

Specifying and Compiling Applications for RaPiD*

Darren C. Cronquist, Paul Franklin, Stefan G. Berg, and Carl Ebeling

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Abstract

Efficient, deeply pipelined implementations exist for a wide variety of important computation-intensive applications, and many special-purpose hardware machines have been built that take advantage of these pipelined computation structures. While these implementations achieve high performance, this comes at the expense of flexibility. On the other hand, flexible architectures proposed thus far have not been very efficient. RaPiD is a reconfigurable pipelined datapath architecture designed to provide a combination of performance and flexibility for a variety of applications. It uses a combination of static and dynamic control to efficiently implement pipelined computations. This control, however, is very complicated; specifying a computation's control circuitry directly would be prohibitively difficult.

This paper describes how specifications of a pipelined computation in a suitably high-level language are compiled into the control required to implement that computation in the RaPiD architecture. The compiler extracts a statically configured datapath from this description, identifies the dynamic control signals required to execute the computation, and then produces the control program and decoding structure that generates these dynamic control signals.

1 Introduction

The RaPiD architecture is a field-programmable architecture that allows pipelined computational structures to be constructed from an array of arithmetic units, registers and memories. These are interconnected and controlled using a combination of static control, which does not change during the computation, and dynamic control, which does. This paper deals with the problems of specifying linear pipelined computations and compiling a specification into the combination of static configuration and dynamic control required to program the RaPiD architecture.

We begin with a programming language that allows a pipelined computation to be described in both time and space. Specific operations are assigned to a specific pipeline stage at a specific time. Time is described using nested loops, while space is described by the innermost loop. Each iteration of this innermost loop is allocated to a specific pipeline stage at a specific time. Since pipelined computations are both regular and repetitive, descriptions in this form are usually quite concise.

When compiling a program, we take the classic approach of partitioning the implementation into datapath and control. The program describes the operations performed by each pipeline stage in each cycle. These operations determine the underlying pipelined datapath. Typically this datapath has a number of dynamic controls to change the functionality and interconnection of elements in the datapath during the computation. These controls are decoded from instruction bits passed down the array, which are in turn produced by a control program. Since each application needs different control, each will use the instruction bits and decoding structure differently, and will have its own control program.

Although the compiler was designed specifically for RaPiD, we believe the RaPiD-C language provides a clean and effective way to specify pipelined computations. For example, a different back-end to our compiler could be used to generate implementations in different technologies such as FPGAs or custom ASICs. In fact, we see the RaPiD architecture model used in a variety of ways. One possibility is to create a single very flexible implementation that could be used for a wide variety of different problems. This implementation would include a set of generic functional units and a very flexible set of interconnection resources. Another possibility would be to create a "custom" RaPiD implementation tailored specifically for one predetermined set of computations. This implementation would trade flexibility for reduced cost.

We begin by giving a brief overview of the RaPiD architecture model. We then present the matrix multiply application to motivate the approach we have taken for language design and compilation. Next, we present RaPiD-C language features and describe how

*This work was supported in part by the Defense Advanced Research Projects Agency under Contract DAAH04-94-G0272. D. Cronquist was supported in part by a Gray fellowship. P. Franklin was supported in part by an NSF fellowship.

programs are written using the language constructs. Finally, we describe the compilation process used to generate and optimize datapath and its control.

2 The RaPiD Architecture

This section provides a brief overview of the architectural details of RaPiD which directly affect the compilation process. For a more thorough description of the architecture, see [1] and [2].

RaPiD is a coarse-grained field-programmable architecture for compute intensive applications. The architecture consists of an abundance of functional units such as ALUs and multipliers as well as general purpose registers (GP-REGs) and RAMs. As an example, a version of the architecture that we have used for benchmarking contains 96 GP-REGs, 48 32-entry RAMs, 48 ALUs, and 16 multipliers, all supporting 16-bit data operands. This benchmark version is approximately 100mm² in a .5 u process and runs conservatively at 100 MHz.

Such a large number of functional units must be interconnected in a cost-effective manner. Although a crossbar would provide the greatest flexibility, the myriad of functional units requiring connections—over 200 in the benchmark version—make this approach infeasible. Instead, RaPiD arranges the functional units linearly above a field-programmable segmented bus structure. A linear structure is easily manageable, yet it reduces implementation cost and control requirements tremendously. By using the small RAMs spread throughout the array as buffers, multidimensional tasks can be performed on RaPiD arrays. The underlying datapath, i.e. which functional units can forward results to each other, is configured statically on a per application basis. During the execution of an application, the data movement between functional units can change every cycle via a decoded instruction.

A simple example is shown in Figure 1. A register is used to hold a constant value, such as a coefficient for a FIR filter. The underlying datapath is statically configured so the register can load from either an input stream or its previous value. During the execution of the instruction stream, dynamic control directs data movement on a cycle-by-cycle basis. In summary, *static control* determines the extent that data can flow in a given application. *Dynamic control* determines the cycle by cycle data movement under the restrictions placed by static control. For example, the dynamic control would specify when the register should load from the input stream and when it should hold its value by loading from the feedback path.

