The Chimaera Reconfigurable Functional Unit

Scott Hauck, *Senior Member, IEEE*, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao

***Abstract*—By strictly separating reconfigurable logic from the host processor, current custom computing systems suffer from a significant communication bottleneck. In this paper, we describe Chimaera, a system that overcomes the communication 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 instructions and a speculative execution model key to high-performance, general-purpose reconfigurable computing. Chimaera also supports multi-output functions and utilizes partial run-time reconfiguration to reduce reconfiguration time. Combined, the 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.**

***Index Terms*—Adaptive systems, field-programmable gate arrays (FPGAs), reconfigurable architectures.**

I. Introduction

RECONFIGURABLE systems have provided significant performance improvements by adapting to computations not well served with current processor architectures. Adaptive computing systems developed to date accelerate processing by offering custom logic implementations of computation kernels. However, purely field-programmable gate array (FPGA)-based systems are usually unsuitable for complete algorithm implementation. In most computations, there is a large amount of code which is executed relatively rarely. Attempting to map all of these functions into reprogrammable logic would be very logic inefficient. Also, reconfigurable logic is much slower than the 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.

Manuscript received December 11, 2002, revised July 29, 2003. This work was supported in part by a grant from the Defense Advanced Research Projects Agency and by a grant from the National Science Foundation (NSF). The work of S. Hauck was supported by a NSF CAREER Award and a Sloan Research Fellowship.

S. Hauck was with Northwestern University, Evanston, IL 60201 USA. He is now with the Department of Electrical Engineering, University of Washington, Seattle, WA 98195 USA (e-mail: hauck@ee.washington.edu).

T. W. Fry was with the University of Washington, Seattle, WA 98195 USA. He is now with IBM Microelectronics, Waltham, MA 02454 USA (e-mail: fry1@us.ibm.com).

M. M. Hosler was with Motorola Corporate Research Labs, Schaumburg, IL 60179 USA. He is now with Arrow Electronics, Dayton, IL 61350 USA (e-mail: mhosler@pobox.com).

J. P. Kao was with Intel Chandler, AZ 85224 USA. He is now with the University of Michigan Business School, Ann Arbor, MI 48103 USA (e-mail: jeffkao@umich.edu).

Digital Object Identifier 10.1109/TVLSI.2003.821545

Most current mixed processor-FPGA systems suffer from a communication bottleneck between the processor and the reconfigurable logic [11]. 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 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.

Initial work on integrating processors and reconfigurable logic has been done [1], [2], [5], [6], [8], [13], [18]–[23]. For example, the PRISC architecture allows the CPU to issue instructions to a reconfigurable array to optimize an application. However, these systems typically use standard FPGA architectures which were not designed specifically to optimize the needs of an integrated FPGA-processor system.

In this paper, we describe Chimaera, a hardware system consisting of a microprocessor with an integrated reconfigurable functional unit (RFU) designed small enough to fit onto the microprocessor itself. In order to coexist on a microprocessor's die, the Chimaera architecture was designed to be a stripped-down FPGA optimized for small function acceleration. By implementing a "light" FPGA array, the Chimaera architecture is designed to use relatively little chip real estate within a microprocessor's die and improve the overall cost/performance ratio of the system, rather than only optimize speed by including more functionality than required.

II. Chimaera Execution Model

The primary strength of a reconfigurable coprocessor (RCP) or functional unit is the ability to customize hardware for a specific program's requirements. When a communications program is active, the reconfigurable logic might contain data compression and decompression routines, and when a rendering package is running, the reconfigurable logic would be switched to support graphics operations. More complex applications, 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 large 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 by 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. Since the reconfigurable logic is somewhat symmetric, a given instruction may be placed into the RFU wherever there is space available. 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. The Chimaera system deals with these intermediate states by using an architecture with no forbidden states by employing hardware support to avoid these problems. As a desirable side effect, 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 instruction's result 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 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. The reconfiguration time can be quite significant. Thus, care must be taken to avoid constant reloading of the RFU. Techniques such as prefetching, caching algorithms, and caching hierarchies have been developed in order to avoid or reduce these reconfiguration penalties [16].

Normal instructions in the host processor specify not only 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. Since the operands are fetched in the previous cycle and written back to the registers in the next cycle, this would mean that the RFU would have exactly one cycle (the instruction's execute cycle) to compute its function. In addition, such a scheme would limit the RFU to having only two source operands, limiting the complexity of the computations.

In Chimaera, we have chosen another approach. 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). The RFU configuration itself determines from which registers it reads as 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

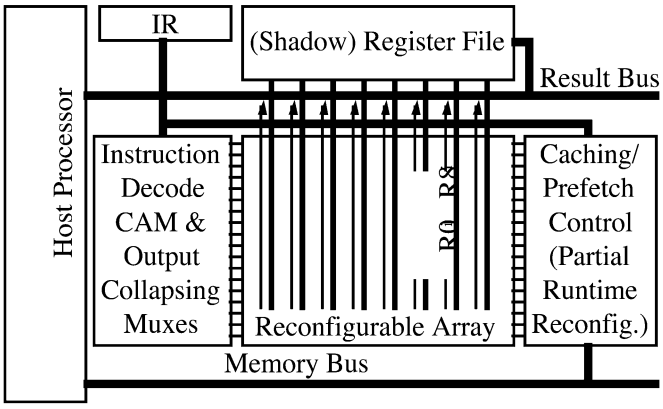


Fig. 1. Overall Chimaera architecture.

