problems when fanout is large.

What next?
Although we have shown the implementations as schematics, what we actually have at this point is a structural (RTL) description in an HDL such as Verilog or VHDL (Chapter 6). In a system with a commercial FPGA as its reconfigurable fabric, there is likely a fixed wrapper circuit that handles the details of connections between the compiler-generated circuit and the FPGA pins connected to the CPU and external memory. The wrapper and compiled circuit together are fed through commercial tools to perform the gate-level optimizing, mapping, placing, and routing.

7.3 USES AND VARIATIONS OF C COMPILATION TO HARDWARE

Now that we have covered the technical aspects of compiling C to hardware, we will return to higher-level programming and system-level design.

7.3.1 Automatic HW/SW Partitioning

Once we have a common source language, here C, and compilation tools that can compile a program, or parts of it, to either the CPU or the reconfigurable fabric, the remaining problem is to partition the program between the

two resources. This partitioning can be performed manually, with the user adding annotations about where to run blocks of code (e.g., loops, procedures), automatically, with the compiler making all the decisions, or some combination of the two.

Even when partitioning is manual, the use of a common source language allows rapid exploration of the design space of different HW/SW mappings. The program can be written and debugged entirely on the CPU and the programmer need only modify the allocation directives to move code onto the hardware or to change which code is allocated to it. Profiling can help the user converge on a good split.

Nonetheless, in the purely manual case the program developed ends up tuned to a specific machine, with a specific amount of hardware, specific relative speeds for the RF and the CPU, and communication between the two. Ideally, we have a single source program to run on multiple hardware platforms with varying hardware and performance. An intermediate solution is for the directives to *suggest* which software blocks might be most profitable on the RF, then to allow the compiler, perhaps with runtime feedback, to decide which of the suggested set to actually run on the hardware based on performance benefits and capacity.

Ultimately, the compiler and runtime system should take full responsibility for determining the right code and granularity to move to the reconfigurable fabric. This is an active area of research and development. Chapter 26 discusses issues and techniques for hardware/software partitioning in more detail.

The Garp C compiler [5, 7] provides an example of automatic partitioning. It starts by marking all loops as candidates for the reconfigurable fabric. Then, for each loop, it removes any paths from this candidate that include operations not supported on the array (removed paths are executed in software on the CPU). The compiler further trims the less taken paths in the loop until the remaining loop paths fit on the fabric capacity. Finally, it trims paths to improve performance. At this point, if any paths remain in the candidate loop, the compiler evaluates HW versus SW performance for the loop, considering the overhead costs for paths switching between HW and SW. If a loop is faster on the CPU, it is given a completely SW implementation. The Garp hardware supports fast configuration loads, and it caches configurations in the array, so there is a hard bound to the size of each loop but no limit on the number of accelerated loops.

For conventional FPGAs that do not support fast configuration swaps, it may be necessary to allocate all hardware logic at startup and keep them resident throughout operation. In these cases, the bound is on the total capacity of all hardware allocated to the RF, not just a single loop. The compiler may start with all feasible candidates, as in the Garp C compiler case, but then must select a subset that fits in the available capacity and maximizes performance.

7.3.2 Programmer Assistance

Useful code changes

As Section 7.2.4 shows, the compiler does many things to try to expose parallelism and optimize the implementation. However, discovering many of the

optimization opportunities requires very sophisticated analysis by the compiler, and sometimes it simply cannot prove that a particular optimization is always safe. Consequently, there are many ways a programmer might restructure or modify the application code to assist the compiler and achieve better performance on the target system. Some of these transformations have been studied to some degree in a research setting, but have not yet been fully automated in production compilers.

Loop interchange, reversal, and other transforms A loop nest can be altered in ways that still obey all required scalar and memory dependencies but that improve performance. For example, a compiler may automatically exploit memory accesses that are unit stride (A[0], A[1], A[2], ...) by streaming or prefetching. Even without explicit stream fetch support, unit stride accesses will improve cache locality, so the programmer should strive for them within the innermost loops. From one iteration to the next, loop interchange typically affects the loop-carried dependencies of the innermost loop; this impacts how effectively the block can be pipelined. If the programmer can structure the loop nest so that the innermost loop has no loop-carried dependencies, pipelining will be very effective. When the unit of HW implementation is an inner loop, another consideration is the overhead of switching between SW and HW execution. To reduce the relative cost of the overhead, it is best if possible to interchange the loops so that the innermost loops have high loop counts—as long as this does not adversely affect other aspects such as cache performance, unit stride, or loop-carried dependencies.