The RaPiD architecture provides *hard* control signals, which are fixed by the configuration data, and *soft* control signals, which can change each clock cycle. Soft control signals drive ALU functions, input and output stream enables, RAM writes and increments, and multiplexer selects. As a result, RaPiD

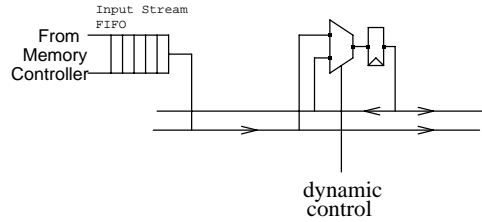


Figure 1: A register configured to load or hold its value for a constant multiply. The interconnect shown is configured statically but the dynamic control signals can change every cycle.

contains a substantial number of soft control signals—over 1000 in the benchmark version. To reduce the circuitry required to generate these signals, a pipelined control path that parallels the datapath is used to generate these signals from a narrow instruction (eg. 32 bits) inserted at the beginning of the array. This “instruction” contains all the information required by the various pipeline stages to compute their dynamic control signals. The control path comprises a set of 1-bit segmented busses similar in structure to the datapath busses, as shown in Figure 2. This bus structure allows the instruction bits to be individually manipulated as they proceed down the RaPiD array.

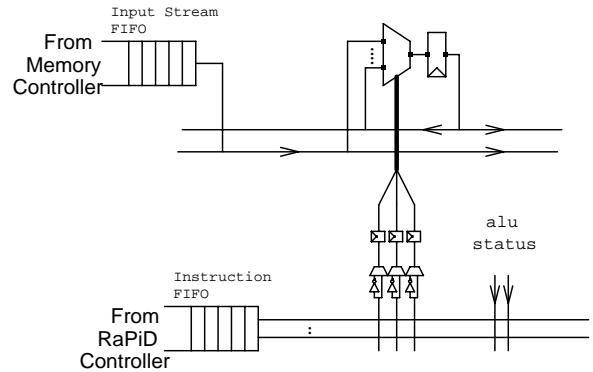


Figure 2: Soft control circuitry used to dynamically control the circuit shown in Figure 1

The majority of soft control bits are static for a given application, and are wired to a constant 0 or 1. Other bits are wired to one of the other control wires. This control often comes directly from the controller, pipelined as needed. However, more complex decoding using 3-input look-up tables (LUTs) can be used to decode several instruction bits into the appropriate control or to combine pipelined control with status information (e.g. ALU carry). The LUTs also contain optional registers, allowing for simple finite state machines (FSMs) occasionally required by non-pipelined control. Such FSMs can be used to activate a function for several cycles in each stage, one stage at a time.

If RaPiD supported deep pipelining within the control path, this could be done easily by placing enough registers between each stage. Instead, this can be constructed using a FSM; a stage can remember whether or not its RAM is active, and one instruction bit can be used to deactivate one stage and activate the next, requiring only two control lines. This is used in implementing a 2-D DCT on RaPiD [2].

The number of busses required in the control path varies by application, but is not large because control signals tend to be reused extensively. The benchmark version of the RaPiD architecture provides 31 busses, which can be pipelined and otherwise manipulated individually. This is more than enough for the current set of applications, even using non-optimal mappings produced by automated CAD tools.

2.1 Datapath Controller

The RaPiD array consumes one instruction per cycle which drives the beginning of the control buses. Generating this instruction stream is nontrivial because it often controls several tasks running in parallel; the matrix multiply example in the next section illustrates this. The RaPiD datapath controller contains several simple microcontrollers (instruction generators) whose output is combined to form instructions for the RaPiD array.

Each instruction generator executes a microprogram to generate a stream of instructions. The generators are optimized to handle nested and sequential loops; they also contain microinstructions for performing simple arithmetic synchronization.

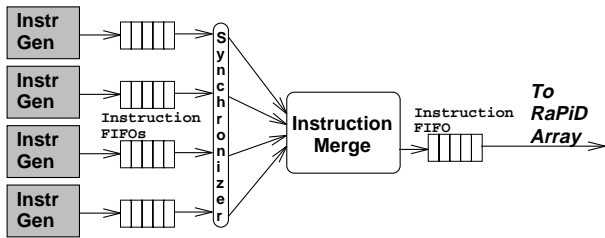


Figure 3: *The RaPiD controller.*

The instruction streams produced by the instruction generators are synchronized via SIGNAL and WAIT tags. The stream containing a WAIT tag is stalled until the matching SIGNAL occurs in another stream. After the instruction streams have been synchronized, they are combined in a bitwise fashion, and the final instruction then passes through the last instruction FIFO and proceeds to the RaPiD array.

RaPiD also contains address generators used for generating sequences of memory addresses used by the memory controller (Figure 4). The address generators use arithmetic microinstructions to produce address

sequences. The memory controller handles these memory requests by placing data in or removing data from the appropriate input or output FIFO.

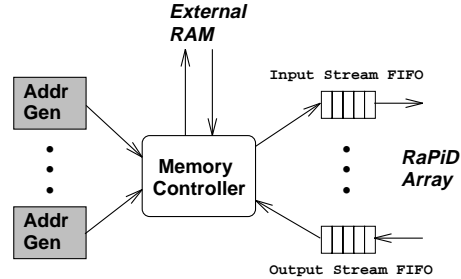


Figure 4: *The RaPiD address generators and I/O streams.*

3 Specification

To map an algorithm to the RaPiD array we have designed a new parallel programming language, RaPiD-C. Although created with RaPiD in mind, the methods of specification and compilation could be extended to other architectures for implementing pipelined computations. In addition, this programming technique could be used for ASIC design.