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. One result is that an RFU instruction may, in fact, use multiple cycles to execute without stalling the host processor. Also, internal pipelining is not necessary for a complex mapping that requires more than one clock cycle. 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:

$$\begin{aligned}&\mathrm{R0} = \mathrm{R8} - \mathrm{R9}\\&\mathrm{R1} = \mathrm{R10}^{*}2\\&\mathrm{LOAD\ R2}\\&\mathrm{LOAD\ R3}\\&\mathrm{R16} = \mathrm{RFUOP}\#12.\end{aligned}$$

In this example, while the RFU instruction might only have one cycle (its normal execute cycle) to use the value from register R3, 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 R2. As long as late-arriving operands are not needed until near the end of an RFU computation, more complex operations can be done inside the RFU than are possible in a single clock cycle. With careful RFU mapping, creation, and register assignment, and the application of code motion techniques, very complicated computations can be performed.

III. Chimaera Architecture

Fig. 1 shows the overall Chimaera architecture. The main component of the system is the reconfigurable array, which consists of FPGA-like logic designed to support high-performance computations. In the array, all RFU instructions will be executed. The array receives inputs from a shadow register file which duplicates a subset of the values in the host's register file.

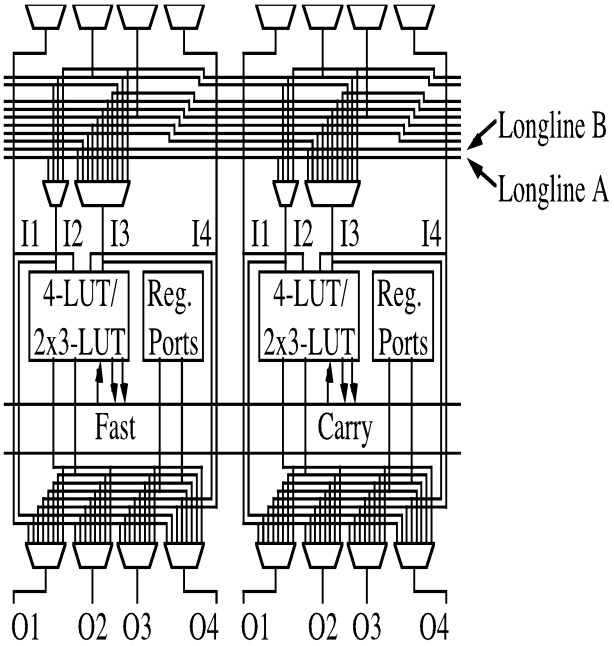


Fig. 2. Chimaera reconfigurable array routing structure (two cells within a row of the array).

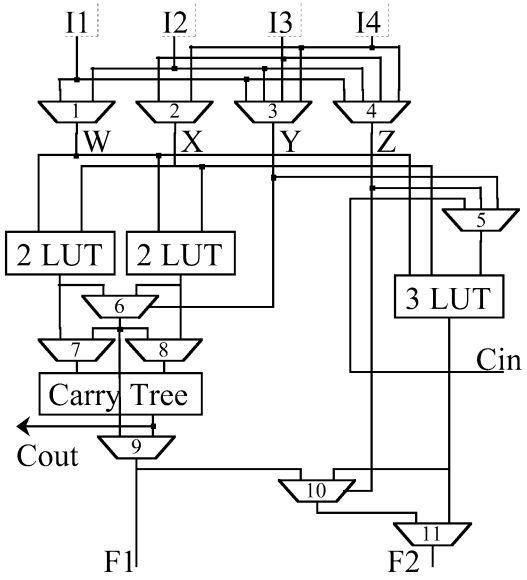


Fig. 3. Chimaera reconfigurable array logic block (the 4-LUT/2 × 3-LUT structure within Fig. 2).

Essentially, all changes to the target registers are broadcast to the array, so that RFU instructions can execute on the new value. Next to the array is a set of content-addressable memory (CAM) locations, one per row in the reconfigurable array, which determine which of the loaded instructions are completed. When an RFUOP is loaded into the array, one or more rows are designated as output rows by placing the RFUOP ID into the CAM line corresponding to that row. The CAM looks at the next instruction in the instruction stream and determines if the instruction is an RFUOP. If so, it checks whether the RFUOP 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 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.

The reconfigurable array is shown in Figs. 2 and 3. The Chimaera architecture was inspired by the Triptych FPGA [4], [7], [10], the Altera FLEX 8000 series [3], and PRISC [19], [20]. In addition, before creating the Chimaera architecture, candidate code segments were analyzed to help determine what routing and logic structures would be most beneficial in an RFU. To reduce the total size of the RFU array, only the most commonly required structures were included in the RFU.

The routing structure is shown in Fig. 2. The reconfigurable logic is broken into rows of logic cells between routing channels. Within each row, there is one cell per bit in the processor's memory word. For example, a 32-bit processor has 32 cells per row. All cells in a given column N have access to the Nth bit of registers R0–R8, allowing it to access any two of these bits. Thus, a cell in the rightmost (zeroth) 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 cells can independently choose which register(s) to access.

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 data-path structures. Most computations tend to involve bits from the same position in a data word and often make heavy use of these direct connections. 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 and the O3 outputs of cells within three 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 connected to any one of the O3 outputs. The routing structure allows for efficient communication of values locally within the array, as well as 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 Fig. 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. In addition, since I3 can read the same O2 outputs as I2, we still have complete permutability of the inputs. The logic block can be configured as a 4-LUT, two 3-LUTs, or a 3-LUT

and a carry computation. The first 3-LUT is created 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. 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. To configure the cell as two 3-LUTS, Y is routed to mux 6 and this result is passed through mux 9. Z is routed through mux 5 and the output of the 3-LUT is sent through 11, making $F1 = 3\mathrm{LUT}(W, X, Y)$ and $F2 = 3\mathrm{LUT}(W, X, Z)$. A 4-LUT is created by sending Y through mux 5, sending the result of mux 6 though mux 9, and sending the output of mux 10 through mux 11. Thus, mux 6 is still part of a 3-LUT and mux 10 becomes the end of the 4-LUT, making $F2 = 4\mathrm{LUT}(W, X, Y, Z)$.

