

The Chimaera Reconfigurable Functional Unit

Scott Hauck, Thomas W. Fry, Matthew M. Hosler, Jeffrey P. Kao

Department of Electrical and Computer Engineering

Northwestern University

Evanston, IL 60208-3118 USA

{hauck, zaphod, mhosler, theta}@ece.nwu.edu

Abstract

By strictly separating reconfigurable logic from their host processor, current custom computing systems suffer from a significant communication bottleneck. In this paper we describe Chimaera, a system that overcomes this bottleneck by integrating reconfigurable logic into the host processor itself. With direct access to the host processor's register file, the system enables the creation of multi-operand instruction and a speculative execution model key to high performance, general-purpose reconfigurable computing. It also supports multi-output functions, and utilizes partial run-time reconfiguration to reduce reconfiguration time. Combined, this system can provide speedups of a factor of two or more for general-purpose computing, and speedups of 160 or more are possible for hand-mapped applications.

Introduction

By adapting to computations not well served by current processor architectures, reconfigurable systems have provided significant performance improvements. These adaptive computing systems develop custom logic implementations of computation kernels, accelerating processing. However, purely FPGA-based systems are usually unsuitable for complete algorithm implementation. In most computations there is a large amount of code that is executed relatively rarely, and attempting to map all of these functions into reprogrammable logic would be very logic-inefficient. Also, reconfigurable logic is much slower than a processor's built-in functional units for standard computations such as floating point and complex integer arithmetic, variable length shifts, and others. The solution to this dilemma is to combine the advantages of both microprocessor and FPGA resources into a single system. The microprocessor is used to support the bulk of the functionality required to implement an algorithm, while the reconfigurable logic is used to accelerate only the most critical computation kernels of the program.

Most current mixed processor-FPGA systems suffer from a communication bottleneck between the processor and the reconfigurable logic [Hauck95]. By placing the

reconfigurable logic in a separate chip from the processor, the limited off-chip bandwidth and added delay interfere in efficient FPGA-processor communication. The resulting overhead requires that large chunks of the application code must be mapped to the reconfigurable logic to achieve any performance benefits at all. This means that relatively few applications can benefit from current adaptive systems, and they must be hand-mapped in order to achieve high enough performance benefits to justify the hardware costs and extra complexities. All of these factors keep reconfigurable computing from entering the mainstream, and drive up the cost and complexity of these systems.

There has been initial work done on integrating processors and reconfigurable logic [French93, Albaharna94, DeHon94, Razdan94a, Razdan94b, Albaharna96, Rajamani96, Wirthlin95, Wittig96]. However, these systems in general use standard FPGA architectures, architectures that have not been designed to effectively support the needs of integrated FPGA-processor systems. Also, as we will show, there are significant opportunities for optimizing these systems by taking advantage of the tight coupling of processor and reconfigurable logic. In this paper we describe Chimaera, a hardware system consisting of a microprocessor with an integrated reconfigurable functional unit being developed at Northwestern University.

The Chimaera Execution Model

The primary strength of a reconfigurable coprocessor (RCP) or functional unit (RFU) is the ability to customize the hardware to a specific program's requirements. Thus, when a communications program is active the reconfigurable logic might contain data compression and decompression routines, while when a rendering package is running the reconfigurable logic would be switched to support graphics operations. A more complex application, such as a complete word processing application, might have different mappings to the reconfigurable logic for different sections of the code, with text search routines active in one phase of the code's operation, and postscript acceleration routines for another. While these operations may not provide as big a performance improvement as custom hardware due to the inevitable overheads inherent in reconfigurable logic, by being able to accelerate most or all

applications running on a system they provide performance gains for a much larger class of problems.

In order to efficiently support these demands, the Chimaera system treats the reconfigurable logic not as a fixed resource, but instead as a cache for RFU instructions. Those instructions that have recently been executed, or that we can otherwise predict might be needed soon, are kept in the reconfigurable logic. If another instruction is required, it is brought into the RFU, overwriting one or more of the currently loaded instructions. In this way, the system uses partial run-time reconfiguration techniques to manage the reconfigurable logic. This does require that the reconfigurable logic be somewhat symmetric, so that a given instruction can be placed into the RFU wherever there is available logic. Also, some FPGAs have forbidden configurations (such as multiple active drivers to the same shared routing resource) which can mean that intermediate states accidentally reached during reconfiguration can destroy the chip. As described later in this paper, the Chimaera system deals with this by using an architecture with no forbidden states, employing hardware support to avoid these problems. This also has the desirable side-effect that a faulty configuration generated by the run-time system will not destroy the processor.

In order to use instructions in the RFU, the application code includes calls to the RFU, and the corresponding RFU mappings are contained in the instruction segment of that application. The RFU calls are made by special instructions which tell the processor to execute an RFU instruction. As part of this RFU call an instruction ID is specified which determines which specific instruction should be executed. If that instruction is present in the RFU, the result of that instruction is written to the destination register (also contained in the RFU call) during the instruction's writeback cycle. In this way, the RFU calls act just like any other instruction, fitting into the processor's standard execution pipeline. If the requested instruction is not currently loaded into the RFU, the host processor is stalled while the RFU fetches the instruction from memory and properly reconfigures itself. Note that this reconfiguration time can be quite significant. Thus, care must be taken to avoid constant reloading of the RFU. We are currently investigating techniques such as prefetching, caching algorithms, and caching hierarchies to avoid or reduce these reconfiguration penalties.