Loop fusion and fission Loop fusion is the combining of successive loops with identical bounds. This can remove memory accesses if the second loop loads values written by the first loop; instead, the value can be passed directly within the fused loop. The reverse, loop fission (splitting one loop into two), can also be useful when the original loop cannot fit in its entirety on the reconfigurable resources. Afterward, the two halves can each fit, but not at the same time, so temporary arrays may need to be introduced to store data produced in the first half and used in the second.

Local arrays When an array is local to a procedure and of fixed size, it is relatively easy for the compiler to do the "smart thing" and implement it using a memory block on the FPGA fabric. But if the program instead uses `malloc`'d or global arrays as temporaries, it is very challenging to safely convert them to local arrays. Thus, changing the code to use local arrays wherever possible can be very useful because on-FPGA memory blocks have much lower latency to/from the computation unit and can be accessed in parallel with each other.

Control structure Most compliers keep the loop, procedure, and block structure in the original code. As noted previously, common heuristics for hardware/software partitioning select loop bodies or procedures as candidates for hardware implementation. If the loop is too large, it may not be feasible on

the array. If the loop is too small, it might not make good use of the array's parallelism. The programmer can often assist the compiler by sizing and organizing loops, procedures, and blocks that make good candidates for hardware allocation.

Address indirection As noted in Section 7.2.3, whenever the address of a variable is taken, the compiler must make conservative assumptions about when the value will be updated, forcing additional sequentialization and increasing memory traffic. Consequently, address indirection and pass-by-reference should be used judiciously with the realization that it can inhibit compiler optimizations. Note that this unfortunate effect can also occur when a global scalar variable is visible beyond the file in which it is declared; with separate compilation, the compiler must assume that code in some other file takes the address of the variable and passes it back as a pointer. Therefore, declaring file-global variables as static helps as well.

Declaration of data sizes On CPUs there is often little advantage to using a narrow data word. Except for low-cost embedded systems, all processors have at least 32-bit words, with high-performance processors trending to 64 bit; even DSPs and embedded processors can typically assume CPUs with at least 16-bit words. Consequently, there is little incentive to software programmers to pay much attention to the actual range of data used. However, in fine-grained reconfigurable fabrics, such as field-programmable gate arrays (FPGAs), narrow data words can be implemented with less area and, sometimes, with less delay. As noted in Section 7.2.3, the compiler can make use of narrower type declarations (e.g., `short`, `char`) to reduce operator size.

Useful annotations

A programmer annotation gives the compiler a guarantee about a certain property of the program, which typically allows the compiler to make more aggressive optimizations; however, if the programmer is in error and the guarantee does not hold in all cases, incorrect program behavior may result. Some annotations can be expressed as assertions. If the assertion fails, the program will terminate, signaling the user (hopefully, the programmer) that the assertion was violated. The compiler knows that when execution continues past the assertion, certain properties must hold.

Annotations and assertions can be used as ways to communicate information to the compiler that it is not capable of inferring itself. In this way they may be an alternative to very advanced compiler analysis, or a complement when the analysis is simply intractable. Following are two examples of useful annotations:

- *Pointer independence:* declaring that a pair of pointers will never point to the same location, so that an ordering edge between accesses using those pointers can always be removed safely.
- *Absence of loop-carried memory dependences:* declaring that the memory operations in different iterations of the loop are always independent (to

different locations), which typically allows much greater overlap and greater performance when using pipelined scheduling.

Integrating operator-level modules

Even when writing C code for CPUs, the compiler does not always generate optimal machine code, and it is occasionally necessary to write assembly code for key routines. Similarly, when the C compiler does not provide the tight implementations of which the RF is capable, it may be necessary to provide a direct hardware implementation. Here, the "assembly" may be a VHDL (Chapter 6) implementation of a function or a piece of dataflow. As in the assembly language case, the developer can start with a pure C program profile, the code, and then judiciously spend his customization effort on the code's most performance-critical regions.