To perform a carry computation, the output of the two 2-LUTs are passed through muxes 7 and 8 into the carry tree. The carry tree is a modified Brent Kung adder [12] which takes two inputs from each cell and produces one output (the value of Cout). The output of the carry tree is then routed from the previous cell through mux 5 and the output is passed through mux 11. Thus, $F1 = \mathrm{Cout}$ and $F2 = 3 - \mathrm{LUT}(W, X, \mathrm{Cin})$, with the left 3-LUT configured to compute the propagate and generate values for this bit position, and the right 3-LUT generating the sum. The modified Brent Kung adder also supports an inverse propagate and generate value for the purpose of checking parity, something which standard adders may not perform. By using the carry configuration arithmetic, logical operations such as addition, subtraction, comparison, parity, and others can be supported efficiently.

There are some unusual aspects of the Chimaera architecture design, allowing 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 state-holding elements would complicate the speculative execution model of the system. 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 state-holding 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.

In addition to the Chimaera RFU not having internal state-holding 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 and 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 four 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 reads 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. In such a situation, the proper value will no longer be available in R0, and the RFU instruction will compute the wrong value. As a result, we require that a register remain stable between the time an RFU instruction reads that register and the time it completes. In addition, under such a model, pipelining latches become unnecessary and so 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 been designed to efficiently support partial run-time reconfiguration. Instead of requiring that every time a new instruction needs to be loaded into the RFU the entire reconfigurable array must be reconfigured, we will 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 must be avoided. Avoiding these configurations during run-time reconfiguration is difficult. In some cases, it is required that the portion of the FPGA be overwritten by a default "safe" configuration before the new configuration is loaded. Such methods slow down reconfiguration time and mean that a corrupted configuration could destroy the system. In Chimaera's reconfigurable array, no writing elements have more than one possible driver. All 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 Fig. 4). Along each longline, a control signal 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 an active bit is found, that bus writer is enabled. 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

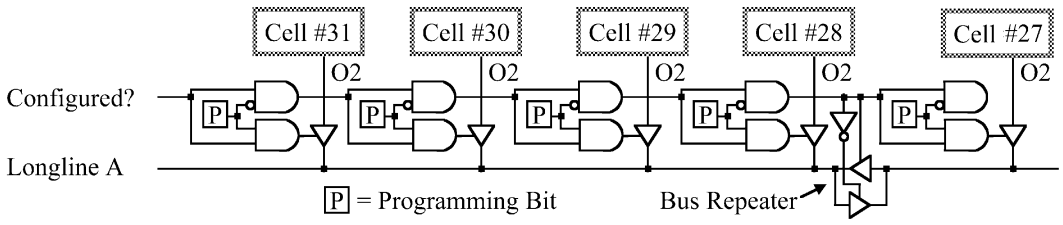


Fig. 4. Control logic for the longlines, including logic to avoid multiple writer conflicts and a repeater.

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. Such partial reconfiguration is possible since no vertical routing element extends beyond a single row. 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 which needs to be described is the RFU's decode unit. The host processor's decode logic will determine if the current instruction is the RFUOP opcode. If so, the decode unit 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 multirow 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 Ith cell producing the Ith 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. Each output will require a separate RFU call to write it back to the register file. In doing so, it allows 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 will determine which of a set of values will be assigned to a register. For example, consider the code segment

$$A = B + C;\ \text{if}\ (D == E)A = A + F.$$

In the RFU structure described so far, this sequence of instructions would require four rows: one to test if D and E are equal, one to compute $B + C$, one to add 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 such a selection. 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 does not 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 rows 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. Since the F1 signal is generated by a 3-LUT and uses signals from inside the reconfigurable array, 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.

IV. Application Examples

In this section, we give some examples of using the Chimaera RFU to accelerate standard software algorithms. An automated Chimaera compiler has been developed at Northwestern University (Evanston, IL), and those results are presented elsewhere [24]. Here we will focus on hand-selected and mapped examples to demonstrate the potential of this architecture. We have mapped critical portions of some standard algorithms by hand to our architecture. Mapping these examples by hand does somewhat restrict our achievable speedups. The reason is that typically we have to optimize only one or two short code sequences in the inner loop of the program in order to achieve acceptable speedups for each algorithm. The production version should be able to find many such opportunities in a single program and

```
Line    Source Code                     Standard                RFU-based
  1:    v = p1[i] + p1[i+1] + 1;        ldub [g4 + g0], g2      ldub [g4+g0], g2
  2:    v = v >> 1;                     ldub [g4 + 1 ], g3      ldub [g4+1],  g3
  3:    v = v - p2[i];                  add  g2, g3, g2         ldub [o7+i4], g4
  4:    if( v >= 0 )                    add  g2, 1,  g2         nop
  5:       s += v;                      ldub [o7 + i4], g3      RFUOP g1, 1
  6:    else                            srl  g2, 1,  g2
  7:       s -= v;                      subcc g2, g3, i3
  8:    Cont:                           bnge, a Cont
  9:                                    sub  g1, i3, g1
 10:                                    add  g1, i3, g1
 11:                                 Cont:
```

Fig. 5. Source code (left) of an inner loop of the MPEG2 encoder, code produced by the the gcc compiler with the -O2 flag (center), and the Chimaera RFU code (right).

achieve higher performance gains. So far, the results of this research have been very promising [24].

In order to test our results, we have taken various 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. Since we are working with the MIPS instruction set, all branches and loads are followed by a single delay slot which 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. 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 comes from the left 3-LUT in the cell, though it can be forced to true via the configuration.