Normal instructions in the host processor not only specify the instruction to be performed and the destination for the result, but they also specify up to two source registers for the operands of the instruction. We could use a similar scheme for the RFU instructions as well. However, this would mean that the RFU would have exactly one cycle (the instruction's execute cycle) to compute its function,

since the operands are fetched in the previous cycle, and written back to the registers in the next cycle. Also, this would limit the RFU to having only two source operands, limiting the complexity of the computations. In Chimaera, we have chosen another approach. As shown in Figure 1, the reconfigurable logic is given direct read access to a subset of the registers in the processor (either by adding read connections to the host's register file, or by creating a shadow register file which contains copies of those registers' values). An RFU configuration itself determines from which registers it reads its operands. A single RFU instruction can read from all of the registers connected to the RFU, allowing a single RFU instruction to use up to nine different operands. Thus, the RFU call consists of only the RFUOP opcode, indicating that an RFU instruction is being called, an ID operand specifying which instruction to call, and the destination register operand. Just as importantly, an RFU instruction currently loaded into the RFU does not have to wait for the occurrence of an RFU call in the instruction stream to begin executing, since it already knows which registers it needs to access. In fact, all loaded RFU instructions "speculatively" execute during every processor cycle, though their results are only written back to the register file when their corresponding RFU call is actually made. This means that an RFU instruction can in fact use multiple cycles to execute without stalling the host processor. For example, assume that RFU instruction #12 uses the values in register R0...R3, and these values are computed in the four previous cycles. The instruction stream for this situation might look like the following:

```
R0 = R8 - R9; R1 = R10 * 2; LOAD R3; LOAD R4; R16 = RFUOP #12;
```

In this example, while the RFU instruction might only have one cycle (its normal execute cycle) to use the value from register R4, it will have at least four cycles to use the value from R0, three cycles to use the value from R1, and two cycles to use the value from R3. As long as late-arriving operands are not needed until near the end of an RFU computation, much more complex operations can be done inside the RFU than are possible in a single clock cycle. Thus, with careful RFU mapping creation and register assignment, and the application of code motion techniques, very complicated computations can be performed.

The Chimaera Architecture

The overall Chimaera architecture is shown in Figure 1. The main component of the system is the Reconfigurable Array, which consists of FPGA-like logic designed to support high-performance computations. It is here that all RFU instructions will actually be executed. This array gets its inputs directly from the host processor's register file, or a shadow register file which duplicates a subset of the

values in the host's register file. Next to the array is a set of Content Addressable Memory locations, one per row in the Reconfigurable Array, which determine which of the loaded instructions are completed. The CAMs look at the next instruction in the instruction stream and determine if the instruction is an RFUOP, and if so whether it is currently loaded. If the value in the CAM matches the RFUOP ID, the value from that row in the Reconfigurable Array is written onto the result bus, and thus sent back to the register file. If the instruction corresponding to the RFUOP ID is not present, the Caching/Prefetch control logic stalls the processor, and loads the proper RFU instruction from memory into the Reconfigurable Array. The caching logic also determines which parts of the Reconfigurable Array are overwritten by the instruction being loaded, and attempts to retain those RFU instructions most likely to be needed in the near future. Reconfiguration is done on a per-row basis, with one or more rows making up a given RFU instruction.

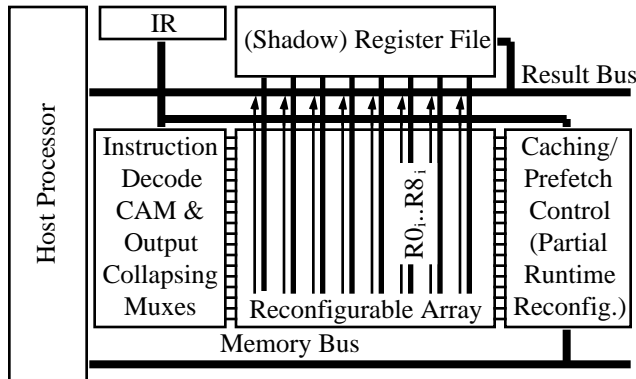


Figure 1. The overall Chimaera architecture.

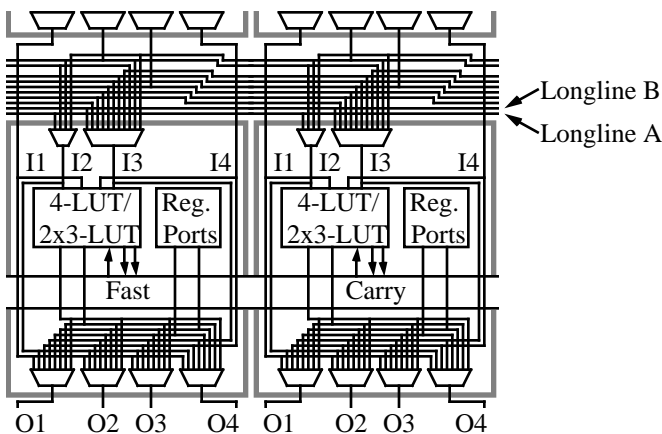


Figure 2. The Chimaera Reconfigurable Array routing structure.