3.1 Motivation: Matrix Multiply

To motivate a specification language, we first look at a common, well-studied application—matrix multiply. The problem takes an $L \times M$ matrix \mathbf{A} and an $M \times N$ matrix \mathbf{B} and computes the $L \times N$ matrix $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, as shown in the nested-loop specification of Figure 5. As it stands, this high-level specification is far from a

```

for (i=0; i < L; i++)
  for (j=0; j < M; j++)
    for (k=0; k < N; k++)
      if (j==0) C[i][k] = A[i][j]*B[j][k];
      else C[i][k] += A[i][j]*B[j][k];

```

Figure 5: *Nested loop specification for matrix multiply*

mapping to a pipelined linear array. In particular, the parallelism and the data I/O are not specified, and the algorithm must be partitioned to fit on the target architecture.

Automating these processes is a difficult problem for an arbitrary specification, and one we do not solve here. Instead, we propose a language that is C-like and requires the programmer to specify the parallelism, data movement, and partitioning. To this end, the programmer uses well known techniques of loop transformation [5] and space/time mapping [4, 3]. The resulting specification is a nested loop where outer loops

specify time and the innermost loop space.¹ In the context of RaPiD-C, the space loop refers to a loop over the *stages* of an algorithm, where a stage is one iteration of the innermost loop. The compiler maps the entire stage loop to the target architecture by unrolling the loop to form a flat netlist. Hence, the stage loop, also called a **Sloop**, has implicit parallelism since it executes in a single cycle on the target architecture.² As a result, the programmer must permute and tile the loop-nest so that the computation required after unrolling the innermost loop will fit onto the target architecture. The remainder of the loop nest determines the number of times the **Sloop** is executed.

The programmer first transforms the original specification by choosing a loop to iterate over the stages, optimizing for speed, memory, scalability, etc. Partitioning of the algorithm is based on the number of functional units and available memory in the target architecture. For example, consider mapping matrix multiply to an architecture with S multipliers, at least 3 RAMs per multiplier, and R words of memory per RAM. The innermost loop is partitioned by the number of stages (i.e. multipliers) and the outer loops by the size of the RAMs. Since the stage loop is the innermost loop, k has been replaced by the stage iteration variable s . Loop permutation is applied, yielding the code in Figure 6.³ From now on, instead of explicitly stating the innermost loop as **for** ($s=0$; $s < S$; $s++$), we will simply write **Sloop**.

```

for (f=0; f < L; f+=R)
  for (g=0; g < M; g+=R)
    for (h=0; h < N; h+=S)
      for (i=0; i < R; i++)
        for (j=0; j < R; j++)
          for (s=0; s < S; s++)
            if (j+g==0) C[i+f][s+h] = A[i+f][j+g]*B[j+g][s+h];
            else C[i+f][s+h] += A[i+f][j+g]*B[j+g][s+h];

```

Figure 6: *Matrix multiply partitioned to space and time*

Memory accesses are determined by examining indexing in every stage on every cycle (recall that the **Sloop** executes in a single cycle). Since $A[i+f][j+g]$ is independent of s , the appropriate **A** matrix value will be broadcast to the entire array on every cycle. Expression $B[j+g][s+h]$ references R elements (the length of the j loop) of the **B** matrix in each stage, which can be stored in a RAM in each stage. Moreover, since h changes every R^2 cycles, a new set of elements must be loaded every R^2 cycles. To prevent the array from stalling, the RAMs can be double-buffered. Finally, expression $C[i+f][s+h]$ references R elements (the length of the i loop) of the **C** matrix in

¹Since RaPiD is a linear architecture we have a singly nested loop for space. An n -dimensional architecture would have an n -nested loop dedicated to space.

²In actuality, pipelining and retiming usually cause the **Sloop** to be executed on a diagonal of the time axis instead of the same cycle.

³For ease of presentation, we assume that R divides L and M , and that S divides N .

each stage, which again can be stored in a RAM in each stage. Since every stage produces a result on the same cycle, these results are pipelined down the array instead of being broadcast; the final stage stores the values from this pipeline into the **C** memory.