Row	CAM	Flag	Computation
1	null		read p1[i], p1[i+1]; output v1 = p1[i] + p1[i+1]
2	null		output v2 = (v1 + 1) >> 1
3	null		read p2[i]; output v3 = v2 - p2[i]
4	null		neg = v3 < 0;
5	1	!neg	read s; output s + v3;
6	1	neg	read s; output s - v3

Fig. 6. Detailed view of the fifth and sixth rows for RFU instruction #1. The fifth row adds the values $v3$ and s, while the sixth row subtracts them. Depending on the neg value, which was computed in the previous row, one of these two rows is selected. Since columns 30 through 0 are identical, they are represented here as column X. Also, detailed routing on a multiplexer-by-multiplexer basis is not shown.

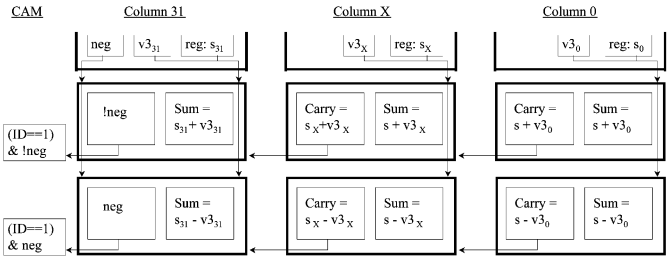


Fig. 7. RFU contents of the MPEG2 encoder example. The RFUOP selects from either the fourth or fifth row, depending on the value of the negative flag. This method saves an extra row, which otherwise would need to select from either the fifth or sixth row.

A. MPEG2 Encoding

MPEG2 encoding is part of the MediaBench [15] benchmark suite. The target applications for this benchmark are various types of compression and video processing. Roughly 89% of the execution time for MPEG2 encoding is spent within the function "dist1." For this reason, we aggressively targeted this function for RFU mappings. An example of one targeted area is shown in Fig. 5. The C code and resulting assembly are given. As shown, the code performs two additions, a shift and one subtraction in the source code lines 1–3, which typically take four instructions. However, we can do better than just mapping this basic block of code. Because of the RFU's ability to select from one of two rows for a given configuration with the decode CAM, we can compute both the if and else branch within the mapping as well. Upon execution, the RFU will test whether "$v >= 0$" and compute both line five and line seven in parallel. The result of the test is stored in the variable "neg" and passed to the last two rows of the RFU in order to compute the CAM value. The resulting assembly code is given.

As can be seen in Fig. 5, the entire section of code is broken down into one RFU operation, resulting in 50% fewer instructions needed for execution. Figs. 6 and 7 shows the RFU mapping for the code. The mapping took six rows. Rows one through three perform the two additions, the shift, and the subtraction. Row four tests whether the resulting value is less than zero. The last two rows perform both sections of the branch statement. The "neg" flag is sent to the instruction decode CAM. If "neg" is true, the F2 value from row six is returned from the RFU. In row five, we negate the "neg" flag and send the resulting value to the decode CAM, since we want this row's value if the "neg" flag is false. Since the result bus will read the value from the correct branch, an extra row to select from the values in rows five and six is saved.

Three different RFU configurations totaling 20 rows are used within the "dist1" function, the core of the motion estimation

kernel. Two of these mappings, including the one shown in Fig. 6, are used in multiple sections of the code. Because of these multiple uses for the configurations, RFU cache misses will decrease, resulting in less configuration overhead. Also note 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). With only these three mappings, we achieved a speedup of 1.59 times over the standard software-only version for the "dist1" function and a total improvement of 1.50 for the overall application. However, there are many other opportunities for such optimizations, and a more aggressive approach would result in further improvements.

B. Data Encryption Standard (DES) Encryption

DES has been the standard for encryption for the past 20 years, and is probably the most widely used encryption algorithm. DES encrypts blocks of 64 bits of data using a 56-bit key. It performs an initial permutation, cycles through 16 block rounds using precomputed subkeys, and performs a final permutation which is the inverse of the initial permutation. After an initial permutation, the 64-bit block of plain text is broken into a right and left half. Then for each round, the right half is combined with the round's subkey through a function f. The result of the computation is XOR'd with the original left half to produce the right half for the next round. The original right half becomes the new left half for the next round as well. After 16 rounds, the resulted two halves are combined through an inverse of the initial permutation to produce the 64 bits of cipher text. Thus, for each round, the calculation is

$$L = R_{(i-1)}$$
$$R_i = L_{(i-1)} \text{ XOR } F(R_{(i-1)}, K_i).$$

Function F is the block cipher shown in Fig. 8. It is implemented in Chimaera with the use of seven mappings. One for the initial permutation, one for the final permutation, four for the S-box substitution mappings, and one for the P-box permutation. The initial and final permutations are the inverse of each other. Both read in two registers (for the input 64 bits) and produce two rows of output (for the output 64 bits). The P-box permutation is similar to the initial and final permutations but operates only on 32 bits of data. Finally, the S-box substitution is broken up into four separate pieces. Each mapping has eight S-boxes within it and produces one bit for each box. For example, the first mapping calculates bit 0 for all eight S-boxes. Four of these mappings will produce the 32-bit result. Because of the ordering of data for the S-boxes, we broke the expansion permutation down to only a rotate left and a rotate right to produce the 48 bits from the 32 bits. The rotate left by 1 will produce one register where we will use all 32 bits. The rotate right by 2 will produce a second input for the S-box mappings, but we only use 16 of these values.

The routing for one box from the S-box mappings can be seen in Fig. 9. This is repeated eight times for the eight S-boxes. The mapping creates a six-input LUT which produces one bit for each S-box. Thus, four separate six-input LUTs are required for each S-box. In order to keep the S-box mappings to only three rows, we were able to take advantage of the fact that the carry tree can be used as an extra routing channel to pass signals from lower to higher bit positions.