The Reconfigurable Array itself is shown in Figure 2 and Figure 3. This architecture has been inspired by the Triptych FPGA [Hauck92, Borriello95, Ebeling95], the

Altera FLEX 8000 series [Altera95], and PRISC [Razdan94a, Razdan94b]. The routing structure is shown in Figure 2. The reconfigurable logic is broken into rows of logic cells between routing channels. Within that row, there is one cell per bit in the processor's memory word, so for a 32-bit processor there are 32 cells per row. All cells in a given column I have access to the I th bit of registers R0-R8, allowing it to access any two of these bits. Thus, a cell in the rightmost (0th) column in the reconfigurable array can read any two least significant bits from registers R0 through R8. Which register(s) a cell accesses is determined by its configuration, and different cells within the array can choose which registers to access independently.

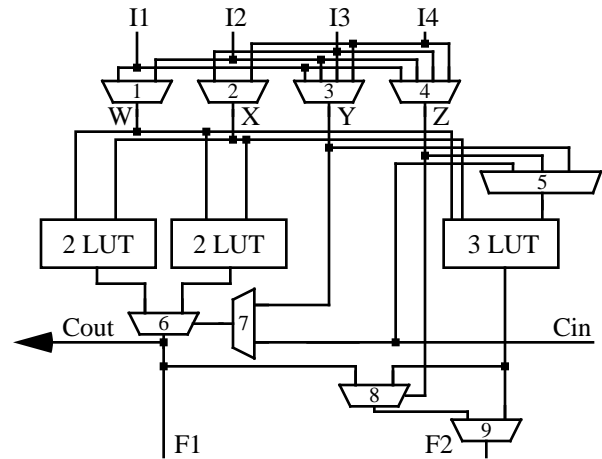


Figure 3. The Chimaera Reconfigurable Array logic block.

The cells of the array send four outputs O1..O4 and receive four inputs I1..I4 from the rest of the array. Inputs I1 and I4 come from the cell directly above, yielding high-speed connections to support regular datapath structures. Most computations will tend to involve bits from the same position in a data word, and thus will make heavy use of these direct connects. Inputs I2 and I3 can come from further away in the array (although they also draw exclusively from the outputs of the cells in the row above them). Input I2 for a cell in column C can choose from the O2 outputs from the cells in column C , $C+1$, or $C-1$ in the row above it, or from longline A. Input I3 can read from the same O2 outputs as I2, as well as the O3 outputs of cells within 3 of this cell ($C-3$ through $C+3$) and longline B. The longlines span the width of the array, with longline A connected to any one of the O2 outputs from the row above, and longline B connect to any one of the O3 outputs. This structure allows for the efficient communication of values locally within the array, as well as the global communication of any two values through the longlines. Each of the outputs of a cell are independently chosen from

any of its four inputs, the two outputs from the function block, and the two values read from the registers.

Chimaera's logic block is shown in Figure 3. The logic block takes the four inputs to the cell and shuffles them (via muxes 1-4) into intermediate signals W, X, Y, and Z. Since I1 and I4 can be interchanged in the routing structure without conflict, and W and X can be interchanged in the logic block without changing the possible functionality, we can use 2:1 muxes for W and X and still have complete permutability of the inputs. The logic block itself can be configured as a 4-LUT, two 3-LUTs, or a 3-LUT and a carry computation. This is done by realizing that a 2:1 mux controlled by an internal signal (not a programming bit) choosing between two N-LUT outputs, where those N inputs are identical, creates a (N+1) LUT. Thus, although mux 6 looks like it just chooses between two values, it actually forms a 3-LUT with the two 2-LUTs generating its input. Thus, to configure the cell as two 3-LUTs, Y is routed through mux 7, Z is routed through mux 5, and the output of the 3-LUT is sent through 9, making $F1 = 3LUT(W,X,Y)$ and $F2 = 3LUT(W,X,Z)$. A 4-LUT is created by sending Y through muxes 5 and 7, and the output of mux 8 through mux 9. Thus, mux 6 is still part of a 3-LUT, and mux 8 becomes the end of the 4-LUT, making $F2 = 4LUT(W,X,Y,Z)$. To perform a carry computation, Cin (from the cell one to the right in the same row) is sent through muxes 5 and 7, and the output of the 3-LUT through mux 9. Thus, $Cout$ and $F1 = 3LUT(W,X,Cin)$, and $F2 = 3LUT(W,X,Cin)$, with the left 3-LUT configured to compute the Carry value for this bit position, and the right 3-LUT generating the Sum. By using this carry configuration arithmetic and logical operations such as addition, subtraction, comparison, parity and others can be supported very efficiently. Note that Figure 3 represents the functionality possible in Chimaera's logic block, but not the actual implementation. Specifically, Chimaera's carry chain is not a simple ripple-carry, but instead uses an innovative high-performance carry chain structure that can provide the same functionality with more than an order of magnitude less delay [Hauck97].

There are some unusual aspects of the Chimaera architecture designed to allow it to efficiently provide custom instructions for its host processor. First, there are no state-holding elements in the Reconfigurable Array. Most FPGAs have Flip Flops or Latches in their logic block in order to implement sequential logic. However, such elements would require special consideration during context switches and during the loading of new instructions, since this state would need to be properly maintained over time. Also, these stateholding elements would complicate the speculative execution model of the system, since not

only must there be support to write the proper result back to the register file, it would also need to have control over when the stateholding elements are overwritten. Instead, we use the register file of the host processor as the only storage elements in the system, and allow the standard context switch mechanisms to handle all storage management issues. Sequential computations can still be implemented, with the result of one RFU instruction becoming the input to a subsequent RFU instruction by storing the value in a register accessible by the Reconfigurable Array.