Matrix multiply can now be broken down into three processes: preload, computation, and output. These processes all run in parallel but need to be precisely synchronized to allow them to communicate. The preload loop-nest must complete one iteration of its h -loop before the computation loop-nest begins. The computation loop-nest must complete one iteration of its g -loop before the output loop-nest can begin. To simplify this specification, the language supports parallel loop nodes and Signal/Wait synchronization pairs. The double buffering of the **B** values is performed by a two-dimensional array of $S \times 2$ RAMs which is indexed by the stage number and a boolean toggle bit to flip between the RAMs on the appropriate cycle. This sums up the major features of RaPiD-C, which is described in more detail in the next section. Pseudo-code for the RaPiD-C implementation of matrix multiply is shown in Figure 7. To clarify the structure of this code, we often write a control tree as shown in Figure 8. Control trees will be described in detail in the next section.

```

Par {
  // Preload loop-nest
  for (f1=0; f1 < L; f1+=R)
    for (g1=0; g1 < M; g1+=R)
      for (h1=0; h1 < N; h1+=S) {
        for (j1=0; j1 < R; j1++)
          for (i1=0; i1 < S; i1++)
            Sloop
              if (i1==s) ramB[s][!toggle][j1] = B[j1+g1][s+h1];
              Signal(comp); Wait(preload);
            }
      }
  // Computation loop-nest
  for (f2=0; f2 < L; f2+=R)
    for (g2=0; g2 < M; g2+=R)
      for (h2=0; h2 < N; h2+=S) {
        Signal(preload); Wait(comp);
        for (i2=0; i2 < R; i2++)
          for (j2=0; j2 < R; j2++)
            Sloop {
              if (i2==0 && j2==0) toggle = !toggle;
              if (j2==0 && g2==0)
                ramC[s](i2) = A[i2+f2][j2+g2]*ramB[s][toggle](j2);
              else
                ramC[s](i2) += A[i2+f2][j2+g2]*ramB[s][toggle](j2);
              if (j2==R-1 && g2==M-R) {
                pipeOut = ramC[s](i2);
                Signal(output);
              }
            }
      }
  // Output loop-nest
  for (f3=0; f3 < L; f3+=R)
    for (h3=0; h3 < N; h3+=S)
      for (i3=0; i3 < R; i3++) {
        Wait(output);
        for (j3=S-1; j3 ≥ 0; j3--)
          Sloop
            if (s==S-1) C[i3+f3][h3+j3] = pipeOut;
          }
      }
}

```

Figure 7: *Matrix multiply after I/O, ram allocation*

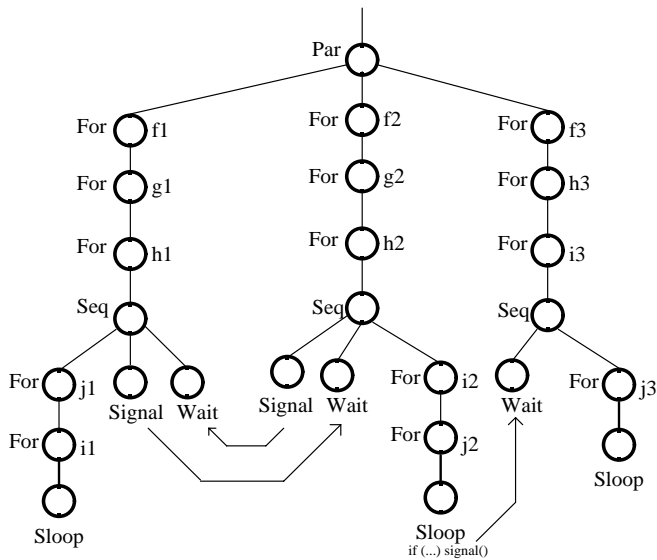


Figure 8: Control tree for matrix multiply

3.2 Control Trees

Control trees are a convenient representation of a RaPiD-C program’s loop structure. They are particularly useful while manipulating a program’s structure by performing loop transformations, and as an aid to explaining a program’s control structure. This section presents them, while using them to explain the control structures available in RaPiD-C.

RaPiD-C uses a control tree to specify the loop structure of a particular application. In a complete RaPiD-C program, this tree is part of the code, as shown in Figure 7; however, while determining what a program’s control tree should look like, it is often useful to draw it separately. This section uses control trees to reintroduce the control constructs already shown above.

Figure 9 shows two simple RaPiD-C trees. Tree (a) represents two nested loops in time. The inner loop contains the stages loop, or `Sloop`. This loop automatically iterates the reserved variable `s` over all of the stages ($0 \leq s < S$). Tree (b) illustrates a computation split into two loops; the `j` loop will begin after the `i` loop completes. Note that each branch has its own `Sloop`.

Each `Sloop` in a control tree contains code to be compiled to the target architecture. Inside `Sloop` blocks, a programmer uses a C-like syntax with special objects representing some features of the architecture. Since each block actually executes `S` times, conditional statements can check the value of `s` to restrict code segments to specific stages. Conditionals can also compare against a `For` node’s iteration count such as `i==3`. In addition, the conditionals `.first`, `.last`, or `.live` test on the first, last, or any iteration

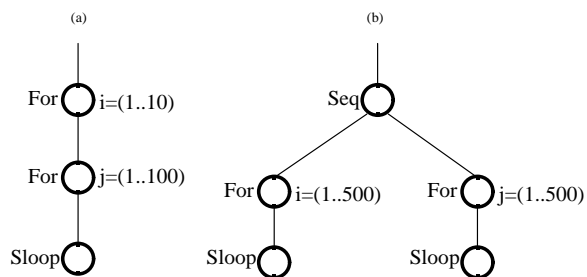


Figure 9: Simple RaPiD-C control trees. (a) Nested loops. (b) Loops to be run in sequence.

of a loop, respectively. Code which needed to be executed every 100th iteration can easily be coded in the `Sloop` for Figure 9(a) with the condition `j.last`.

Note that a condition specifying every 100th iteration of Figure 9(b) is more complex. Control trees should represent the actual control needed by an application. The RaPiD-C code for matrix multiply shown in Figure 7 contains only relatively simple conditions, indicating that its control tree captures the loop and control structure of the application.