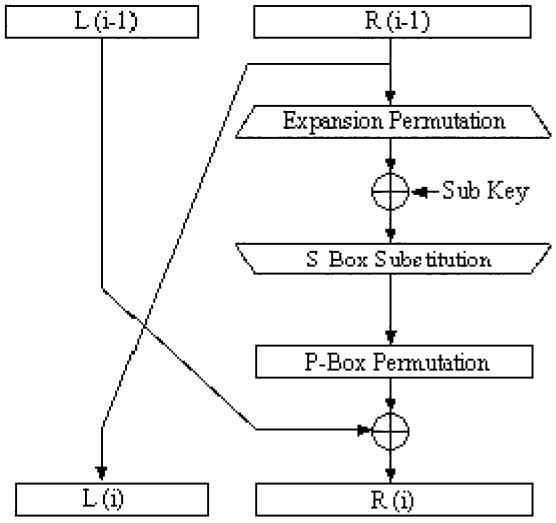


Fig. 8. Block round.

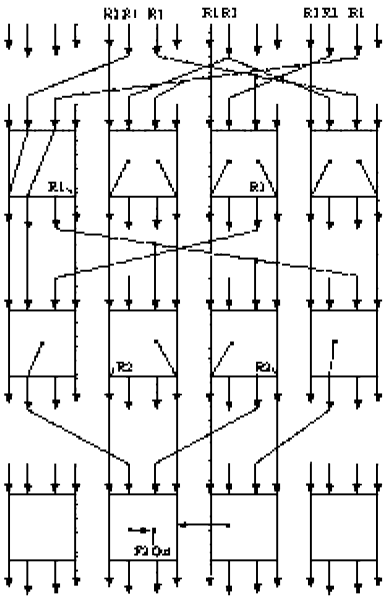


Fig. 9. S-box routing.

Name	Rows	Delay Cycles	Instances per Block
IP	8	3	1
S-Box	3	3	16
P-Box	5	3	16
IP^{-1}	8	3	1

Fig. 10. Mapping statistics.

The three permutation mappings essentially reorder the bits. Fig. 10 shows that these permutations execute very quickly, in less than three clock cycles, since they only use the routing channels of Chimaera to reorder the bits and do not use any internal logic of the cells (except to output the final result). In fact, any reorder of bits will typically take three clock cycles or less.

```
1:   RLA  R0, R2, 1
2:   RRA  R0, R3, 2
3:   LW   R4, M[0]
4:   LW   R5, M[1]
5:   XOR  C0, R2, R4
6:   XOR  C1, R3, R5
7:   NOP
8:   NOP
9:   CHIM R6, 'Sbox'
10:  SL   R7, R6, 2
11:  CHIM R6, 'Sbox'
12:  SL   R6, R6, 1
13:  XOR  R7, R6, R7
14:  CHIM R6, 'Sbox'
15:  XOR  R7, R6, R7
16:  CHIM R6, 'Sbox'
17:  SR   R6, R6, 1
18:  XOR  C2, R6, R7
19:  NOP
20:  NOP
21:  CHIM R8, 'Pbox'
```

Fig. 11. Assembly code for one block round of DES.

The entire block round from Fig. 8 can be reduced to only 21 instructions with the Chimaera RFU. Fig. 11 shows the assembly code for the block round. Our notation uses C[1–8] to represent a register the Chimaera array can read from. An interesting item to note is that even though the S-box mappings take three clock cycles to execute, the delay is only incurred during the first instance of the block. By realizing that although the processor does not have the ability to perform more than one calculation at a time, the RFU can be configured to compute multiple calculations at once. Thus, each subsequent instance of the S-box mapping runs in parallel with the first mapping and will have enough time to compute its value even before being called. By taking advantage of this parallel execution, six clock cycles per round (96 clock cycles per block) are saved.

To analyze the performance of the Chimaera DES implementation, we compared it to a software version of DES running on a MIPS processor. The software implementation we used was taken from the *Applied Cryptography* book by Bruce Schneier [25]. It is generally considered to be one of the fastest publicly available versions of DES out there. The software-only version requires 2725 total clock cycles per 64-bit block, while the Chimaera version executes in just 349 clock cycles. The final speed up yields 7.8 times faster execution.

C. Other Examples

For completeness, in addition to these two examples, we analyzed nine other applications and hand-mapped these to the Chimaera RFU. We attempted to select a broad range of applications including compression, encryption, and video processing, among others. These examples range from simple configurations such as the MPEG2 encoder to completely revamped algorithms that fully take advantage of the RFU, such as the skeletonization algorithm, DNA string comparison, and the Game of Life [9]. Overall, the performance numbers are very impressive. Typically, speedups of over 100% are achieved, and in some cases, speedups over 1000% are realized. In all of these applications, the algorithms are redone to take advantage of the fast bit-wise manipulations the Chimaera array offers.

V. CAD Mapping Software

In order to effectively make use of the RFU, the system must be able to identify sections of a program most often called during run time. Compiling and tech mapping, which is the topic of another paper [24], will produce a graph of logic blocks which must be placed into a LUT within the RFU and routed accordingly. The place and route CAD tools must be able to develop an efficient placement in reasonable time that utilizes as few rows of the RFU as possible. The goal of our place and route tool was not only to be able to compute a valid solution for ideal code segments, i.e., basic blocks of assembly code, but also to develop accurate solutions for hand-mapped net lists as in the Application Examples section. An efficient place and route tools is possible by taking advantage of the following properties of the Chimaera RFU:

1) the structure of the Chimaera RFU matches that of the host processor's ALU;
2) the downward flow of computations simplifies the allowed ordering of logic blocks and reduces the solution space;
3) the routing structures are limited, with usually only a few reasonable paths;
4) the number of logic cells to map is much smaller than a typical place and route job.