Not only does the Chimaera RFU not have internal stateholding elements to implement sequential logic, it also lacks pipelining latches. This means that the registers a mapping accesses must remain at their proper value until the instruction is completed. An alternative to this would be to insert latches into the signal flow, allowing an input register to change before the instruction executes as long as the value was stable while it was being accessed. However, because of context switches due to multiprogramming, as well as stalls in the host processor, this turns out to be impractical. Specifically, imagine that we have an RFU instruction that reads the value of register R0 four cycles before it completes (i.e. there are 4 sets of pipeline latches between the register access and the output), and the instruction stream stores a new value into R0 two cycles before the RFU instruction is called. During normal operation, the RFU sees the old value stored in R0 (the value it is designed to use) and the result is properly computed. However, there may be a multiple cycle stall or context switch between the storing of the new value into R0 and the calling of the RFU instruction. This means that the proper value is no longer available in R0, and the RFU instruction will compute the wrong value. Since this means that we must always require that a register remain stable between the time an RFU instruction reads that register and it completes, pipelining latches become unnecessary, and are not present in our architecture.

Another interesting aspect of this architecture is the strictly downward flow of information and computation through the array. There is no way to send signals back to a higher row in the system. This structure mimics both the linear sequence of instructions found in an instruction stream, as well as the unidirectional flow of information found in strictly combinational logic. Inputs are accessed at any level in the computation, with early processing occurring near the top of an instruction, and results being produced at the bottom. Signals that travel across several rows must route through unused inputs and outputs in the intervening cells.

The routing structure has also been designed to efficiently support partial run-time reconfiguration. Instead of

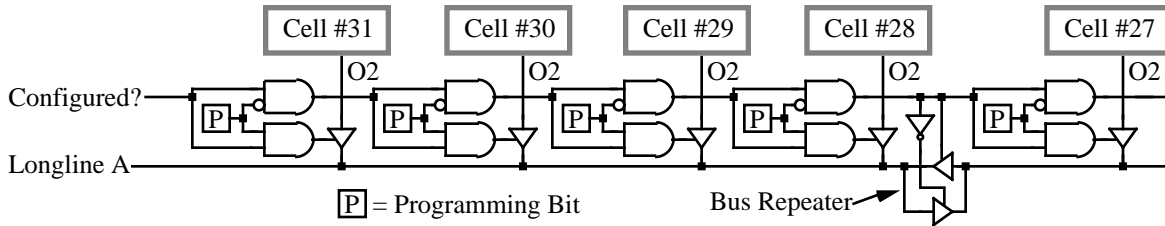


Figure 4. Control logic for the longlines, including logic to avoid multiple writer conflicts and a repeater.

requiring that every time a new instruction needs to be loaded into the RFU the entire reconfigurable array must be reconfigured, we will instead only change the contiguous set of rows required to hold the new instruction(s). In a normal FPGA, there are some configurations (such as multiple active writers to a single shared routing resource) which can destroy the FPGA with excessive current flows, and thus must be avoided. Avoiding these configurations during run-time reconfiguration is difficult, and may require that the portion of the FPGA be overwritten by a default “safe” configuration before the new configuration is loaded. This slows down reconfiguration, and means that a corrupted configuration could destroy the system. In Chimaera’s Reconfigurable Array only the longlines have more than one possible driver. All others are multiplexer based, meaning that regardless of the state of the programming bits there will be only one active driver. The longlines require a different solution (see Figure 4). Along with each longline is a control signal which travels from left to right. During normal operation, the “Configured?” input is true. The value passed from a cell to its neighbor on the right is true so long as none of the drivers to the recipient’s left are active. Once such a bit is found, that bus writer is enable. The control line from this cell is false, ensuring that no other bus writer will be turned on. Thus, even if the configuration bits are set to turn on more than one writer, all but the leftmost will be disabled. During configuration of this row, the “Configured?” signal is set to false, ensuring that none of the longline drivers will be enabled. The control signal for disabling drivers is also useful for controlling bus repeaters. Since the longlines span the width of the reconfigurable array, the capacitance of this line would either greatly slow down signal propagation, or require unreasonably large drivers. We can instead insert repeaters into the system, breaking the longline into shorter segments and boosting signal drive.

As mentioned earlier, the Chimaera RFU supports partial run-time reconfiguration on a per-row basis. Specifically, when a new instruction is loaded it will overwrite one or more rows of the system. While not all rows need be changed, if an instruction wishes to use any portion of a row it must use that entire row. What constitutes a “row” is

an important consideration. In Chimaera, all primary inputs to an RFU instruction come from the register access ports. Also, the result of an instruction comes from the F2 output of the function block. Thus, the natural breakpoint between one RFU instruction and the next is in the middle of a cell. A “row” for reconfiguration purposes consists of the register access ports and output muxes of one row of cells, the input muxes and logic blocks of the row of cells below it, and the routing channel between these cells.