RaPiD-C uses `Par` nodes to indicate branches which should run in parallel. RaPiD-C also contains synchronization primitives; a `Wait` node stalls until it receives a signal from either a `Signal` node or a signal statement in a `Sloop`. Figure 10 shows a control tree in which the right `Sloop` will start executing as the left `Sloop` begins its second iteration.

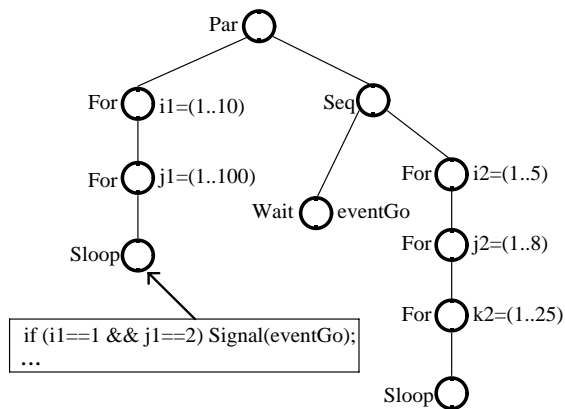


Figure 10: A `Signal/Wait` pair to run two nested loops in parallel, offset by one cycle

Control trees can also contain `Inf` nodes; they are similar to `For` nodes but execute on every cycle, halting immediately when all other control is exhausted. Table 1 summarizes the node types presented in this section.

Table 1: *Control tree node types in RaPiD-C*

Node Type	Children Execution	Length (in cycles)
Seq	In sequence, one at a time	Sum of lengths of children
Par	In parallel, all starting simultaneously	Length of longest child
For	Its single child, loop iteration times	# of iterations * child length
Inf	Its single child, many times	Number of cycles in entire tree
Sloop	No children	One cycle
Wait	No children	Until signaled
Signal	No children	Zero cycles

Table 2: *Data types in RaPiD-C*

Data Type	Specifies
Word	Single width variable
Long	Double width variable
Bool	Single bit value, used for conditional statements
Ram	Fixed size RAM local to a stage
Pipe	Inter-stage communicator

3.3 Communication

A RaPiD-C application needs to communicate among its stages and with the outside world. This is provided for with separate mechanisms. Communication with the outside world is done through array references. Inter-stage communication is accomplished using pipes that connect a number of stages together.

The programmer can specify an arbitrary number of external arrays that can be read from or written to inside a RaPiD-C program (see arrays A, B, and C in Figure 6). A limitation imposed on the programmer is that all references must be data independent since memory addressing is determined at compile time.

Pipes are used to communicate values between stages. A pipe is just a global bus with a number of optional registers between stages. The programmer can therefore use them to feed data into or out of the array or to communicate intermediate results from one stage to the next. Figure 7 shows an example of a pipeline used for writing the result matrix to the external array C. All stages output their final results to `pipeOut` at regular intervals. The last stage reads from `pipeOut` and stores the read values in array C.

3.4 More RaPiD-C Types

RaPiD-C has several predefined types to support both computation and data communication within an algorithm, as shown in Table 2.

The types `Word` and `Long` represent the single and double precision data types for use within a stage. For computation that is similar across several stages, typically an array of `Words` or `Longs` is defined.

Type type `Bool` is used for control defined by the programmer. For example, double buffering two RAMs requires a toggle signal. For example, the code `if (i2==0 && j2==0) toggle = !toggle;`

Table 3: *Operators for RAMs.*

Operator	Action
<code>ramFoo.address = x</code>	Set the address register to value x, mod size
<code>ramFoo.address++</code>	Increment the address register, mod size
<code>ramFoo = y</code>	Set the ram value for the current address to y (y is of type word)
<code>y = ramFoo</code>	Read the value for the current address
<code>ramFoo(i)</code>	Automatically address ram with respect to For loop <i>i</i>

flips the toggle bit on the appropriate cycle in matrix multiply (see Figure 7).

The type `Ram` specifies a fixed-size local memory in a stage of the target architecture. `Ram` is accessed via an implicit address register that can be assigned, cleared and incremented. Table 3 lists the valid operators on a ram.

The `Ram` type represents an architecture-specific device. When specifying applications targeted at other architectures, other architecture-specific types might be called for.

4 Compilation

A RaPiD-C program clearly specifies an algorithm’s hardware requirements. As a matter of fact, the union of all `Sloop` blocks is very close to a structural description of the algorithm. One difference from a true structural description is that `Sloop` statements are specified sequentially but execute in parallel. A netlist must be generated to maintain these sequential semantics in a parallel environment. Another difference is that control is not explicit but instead embedded in a nested-loop structure. This control must be extracted into multiplexer select lines and functional unit control. Then, an instruction stream must be generated which can be decoded to form this control. A final difference from a true structural description is the implicit decoupling of data I/O. Address generators must be instantiated to take the data to and from memory at the appropriate time. Hence, compiling RaPiD-C into a structural description consists of four components: netlist generation, dynamic control extraction, instruction stream/decoder generation, and I/O address generation.

The compilation process produces a structural specification consisting entirely of components on the target architecture. The netlist is then mapped to the architecture via standard FPGA mapping techniques including pipelining, retiming, and place and route. The place and route solution fully specifies the static setting required to program the array.