The Chimaera RFU was specifically designed to efficiently handle the typical instructions of a host processor. Each row of the RFU is capable of computing a logic operation such as AND, OR, XOR, XNOR, a simple arithmetic operation such as addition and subtraction, or a conditional branch such as greater than, less than, or some combination of the above. A basic block of executable code usually will have a logical placement where each line of code naturally maps within a row of the RFU and logic blocks line up with the bits within the shadow register file. Thus, the placement problem is greatly simplified by the RFU's structure and becomes more straightforward than placement of random logic.

The unidirectional flow of computations restricts and simplifies the placement problem further. Since computational results can only propagate to cells below themselves in the RFU, any logic block that reads data from another logic block must be positioned after the first block. By taking advantage of this property, each row of the RFU may be placed and routed by selecting from the list of unplaced cells which are ready, i.e., all of their inputs have already been calculated. Once a row is completed, the next row may be placed and routed from an updated list of available unplaced logic blocks. Since each row is placed somewhat independently of one another, the problem space is greatly reduced to placement of, at most, the 32 logic blocks for the current row. Last, the size of each mapping is much smaller than a typical design which must be placed and routed. The average number of logic blocks for the mappings of the sample applications above was less than 150 per mapping. With FPGA designs today reaching over 1 million gates, mappings for

the Chimaera RFU are much simpler. All of the properties above allow for a very fast place and route tool to be developed without sacrificing the quality of results.

A. Placement

The placement algorithm developed for the Chimaera RFU was designed to operate quickly while generating placements using the minimum number of rows. A valid placement has to meet the following requirements to fit within the structure of the RFU:

- logic blocks which produce an output (read by the host processor) must be placed in the correct column and within the same row as each other;
- each cell of the RFU may be mapped to one four-input logic block or two three-input logic blocks;
- values along the carry chain are read/written by three input LUTs, where one of the inputs is the current value of the carry chain.

The placement algorithm begins each row by generating a list of valid logic blocks. A valid logic block is one where all of the inputs are either read in from the shadow register file or have been produced by a logic block in a previous row. Each valid logic block is then scanned for the possible start of a carry tree. A valid use of the carry tree logic is identified when a three (or less)-input logic block:

- outputs to, at most, two three-input logic blocks;
- the two logic blocks' other inputs are identical;
- the two logic blocks' other inputs are ready at the current row.

These rules will identify standard arithmetic logic such as addition, where the chain of three-input logic blocks generates the carry value, and the second logic block generates the summation value. In addition, mappings will be compressed vertically by identifying ways of placing more logic blocks within the same row, which otherwise would have been placed in later rows. Once valid carry chains are identified, valid three (or less)-input logic blocks, which have two common inputs and can be placed within the same cell, are paired. This pairing will place as many logic blocks in the current row as possible and maximize cell density. Then blocks are selected from the longest carry chain to the shortest, and finally, individual logic blocks are randomly selected if space is left. The selection stops once 32 items have been selected for the current row or there are no more valid logic blocks for the row.

Once up to 32 blocks (a carry tree is considered to be one block during placement) have been selected for placement within the current row, a cost matrix is generated. The cost matrix values represent the relative difficulty of routing the connections of each logic block at each column. Higher costs represent more columns that a signal would have to travel. In addition, the cost values jump once more flexible routing logic has to be used (such as the longline routing logic, or the I3 input which can read values up to three columns away). The problem may now be thought of as minimizing the cost of placing 32 blocks in 32 bins, where there is an individual cost for each block in each bin. Such a problem is not NP-hard and may be computed efficiently by bipartite weighted matching. Using bipartite weighted matching, the run time required to find a minimum cost assignment is $O(n * (m + n \log n))$, where n is the number of nodes to the graph, and m is the number of edges. Given that n is, at most, 32, and m is, at most, 32^2, bipartite weighted matching will compute a solution within a reasonable amount of time. The Library of Efficient Data types and Algorithms (LEDA) was used to compute the minimum weighted assignment [14]. LEDA is a collection of common data types and algorithms and was selected for both its speed and ease of use.

Once the minimum cost placement is determined, logic blocks are placed in their optimal column for the current row. The input signals are then routed (described below). Routing is performed from the most difficult signal to the most direct. Because of the limited routing channels available, it may not be possible to produce a valid routing assignment for the current row's placement. When an input signal fails to be routed, it is determined that that logic block cannot routed if placed on the current row and it is removed from the list of valid logic blocks for the current row. Doing so effectively forces the logic block to be placed within a lower row so that more routing resources are available to route the signal. Several other checks are performed when a logic block is removed from the current row. If the logic block produces an output which is read from the RFU to the register file, then all logic blocks which produce a corresponding value to be read from the RFU, and thus must be on the same row, are removed from the current row. In addition, if the logic block was part of a carry chain four columns or longer, all of the logic blocks are removed from the current row. Otherwise, just the logic block that could not be routed is removed. Experience shows that longer carry chains generally are part of a common computation, such as a 16-bit addition, and should be maintained, while shorter carry chains are just opportunities to compress the mapping and may be separated as needed.

Once a logic block is removed from the list of valid logic blocks, the placement for the row starts over. Logic blocks which were available to be routed on the current row, but which were not selected, may now be selected as well. By continuing in this manner, logic blocks tend to be placed in the highest row possible given the routing structures available, thus yielding a mapping using the fewest rows possible. Once a valid placement and routing assignment has been found for the current row, the algorithm repeats for the subsequent row(s) until all logic blocks are placed and routed.