The last portion of Chimaera that needs to be described is the RFU’s decode unit. As part of the host processor’s decode logic it determines if the current instruction is the RFUOP opcode, and if so it tells the RFU to produce the next result. The RFU must now decide if the requested instruction is currently loaded. This is done by associating a content-addressable memory cell with each row in the Reconfigurable Array, with its value specified by an RFU instruction’s configuration data. This cell contains the ID of the instruction computed in that row, and is checked against the RFU instruction ID contained as one of the operands in the RFU call. Rows that are not configured, or which are in the middle of a multi-row instruction, are set to a default value which can never match an RFU call. If the value contained in that CAM cell matches the value in the RFU call, the value computed in that row is sent onto the result bus and written into the proper register. The value written is the F2 output of the function blocks in that row, with the *I*th cell producing the *I*th bit of the result. If no CAM cell matches the RFU call, the configuration management unit first loads the instruction from memory and then executes it. Note that this organization allows for multi-output mappings. If a single mapping needs to produce multiple values, each of these values is generated at the output of a different row, and each of these rows is given a different RFU instruction ID. Although execution of this multi-output function requires multiple cycles, since each output will require a separate RFU call to write it back to the register file, it does allow these outputs to share logic in the RFU.

A slight modification to this decode scheme has been added to improve mapping density. In many cases, a logical test

<u>Line</u>	<u>Source Code</u>	<u>Standard</u>	<u>RFU-based</u>
1:	disp = hsize_reg - i;	bne v0, zero, Probe	RFUOP a0, 1
2:	if (i == 0)	subu a0, s1, v0	Probe:
3:	disp = 1;	li a0, 1	RFUOP t0, 2
4:	Probe:	Probe:	lw v1, 0(t0)
5:	if ((i - disp) < 0)	subu v0, v0, a0	RFUOP v0, 3
6:	i += hsize_reg;	bgez v0, NEXT	
7:	temp = htabof[i];	sll t9, v0, 2	
8:		addu v0, v0, s1	
9:		sll t9, v0, 2	
10:		NEXT:	
11:		addu t0, s0, t9	
12:		lw v1, 0(t0)	
13:		noop	

Figure 5. Source code (left) of a portion of the compress benchmark, the implementation produced by a standard C compiler (center), and a version using the Chimaera RFU (right).

Row	CAM	Flag	Computation
1	1	Izero	read i; Izero = (i == 0); output 1
2	1	!Izero	read hsize_reg; output hsize_reg - i
3	3	pos	read i, disp; pos = (i - disp) >= 0; output v1 = i - disp
4	2	pos	read htabof, hsize_reg; output (v1 << 2) + htabof
5	3	!pos	read i, disp; output v2 = (i - disp) + hsize_reg
6	2	!pos	read htabof; output (v2 << 2) + htabof

Figure 6. RFU contents for the Compress example. Note that instruction 1 (the first two rows) is separate from the rest, and does not have to be placed adjacent to the other instructions, but instructions 2 and 3 (rows 3-6) share logic, and thus must be placed contiguously.

will determine which of a set of values will be assigned to a register. For example, consider the code segment:

A = B + C; if (D == E) A = A + F;

In the RFU structure as described so far, this sequence of instructions would require 4 rows: one to test if D and E are equal, one to compute B + C, one for adding F to that value, and a final row to choose between the values (B+C) and ((B+C)+F) based on the value of the test. We can do better than this. In addition to checking whether the value of the CAM matches the RFU instruction ID in the RFU call, it also checks the value of the F1 signal in cell #31 of that row. If the CAM value matches the instruction ID and the F1 signal is true, the row produces the result. Otherwise, this row doesn't match the RFU call. We can use this logic to remove the fourth row from the mapping just proposed. Instead of muxing together the two potential output values, we instead assign the same RFU instruction ID to both row two (B+C) and three ((B+C)+F). To choose between them, we configure the leftmost cell in each row to output the value of the test done in the first row onto its F1 signal, with the second row outputting true if the test is false, and the third row outputting true if the test is true. Thus, the

addition of this small extra logic in the instruction decode CAMs allows the muxing together of values often required in computations. Note that since the F1 signal is generated by a 3-LUT, and uses signals from inside the reconfigurable array, very complicated multiplexing can be accomplished, with multiple rows assigned the same RFU instruction ID and computing a possible output. There are also provisions for disabling this logic by forcing the signal to the CAM to true in cases where the F1 3-LUT in cell #31 is needed for other logic.

Application Examples

In this section we give some examples of using the Chimaera RFU to accelerate standard software algorithms. Since we do not yet have an automatic mapping system to generate Chimaera implementations, we have mapped critical portions of some standard algorithms by hand to our architecture. Note that this does restrict the achievable speedups, since we have only optimized one or two short code sequences for each algorithm. A production version should be able to find many such opportunities in a single program, and achieve higher performance gains.