4.1 Netlist Generation

Generating a parallel netlist from a sequential specification is straightforward. Consider the three types of

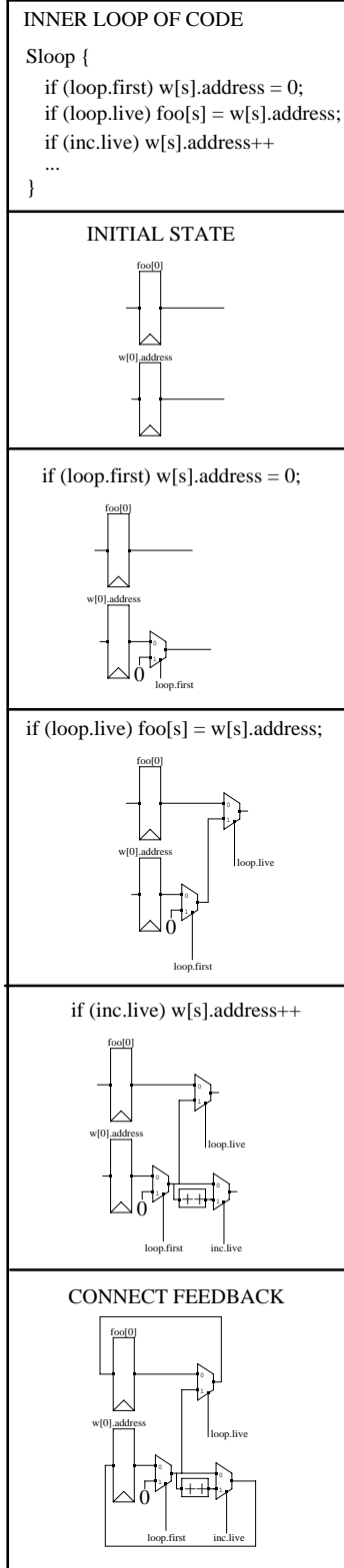


Figure 11: *Generation of a netlist from RaPiD-C.*

data dependencies found in sequential code: true (read after write), anti (write after read), and output (write after write). The idea is to convert variables into registers, noting that if a register is read and written on the same cycle, the register's value on the previous cycle is read. An anti-dependence requires the previous value of a variable be read, so a register is sufficient. A true dependence requires the current value of a variable to be read, so data forwarding is used. Finally, an output-dependence is implemented with a register whose input multiplexer gives priority to the latest write in the sequential code.

The compiler converts a RaPiD-C program into a structural specification by interpreting the union of all `Sloop` blocks for each value of `s`. During interpretation, the compiler instantiates registers for variables, ALUs for adds (and other operations), multipliers for multiplies, and multiplexers for `if-then-else` statements. Once interpretation is complete, the final value of each variable is connected to the input of the variable's register, creating state across cycles.

For example, Figure 11 shows the netlist construction for a small set of `Sloop` statements. During the first iteration of the loop, `s` is assigned to zero, creating the variables `foo[0]` and `w[0].address` which are initialized as registers. The first line of code adds a multiplexer to the address register which either holds its current value or loads zero, depending on the value of `loop.first`. The second line updates `foo[0]` to be the *current* value of the address register, if `loop.live` is true. The third line instantiates an incrementer and updates the value of the address register if `inc.live` is true. After the final reference to the variables `foo[0]` and `w[0].address`, the current values are connected as inputs to the original registers, providing support for dependencies across iterations of the control tree.

4.2 Dynamic Control Extraction

Dynamic control can take many forms depending on the versatility of the target architecture. The most common dynamic signals are used to time data movement, as in a multiplexer's select lines, and to specify a change in computation, as in a functional unit's operation signals. As a result, two key steps are required to generate dynamic control: multiplexer merging and functional unit merging.

4.2.1 Multiplexer Merging

The initial netlist generation forms two-input multiplexers for every conditional statement. These smaller multiplexers must be merged to match the size of multiplexers on the target architecture, potentially changing the required control.

To merge multiplexers a depth-first search of the initial two-input multiplexer netlist is performed starting at the output streams, since all required functional

units must be reachable from the output. When a previously unreachable functional unit is found, each input is replaced with a 16:1 multiplexer (or whatever size corresponds to the target architecture). The compiler then performs a depth-first search to determine reachable inputs. An input's select condition is the AND of multiplexer conditions along this path. Figure 12 shows the code and a portion of a generated netlist containing three multiplexers which are merged into a single multiplexer with three inputs. The REG1 input is found to be unreachable since the global context becomes $b \wedge a \wedge \neg a = 0$.

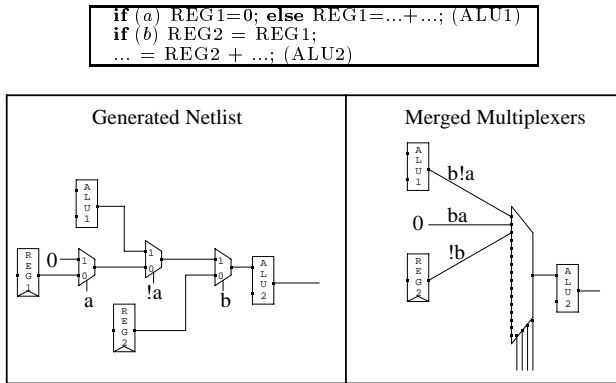


Figure 12: An example of multiplexer merging: three 2-input muxes are merged into a single mux.