B. Routing

The routing algorithm developed is a greedy algorithm which attempts to route from the most difficult signals to the easiest. It does so by first selecting signals which read from another logic block just one row above, then two rows above, and so on. Signals which read from the shadow register file have the most flexibility (since they can be read from any row), and they are routed last. The routing order is further refined by selecting the signals which span the largest number of columns to those that span the least. Traditionally, routing algorithms in RFUs use a minimum cost analysis spanning all or multiple paths from the destination of the signal to the source. However, given the very limited routing structure available within the Chimaera RFU, there generally is a main best path which reaches the signal's source. The routing algorithm developed uses a greedy heuristic that attempts to find and select this path on the first pass.

Routing for each individual signal proceeds as follows. The inputs to the cell are scanned to see if it already exists along one of the available paths. An example of this would be if another logic block required the same signal and routed previously placed it along one of that row's long lines. The signal would then be available at every column within the row and may just be read, thus routing does not need to proceed all the way to the source. If the signal is not immediately available, given the number of columns left to travel and the current position of the signal (i.e., I1, I2, I3, or I4) a routing structure is selected from a predetermined set of rules. The signal has now been routed to the next row above in the mapping. If the source of the signal is within the current cell routing is complete, otherwise, routing continues in the same manner with the new starting point in the new row. The routing fails if the current row of signals is higher than its source row or it is in the top of the mapping (in the case of reading from the shadow register file). If the router failed, it can be assumed that there is no valid route for the signal, since the supposedly optimal route was being selected at each point. In reality, some valid paths will be missed; however, tests show that this situation is very rare and only occurs in a few unexpected situations. For the vast majority of mappings, a valid route will be found if there is one available. Further, this method is exceptionally fast. At each row, a case statement selects the path to the next row. The runtime of the routing job for each signal is $O(n)$ where n is the number of rows which the signal travels through (typically 1–5, and never more than the height of the mapping). Considering that the number of signals is, at most, four per column, or 128 per row, and the number of rows is a small value, the total number of steps required to route a row is lower than 1000. Since each step consists of mostly a large single-case statement, the routing time for a row is minimal. See Fig. 12 for application examples.

C. Results

Fig. 13 compares the results of the place and route tool for a variety of mappings. They are a selection of code segments mapped to the RFU and custom-generated mappings for specific applications. The ADPCM, DES, DNA, Gaussian and MPEG2 mappings are select mappings from the Application Examples section. The Adder functions are simple 32-bit additions and subtractions used to verify the use of the carry tree. Reference [17] describes an image skeletonization mapping and the numbers correspond to mappings shown in Figs. 13, 15, and 16 of that paper. Overall, the Chimaera place and route tools perform extremely well. For all but two mappings, the software tools produced results as well as a hand mapping, which ideally is the best possible solution.

The two examples where one or more rows were required were extremely complex mappings which most likely will not be generated automatically by a Chimaera compiler and will not have to be placed and routed during runtime. In order for the compiler to improve results to match the hand mappings, it would need to predict where logic block inputs values in future rows will be required and move data to those locations. Such predictions would decrease the fast performance of the Chimaera compiler's algorithm, with a relatively minor improvement in overall results. The DNA string comparison mapping involves dense random logic and three separate output rows. The software tools used eight rows, where a hand mapping only took seven. The Gaussian blur mapping involved nine additions and nine shifts; a custom hand mapping required 10 rows, where the software tools needed 13 rows.

Application	Number of Mappings	Total Rows	Speed up
DES encryption/decryption	6	24	**7.81**
Simple Gausian Blur	3	30	**3.53**
RGB-Grayscale conversion	3	27	**5.53**
RC5 64/8/12 encryption	1	21	**3.36**
Game of Life	2	8	**163.5**
Eqntott	2	7	**1.80**
Skeletonization Algorithm	6	14	**53.35**
DNA String Comparison	3	6	**24.54**
MPEG2 Encoder	3	17	**1.50**
ADPCM Coder	5	27	**2.01**
Compress	3	6	**1.11**

Fig. 12. Application examples.

Mapping	Number of Rows Software	Hand
Add & Subtract	2	2
Add, Sub, xor, xnor	4	4
ADPCM - 1	3	3
ADPCM - 3	6	6
ADPCM - 4	4	4
ADPCM - 5	2	2
DES - Initial Perm	8	8
DES - Pbox	5	5
DES - Skey	3	3
DNA - 4	8	7
Gausian - 1	13	10
Skeletonization - 13	2	2
Skeletonization - 15	5	5
Skeletonization - 16	7	7
MPEG Encoding - 1	4	4
MPEG Encoding - 4	7	7

Fig. 13. Mapping results.

The Chimaera CAD software tool was able to place and route each of the mappings in less than one second on a Sparc 5 workstation. By taking advantage of the limited structure of the Chimaera RFU, it is possible to design CAD software tools which limit the search space of the place and route problem. Such a high-speed placement and routing tool is able to generate high-quality mappings for designs in reasonable time, allowing for compilers to optimize applications during the execution of an application.