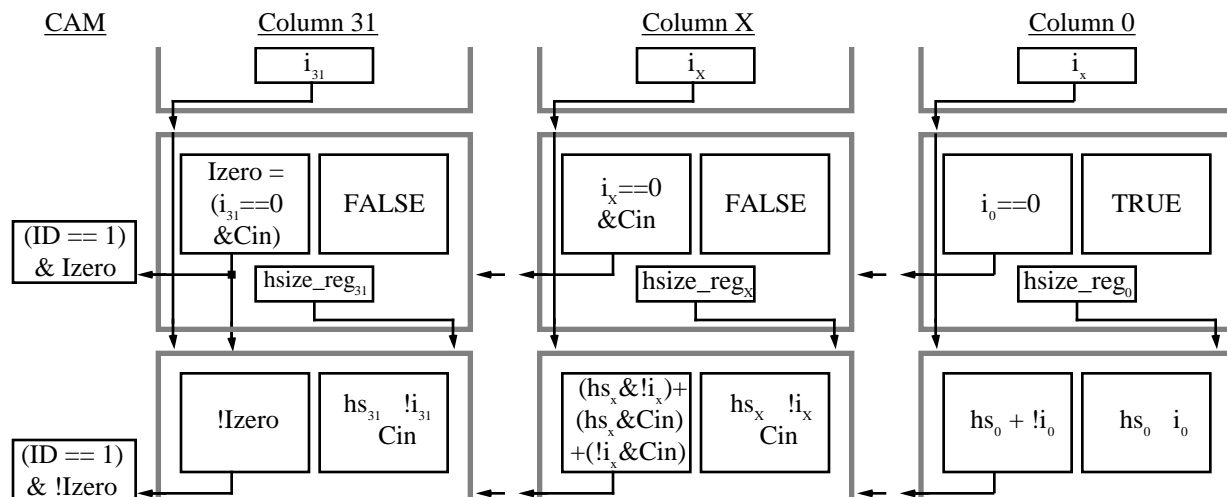


Figure 7. Detailed placement and routing of RFU instruction #1 for the Compress benchmark. Since all columns of the mapping are identical except for the two ends, column X represents the mapping for all columns $31 < X < 0$. The upper boxes in each cell are the two 3-LUTs, and the lower middle box is the register access ports. For simplicity the routing of inputs to the function blocks is not shown.

In order to test our results, we have taken the software programs and compiled them for a MIPS R4000 processor. These assembly language implementations were then optimized by hand, taking critical regions found by the performance evaluator Pixie and mapping them to the RFU. Note that since we are working with the MIPS instruction set, all branches and loads are followed by a single delay slot that must be filled. For simplicity we ignore pipeline stalls from cache misses.

In the examples we will use a textual shorthand to describe a mapping to a row in the RFU. A “read” operation is the accessing of a value from the register file, with all cells reading their bit of that register unless a bit position subscript is given. This read is performed at the top of that row (before the horizontal routing channel), and thus the values are available for the logic blocks to access. An “output” operation means that that value is computed in the cell’s logic block and sent to the F2 signal, where it is available to be written back to the register file. The “flag” signal is the signal sent from cell 31 to the Instruction Decode CAMs, where both the CAM value must match the instruction ID in the RFU call and the flag must be true in order to write this value back to the register file. This flag value normally comes from the left 3-LUT in the cell, though it can be forced to true via the configuration.

Compress

Compress is a member of the Spec92 benchmark suite. It consists of one main loop with a complex control flow, requiring multiple simple RFU instructions to provide a significant speedup. An example is shown in Figure 5

(simplified somewhat to aid understanding). RFU instruction #1 computes the value of the variable “disp” as specified in source code lines 1-3. In the standard computation this requires three assembly instructions: the initial subtract (line 2), the branch to decide if “i” is zero (line 1), and a load to set the value to 1 (line 3). Note that the branch has been moved before the subtract, since this architecture has a single branch delay slot. Performing this computation inside the RFU requires only two rows (Figure 6 and Figure 7). The first row decides whether “i” is zero via a carry-based computation, which results in the internal signal “Izero”. Since if “i” is zero we can immediately assign 1 to “disp”, we also fold this assignment into the same row. Izero is sent to the Instruction decode CAM, which will use this row’s output if Izero is true. The right 3-LUT in these cells are set to the full-word “1” value, the value needed to be assigned to “disp” if “i” is zero. The second row is used to handle the case when “i” is not zero. Here, we negate the “Izero” flag in cell #31 and send it to the decode CAM, since we want the result to come from this row only if it is not computed by the previous row. This row also computes the value of “hs_{reg} - i” via a carry-based subtraction, and writes it to each cell’s F2 signal, from which it is sent to the result bus.

The rest of the logic for this code sequence is handled by a two-output RFU instruction. In the source code, the value of “i” is computed, and then it is used as an array access. Thus, two values need to be computed: “i”, and the address of the memory location to be read. In the RFU mapping, RFU instruction 2 computes the memory access location, which is then loaded, and the writeback of the new “i” value is done in RFU instruction 3 in the load delay slot.

As shown in Figure 6, these two instructions occupy 4 rows in the RFU, and share a significant amount of logic. This mapping contains a constant-length shift of two bit positions. The Chimaera RFU is not able to efficiently support variable-length shifts (which would instead be performed on the host processor), but short constant-length shifts can be performed by the horizontal routing channels. Note also that we read some values from registers multiple times in order to reduce the depth of the logic (which also serves to minimize routing congestion). Different CAM ID values and Flags identify the different rows in the RFU, providing for free muxing of output values. By using just these three RFU operations we achieve a speedup of 1.11 over the standard software version, and many other such optimizations are possible.

```

1:   int cmppt (a, b)
2:   PTERM *a[], *b[];
3:   {
4:     register int i, aa, bb;
5:     for(i=0; i<ninputs; i++){
6:       aa = a[0]->ptand[i];
7:       bb = b[0]->ptand[i];
8:       if (aa == 2)
9:         aa = 0;
10:      if (bb == 2)
11:        bb = 0;
12:      if (aa != bb) {
13:        if (aa < bb)
14:          return (-1);
15:        else
16:          return (1);
17:      }
18:    }
19:    return (0);
20:  }

```

Figure 8. Code from the primary loop in Eqntott, part of the Spec benchmark suite.