4.2.2 Functional Unit Merging

RaPiD-C uses symbols, such as $*$ and $+$, to specify operations on data. Clearly, if the programmer uses a common subexpression, the compiler must be robust enough to map all instances of the expression to the same functional unit. Although a standard common-subexpression elimination algorithm would work for common-subexpressions, the dynamic control of the RaPiD array can optimize some *uncommon* subexpressions. For example, the expressions $x-y$ and $x+y$ are different but could be mapped to the same ALU if they are not both in use during a common cycle. This would require a control signal to be generated to change the ALU function from an addition to a subtraction on the appropriate cycle.

Even expressions with different operands can be merged. If the expressions $x+y$ and $w-z$ are not in use during a common cycle, they could be merged by having both x and y reach one ALU multiplexer input, and both w and z the other. Now three dynamic control signals must be used to change between the two expressions. Since all three signals are equivalent, this second example doesn't use more control path resources than the first. However, the underlying

netlist (static control) becomes more complex, potentially stressing the available routing resources in the datapath; in some cases, the control becomes complex enough that the merging must be rejected.

To support both common and uncommon subexpression elimination, a list of functional units is maintained in every stage. In addition, a boolean *in-use* function is created for each functional unit to record the cycles of operation. The functional units' in-use functions determine when merging can occur. Two functional units can be merged if they are identical (i.e. a common-subexpression), or if their in-use functions are mutually exclusive and the union of their inputs doesn't exceed some internal maximum. This maximum is n for n -input multiplexers but clearly should be substantially smaller due to routing constraints. When there is a choice of functional units to merge, the pair with the larger number of common inputs is selected. Functional units which use static control to determine their functionality must be equivalent in these bits to be merged.

For example, consider an ALU with two data inputs (Left and Right) and four control inputs (F3, F2, F1, and F0). The in-use function, *InUse*, determines when the ALU is actually needed. If we are given *alu1* and *alu2* with mutually exclusive *InUse* functions, they can be merged into *alu3* by applying the code in Figure 13.

```

alu3.Left.Merge(alu1.Left, alu2.Left);
alu3.Right.Merge(alu2.Right, alu2.Right);

alu3.InUse = alu1.InUse || alu2.InUse;
alu3.F3 = alu1.InUse && alu1.F3 || alu2.InUse && alu2.F3;
...
alu3.F0 = alu1.InUse && alu1.F0 || alu2.InUse && alu2.F0;

```

Figure 13: Compiler's code to merge two ALUs. The function *Merge* adds all inputs to *alu3*'s input multiplexer and ORs the control of any common inputs.

4.3 Data Dependent Dynamic Control

Some operations, such as maximum and absolute value, require data-dependent dynamic control to be generated. For example, the statement `sum += |x-y|` compiles to a netlist which connects the `sign` status signal of an ALU computing $x-y$ to the add/subtract control input of a second ALU. If the sign is positive, the second ALU adds the first result to sum, and if negative it subtracts the first result from sum. In more complex data dependent conditions, decoding may be required as shown in the next section.

4.4 Instruction and Decoder Generation

Each dynamic control signal is represented by a boolean function of the following variable types: an event in the control tree, a status bit from the datapath, and a condition on *s*. Examples of such

boolean variables include the first iteration of a For node (`i.first`), the carry condition on an ALU (`alu.carry`), and the equivalence of a For node and `s` (`i==s`), respectively. Each control signal is paired with an in-use function to aid optimization. Given this dynamic control information, a set of boolean functions, whose concatenation forms an *instruction*, must be found from which all dynamic signals can be decoded. This set of functions is limited by the instruction width of the target architecture.

Mapping all dynamic control into a fixed-width instruction involves finding common subexpressions within the dynamic control signals and using in-use information efficiently. In addition, this process may require multi-level minimization, Shannon decomposition and/or compilation to state machines, depending on the complexity of the dynamic control functions.

A dynamic signal comprised entirely of events from the control tree is independent of `s` and can be broadcast to all required stages. For example, the function `i.live && j.first` compiles directly to a bit in the instruction, which is then broadcast to each required functional unit and multiplexer. Since these variables are independent of `s`, they can be computed outside of the array by an external microcontroller.

A dynamic signal whose function is dependent on `s` may require decoding. This might be in the form of a state machine or a simple pipeline. For example, consider a RaPiD-C program containing a For node `i` and a dynamic control function `i==s`. Because the variable `i==s` is dependent on `s`, it can't be directly generated outside the array. However, if `i` has an increment of one and a range which includes zero to the number of stages, this dynamic control signal can be compiled into a singly pipelined control line driven by the boolean function `i==0`, which is independent of `s`. Similarly, a conditional of the form `i==ks` can be realized by creating a control pipeline with `k` registers per stage.

A dynamic control signal that is a function of more than one variable often requires special decoding. For example, the statement `if (i.first && X==Y) FOO = Z;` requires local decoding since the condition `X==Y` is compiled to the "is result zero?" output of an ALU subtract. The binary AND of this signal and `i.first` must be formed in the stage associated with the code, as is shown in Figure 14.