It is fairly easy to integrate a custom operation into the flow we have described. The designer simply needs to create the module via HDL or schematic capture, and tell the compiler the latency, in cycles, of the design. The operation can be accessed from C source code using function call syntax, instantiated, and scheduled in parallel with other "native" C operations in the hyperblock. For example, in this code snippet:

```
x = bitreverse(a);
y = a ^ b;
z = x + y;
```

the `bitreverse` module would have one cycle latency and could be scheduled in parallel with the XOR (`^`) module.

The power of this approach is greatly increased with a *module generator*. In this case, the HDL module is not just copied from a library; instead, it is dynamically generated by the compiler. This allows constant arguments to the module instantiation to specialize it, for example,

```
X = bit_reverse_range(a,8,15);
```

which will generate a module that will reverse the bits of `a` from bit 8 to bit 15 to produce `x`. A detailed interface between compiler and dynamic module generator is described in work by Koch [10] (see also Chapter 15).

It is useful to always have a functionally equivalent software implementation of each custom operation in order to enable testing of the overall application in a pure software environment. This is required, for example, when adding hand-designed HDL modules in the SRC Computers compiler [14].

Integrating large blocks

Another method for integrating a hand-designed circuit with an otherwise C-compiled program is to treat it as its own hyperblock subcircuit within the compiler, allowing it to manage its own sequencing. The HDL implementation of the custom block in this case receives a `start` control bit, like any other hyperblock, and must send a finish control bit when done. This allows the designer to incorporate custom blocks that have variable latency (e.g., an iterative divider or

a greatest-common-divisor computation). The programmer could use function call syntax to instantiate this larger block as well, but, the compiler would prevent the function from being merged with other blocks into a larger hyperblock.

7.4 SUMMARY

After a decade of research, C compilation for reconfigurable computers is now commercially available in many forms (e.g., SRC Computers [14] and Lau and colleagues [11]). While today's commercial compilers cannot generally compile arbitrary ISO C code or take arbitrary C code and expect to fully extract the performance of the reconfigurable fabric, they have closed the gap so that nontrivial code acceleration is possible with minor programmer effort. A developer can use the C compiler to rapidly get applications running on a suitable reconfigurable platform. C code developed or tuned with an understanding of the reconfigurable platform and the capabilities of the compiler can achieve higher performance. Although today's C compilers do not free the reconfigurable developer from understanding good application and system architectures, they can allow her to focus her efforts.

C compilation and optimization remain an active area of research, and we expect to see continuing improvements over time. Many opportunities exist for innovative research on aggressive optimization techniques and development of more automated optimizing complier flows.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] M. Budiu, S. Copen Goldstein. Optimizing memory accesses for spatial computation. *International ACM/IEEE Symposium on Code Generation and Optimization*, March 2003.

[3] M. Budiu, M. Sakr, K. Walker, S. Copen Goldstein. Bit value inference: Detecting and exploiting narrow bit-width computations. *European Conference on Parallel Processing*, Springer-Verlag, 2000.

[4] M. Budiu. *Spatial Computation*, Ph.D. thesis, Carnegie-Mellon University, December 2003 (technical report CMU-CS-03-217).

[5] T. J. Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*, Ph.D. thesis, University of California, Berkeley, December 2002.

[6] T. J. Callahan, P. Chong, A. DeHon, J. Wawrzynek. Rapid module mapping and placement for FPGAs. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1998.

[7] T. Callahan, J. Hauser, J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer* 33(4), April 2000.

[8] T. Callahan, J. Wawrzynek. Adapting software pipelining for reconfigurable computing. *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2000.

[9] P. Harbison, G. L. Steele. *C, A Reference Manual*, 4th ed. Prentice-Hall, 1995.

[10] A. Koch. Compilation for adaptive computing systems using complex parameterized hardware objects. *Journal of Supercomputing* 21(2), 2002.

[11] D. Lau, O Pritchard, P. Molson. Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2006.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] SRC Computers. *SRC Carte C Programming Environment v2.2 Guide*, Colorado Springs, 2007.