Eqntott

Eqntott is a member of the Spec92 benchmark suite which

Row	CAM	Flag	Computation
1	null		read aa, bb; same = (aa==bb)
2	1	!equal	read ninputs; equal = (same ((aa==0 aa==2)&&(bb==0 bb==2))); output ninputs
3	1	equal	read i; output i+1
4	2	equal	read aa; aaistwo = (aa==2); output 0 /* Note: Cell 30 generates aaistwo in F1 */
5	null		read bb; newaa = if (aaistwo) 0 else aa; bbistwo = (bb==2)
6	2	aasmaller	aasmaller = ((newaa < bb) && !bbistwo); output -1
7	2	bbsmaller	bbsmaller = (!aasmaller && !equal); output 1

Figure 10. RFU instructions for Eqntott.

spends about 85% of its time in a single routine, “cmppt” (see Figure 8). The routine iterates through a pair of arrays, and does a complex comparison between the values in those arrays. In order to accelerate this algorithm via the RFU, we created two custom instructions (see Figure 9 and Figure 10). RFU instruction 1 is called in the middle of the loop, and determines whether the algorithm breaks out of the loop by generating the new value of variable “i”. To break out of the loop we set “i” to ninputs, while to remain in the loop we just increment it. Once we break out of the loop (or fall out once we have searched the entire length of the arrays), we call RFU instruction 2 to determine what return value to send. Most of the effort in these mappings is comparisons between two values, which can be done efficiently by the carry logic in the RFU. We in fact are able to perform two such tests in row 2 because of the specifics of the comparisons.

```

1: TOP:lh a0, 0(a1)
   /* a1 = &(a[0]->ptand[i]), a0 = aa */
2:   lh a2, 0(a3)
   /* a3 = &(b[0]->ptand[i]), a2 = bb */
3:   addiu a1, a1, 2
4:   RFUOP v0, 1 /* End loop? */
5:   bne v0, v1, TOP /* i!=ninputs? */
6:   addiu a3, a3, 2 /* Delay slot */
7:   jr ra /* Return */
8:   RFUOP v0, 2 /* Return val */

```

Figure 9. Code for lines 5-20 of Eqntott using RFU instructions #1 and #2.

In this mapping the two instructions share some logic. In fact, the computation of RFU instruction 2 actually spans seven rows, since row 7 indirectly uses the value computed in row 1. This computation obviously is too long to fit into a single clock cycle. However, as can be seen in Figure 9, all operands to RFU instruction 2 are available by line 2, and thus this instruction has 6 cycles in which to compute. Instruction 1 actually has 2 cycles to compute in as well, since an unrelated addition was moved in front of the RFU

Row	CAM	Flag	Computation
1	null		read pos ₄ , pos ₂ , pos ₁ , temp; t1 = filter(temp, !pos ₂ , !pos ₁)
2	null		read pos ₃ , pos ₀ ; t2 = filter(t1 _{16..31} , pos ₃ , pos ₀); t3 = filter(t1 _{0..15} , !pos ₃ , !pos ₀)
3	1	TRUE	v = (if (pos ₄) t3 else t2); Cells 31-1 output 0, Cell 0 outputs v;

Figure 12. RFU instructions for a basic implementation of Life. The filter function zeroes any bits of the first operand that are not at any of the bit positions with the corresponding indices (i.e. if pos₀ and pos₁ were passed in, and both were 0, any bit not in a bit position that is a multiple of four would be set to zero). The indices are negated to account for the subtraction in the source code.

Row	CAM	Flag	Computation
1	null		read NW, N, NE, SW, S, SE; C1,S1 = FullAdd(NE,N,NW); C2,S2 = FullAdd(SW,S,S,SE)
2	null		read E; V2,V1,V0 = 2BitAdd(C1S1, C2S2, Cin = E);
3	null		X2, X1 = (if (V2V1V0 > 4) 00 else V1, V0);
4	1	TRUE	read W, Self, OddVals; output Life(X2, X1, W, Self); Odd cells output OddVals;

Figure 13. RFU instructions for a highly parallel implementation of Life. Only a single even bit position’s calculation is shown. All 16 even bit position computations can be performed in parallel in a single RFU instruction, sharing the same rows in the RFU. A similar RFU instruction is needed to compute the odd bit positions.

call. The use of RFU instructions in this critical portion of the algorithm greatly simplifies the routine (which normally requires 23 instructions in place of our 8), and achieves a speedup of 1.8.

Life

The previous examples have been benchmark circuits where we have little control over the algorithm organization or understanding of the details of its operation. In order to test how algorithms can be altered to take advantage of the Chimaera RFU, we developed an implementation of Conway’s Game of Life [Gardner70]. This is a simple cellular automata with states “Live” and “Dead”, where a cell determines its next state based on its own state and that of its eight direct neighbors (including diagonals). If there are 3 live neighbors, the cell becomes alive. If there are 2 live neighbors, the cell retains its current state. Otherwise, the cell becomes dead. We wrote a software version of this algorithm, representing each cell as a single bit, allowing us to store a 128x128 board in an array of 4 by 128 integers.

```
int get_bit(int temp, int
position)
{
    temp >>= (31 - position);
    temp &= 0x00000001;
    return temp;
}
```

Figure 11. The get_bit routine from the Life algorithm. This routine retrieves the bit from variable temp at the specified position.