After generating the control path complete with decoding, the final step is to produce microprogram code that will generate the instruction stream. Since the set of boolean functions comprising the instruction consists entirely of boolean variables from the control tree, the microprogram is similar to the control tree itself. Each parallel task of the tree is mapped to an instruction generator, as shown in Figure 3. Hence, the number of instruction generators places a limit on the number of simultaneous parallel tasks in a RaPiD-C program.

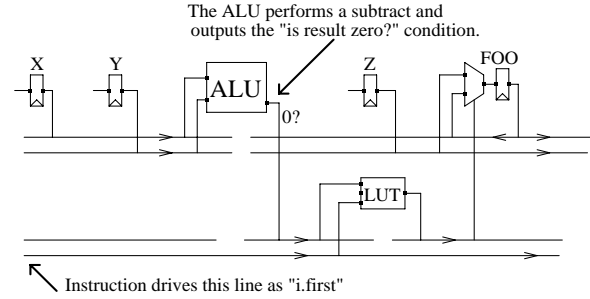


Figure 14: The dynamic condition `i.first && x==y` is generated by configuring a LUT as an AND gate.

An instruction bit depending only on variables from a single parallel task is generated by a single instruction generator. An instruction bit depending on variables from more than one parallel task must be decomposed (using two-level or multi-level minimization) into functions specific to a single parallel task. These functions are computed on their corresponding instruction generators and then later recombined in the merge unit to form the original instruction bit.

4.5 I/O Address Generation

The input/output addresses are generated by a set of address generators, such as shown in Figure 4. Each array reference is extracted from the RaPiD-C specification and dedicated to an address generator.⁴ The memory controller processes these requests in parallel with (and potentially ahead of) the computation on the RaPiD array.

For example, pseudo-code for generating the addresses for matrix `B` of matrix multiply is shown in Figure 15, where `B` represents the base offset of the array in memory. Matrix multiply requires three address generators, one for each array used.

```

for (t=0; t < L; t+=R)
  for (g=0; g < M; g+=R)
    for (h=0; h < N; h+=S)
      for (j=0; j < R; j++)
        for (s=0; s < S; s++)
          // Output the address associated with B[j+g][s+h]
          Output(B + (j+g)*N + (s+h));

```

Figure 15: Matrix `B` address generation

4.6 Pipelining and Retiming

Although the target architecture's functional units and memories may be pipelined, the programmer can assume that the units are combinational to simplify the code. In addition, the programmer can specify

⁴Although there is a limit on the number of address generators, two or more references could be mapped to the same address generator if they occur on different cycles.

data using a broadcast model even though such broadcasts might not meet the required cycle time. A retiming step ensures that the final netlist adheres to the target architecture's pipeline structure and timing requirements. A retimed circuit will adhere to the cycle time of the target hardware, taking into account delays through RaPiD elements, as well as pipelining requirements present in the underlying architecture. Because placement cannot be done until retiming is performed, the retimer conservatively estimates routing delays between elements.

5 Future Work and Conclusions

There are several ways we plan to extend the capability of the RaPiD-C language and compiler to make them more powerful. Some are simple extensions to the compiler to generate more optimized datapaths. These extensions would rely on extracting more information from the control structure to allow better sharing of resources and a more efficient generation of control. Other extensions are more far-reaching such as incorporating automatic time-multiplexing. Currently the programmer must explicitly describe how the time-multiplexing is done, which can be complicated and error-prone. It would be better to present the programmer with an array of arbitrary length and map this to the physical array by automatically introducing time-multiplexing. Another extension would be to have the compiler infer data movement from a description of the computation. That is, the specification would indicate the operations and data items and the compiler would create the dataflow required to satisfy the computation.

One of the disadvantages to many configurable architectures is the difficulty of specifying and compiling the computation. In this paper we have presented a conceptually clean and effective way to specify a pipelined implementation for regular and repetitive computation. This language requires the programmer to map the computation to space and time, but provides simple and concise ways to do this. The compiler is then able to generate the appropriate configuration data and dynamic control structure to implement the computation in a RaPiD array.

The RaPiD-C language can be viewed either as a convenient, sufficiently high-level language for programmers to describe pipelined computations in, or as an intermediate language used by a parallel compiler to describe the space-time mapping derived from an even higher-level description of the computation. Such a compiler currently appears out of reach for many complex computations, but as research in parallelizing compilers progresses, we may reach the point where RaPiD-C is largely used only by the compiler back-end. Until then, it provides a relatively powerful and convenient way for programmers to program the RaPiD architecture.

Acknowledgments

We would like to thank Larry McMurchie, Chris Fisher, and Miguel Figueroa for their contributions to the RaPiD project.

References

- [1] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD—reconfigurable pipelined datapath. In R. Hartenstein and M. Glesner, editors, *6th International Workshop on Field-Programmable Logic and Compilers*, Lecture Notes in Computer Science, pages 126–135. Springer-Verlag, September 1996.
- [2] C. Ebeling, D. C. Cronquist, P. Franklin, and S. Berg. Mapping applications to the rapid configurable architecture. In *Field-Programmable Custom Computing Machines (FCCM-97)*, 1997.
- [3] P. Lee and Z. M. Kedem. On high-speed computing with a programmable linear array. In *Proceedings. Supercomputing '88*, pages 425–32. IEEE Comput. Soc. Press, 1988.
- [4] D. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-35(1):1–12, 1986.
- [5] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991.