In the software version of the algorithm, more than half of the time is spent in the routines “get_bit” and “put_bit”, which read and write the value of individual cells. By simply replacing these routines with RFU instructions we can get a speedup of 2.06. The RFU instruction for “get_bit” is given in Figure 12.

We can do significantly better than this by realizing that although the processor itself does not have routines capable of performing more than one cell calculation at a time, the RFU can be configured to compute multiple cells at once. As shown in Figure 13, the computation for a cell’s next state can be accomplished in two columns of an RFU, with a mapping 4 rows high. Thus, by careful packing we can compute all of the odd bit positions in a single word in one RFU instruction, and compute the even bit positions in the next instruction. This does require the movement of values around in the registers, since every call to an RFU instruction expects that all the operands will be in specific registers. Some required values may thus only be available the cycle before the RFU instruction is called, and we therefore inserted a NOOP in front of some RFU calls to add extra computation time. The bit-parallel implementation of the Game of Life greatly reduces the size of the inner loop, achieving a speedup of about 160 times over a standard software implementation.

Conclusions

Current reconfigurable systems have been able to deliver huge speedups for some types of applications, but require a significant effort to hand-optimize the algorithms. This is

primarily due to the communication bottleneck between the reconfigurable logic and the host processor, which requires careful optimization, and the migration of a significant amount of computation to the reconfigurable logic, to overcome.

In order to extend the benefits of reconfigurable logic to general-purpose computing, we propose integrating the reconfigurable logic into the processor itself. The Chimaera system provides a host microprocessor with a reconfigurable functional unit for implementing custom instructions on a per-application basis. Direct read access to the processor's register file enables multi-input functions and a speculative execution model allowing for multi-cycle operations without pipeline stalls. A novel instruction decode structure provides for multi-output functions and efficient implementation of complex operations. Finally, by using partial run-time reconfiguration, we can view the RFU as an operation cache, retaining those instructions necessary for the current operations.

Through several hand-mappings to the RFU we have demonstrated the power of the Chimaera system. The Compress benchmark demonstrated a speedup of 1.11 after only limited optimization, Eqntott a speedup of 1.8, and basic Life a speedup of 2.06. These optimizations required only local optimization of a small amount of the source code, transformations that should be possible to achieve in an automatic mapping system. The more aggressive parallel optimization of the Life algorithm produced a speedup of about 160, demonstrating the potential for extremely high performance for some applications via careful, hand optimization.

References

- [Albaharna94] O. T. Albaharna, P. Y. K. Cheung, T. J. Clarke, "Area & Time Limitations of FPGA-based Virtual Hardware", *International Conference on Computer Design*, pp. 184-189, 1994.
- [Albaharna96] O. T. Albaharna, P. Y. K. Cheung, T. J. Clarke, "On the Viability of FPGA-based Integrated Coprocessors", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.
- [Altera95] *Data Book*, San Jose, CA: Altera Corp., 1995.
- [Borriello95] G. Borriello, C. Ebeling, S. Hauck, S. Burns, "The Triptych FPGA Architecture", *IEEE Transactions on VLSI Systems*, Vol. 3, No. 4, pp. 491-501, December, 1995.
- [DeHon94] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 31-39, 1994.
- [Ebeling95] C. Ebeling, L. McMurchie, S. Hauck, S. Burns, "Placement and Routing Tools for the Triptych FPGA", *IEEE Transactions on VLSI Systems*, Vol. 3, No. 4, pp. 473-482, December, 1995.
- [French93] P. C. French, R. W. Taylor, "A Self-Reconfiguring Processor", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 50-59, 1993.
- [Gardner70] M. Gardner, "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life'", *Scientific American*, pp. 120-123, October, 1970.
- [Hauck92] S. Hauck, G. Borriello and C. Ebeling, "TRIPTYCH: An FPGA Architecture with Integrated Logic and Routing", *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pp. 26-43, March, 1992.
- [Hauck95] S. Hauck, *Multi-FPGA Systems*, Ph.D. Thesis, University of Washington, Dept. of Computer Science & Engineering, 1995.
- [Hauck97] S. Hauck, M. Hosler, T. Fry, "High-Performance Carry Chains for Reconfigurable Computing", submitted to *IEEE Symposium on Custom Computing Machines*, 1997.
- [Rajamani96] S. Rajamani, P. Viswanath, "A Quantitative Analysis of Processor - Programmable Logic Interface", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.
- [Razdan94a] R. Razdan, *PRISC: Programmable Reduced Instruction Set Computers*, Ph.D. Thesis, Harvard University, Division of Applied Sciences, 1994.
- [Razdan94b] R. Razdan, M. D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", *International Symposium on Microarchitecture*, pp. 172-180, 1994.
- [Wirthlin95] M. J. Wirthlin, B. L. Hutchings, "A Dynamic Instruction Set Computer", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.
- [Wittig96] R. Wittig, P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.