VI. Layout Results

The Chimaera test chip was designed using Magic version 6.5. The reconfigurable array was fabricated by MOSIS using a standard 0.5 μm complementary metal–oxide–semiconductor (CMOS) process (0.6 μm, effective with a 0.35 μm λ). Fig. 14

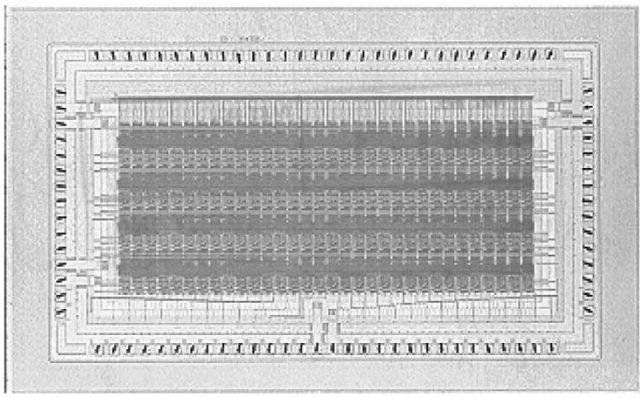


Fig. 14. Chimaera test chip.

shows the resulting test chip die. Each cell requires 52 programming bits, yielding 1664 bits, or 208 bytes, of uncompressed data to configure a single row. Research at the University of Washington (Seattle, WA) and Northwestern University has been done to compress these data [16]. A complete row is roughly 13 400 $\lambda \times 1400\ \lambda$, or 4.69 mm $\times$ 0.49 mm in a 0.6 μm process. Shrinking the array to the 0.13 μm drawn process of today's processors, means 32 Chimaera rows fit within a 1.016 mm $\times$ 3.3 mm area (3.35 mm^2), a relatively small amount of chip real estate. Thus, the array's size is small enough for several rows of the RFU to fit on today's processors.

To analyze the performance of the test chip, we provide the absolute timings, along with a comparison to a processor with a 150-MHz clock cycle. It was found that the 150-MHz clock cycle is the average clock cycle for a MIPS processor fabricated on a similar technology. Note that the first version of the Chimaera array which was fabricated used a 32:1 mux spread throughout the row for the longlines, instead of the bus logic shown previously in Fig. 4. This multiplexer method also ensures no forbidden configuration can ever be reached. However, because of the larger capacitances associated with such a single long wire, the multiplexer-based longline is expected to be slower than the previously mentioned method.

The routing channel numbers in Fig. 15 show the delay from the output muxes of the previous row to the input muxes of the current row, while the longline rows include the cost of using a longline. For the internal cell logic paths, refer to Fig. 3. 2LUT is the left LUT made out of the two 2LUTs and 3LUT refers to the 3LUT on the right side. The reason using the X and D signals are faster than using the A or B signals is that the computation does not actually use a LUT. Rather, it will switch between different LUTs by using mux 6 for the X signal and mux 10 for the D signal. However, if the X or D signal is routed through mux 5 and into the 3LUT, this improvement in speed is lost.

VII. Conclusions

Current reconfigurable systems deliver huge speedups for some types of applications. However, because of the communication bottleneck between the reconfigurable logic and the host processor, these algorithms require a significant effort at hand optimizations. In addition, the migration of a significant amount of computation to the reconfigurable logic is needed to overcome the communication delay.

	Path	Time (ns)	Clock Cycle %
Routing Channels			
	In1	1.2	18.0%
	In2	1.9	28.5%
	In3	2.5	37.5%
	In4	1.2	18.0%
	Long Line A	5.7	85.5%
	Long Line B	6.2	93.0%
Internal Cell Logic			
	A,B $\rightarrow$ 2LUT $\rightarrow$ F1	3.0	45.0%
	X $\rightarrow$ 2LUT $\rightarrow$ F1	2.1	31.5%
	A,B,X $\rightarrow$ 3LUT $\rightarrow$ F2	2.5	37.5%
	D $\rightarrow$ 3LUT $\rightarrow$ F2	1.6	24.0%
	A,B $\rightarrow$ 2LUT $\rightarrow$ Carry $\rightarrow$ 3LUT F2	6.5	97.5%

Fig. 15. Test-chip timing numbers.

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 RFU 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 multicycle 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.

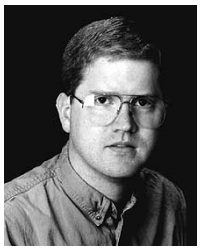
From the timing numbers, it can be seen that computations have time to flow through a couple rows within one clock cycle. In addition, through several hand mappings to the RFU, we have demonstrated the power of the Chimaera system. More simple basic block mappings (Gaussian Blur, Eqntott, and MPEG2 encoding) offer speedups of two to four times at a cost of roughly 3.35 mm^2 of die area. These optimizations required only local optimization of a small amount of the source code, using transformations that should be possible to achieve in an automatic mapping system. Completely revamped applications, coded specifically for the Chimaera system (such as DNA string comparison, the skeletonization algorithm, Conway's Game of Life and DES encryption/decryption) offer 7–160 times improvement in running times. Such careful hand optimizations demonstrate the potential high performance of the system.

The limited, yet powerful, interconnect structure of Chimaera also lends itself well to automatic compilation. As was demonstrated here, a very fast and efficient placement and routing system has been developed that almost always achieves results identical to hand mapping, yet runs in less than one second.

References

[1] O. T. Albaharna, P. Y. K. Cheung, and T. J. Clarke, "Area & time limitations of FPGA-based virtual hardware," in *Proc. Int. Conf. Computer Design*, 1994, pp. 184–189.

[2] ——, "On the viability of FPGA-based integrated coprocessors," in *Proc. IEEE Symp. FPGA for Custom Computing Machines*, Apr. 1996, pp. 206–215.
[3] *Data Book*. San Jose, CA: Altera Corp., 1995.
[4] G. Borriello, C. Ebeling, S. Hauck, and S. Burns, "The triptych FPGA architecture," *IEEE Trans. VLSI Syst.*, vol. 3, pp. 491–501, Dec. 1995.
[5] J. E. Carrillo and P. Chow, "The effect of reconfigurable units in superscalar processors," in *ACM/SIGDA Symp. Field-Programmable Gate Arrays*, 2001.
[6] A. DeHon, "DPGA-coupled microprocessors: Commodity IC's for the early 21st century," in *Proc. IEEE Workshop FPGA for Custom Computing Machines*, 1994, pp. 31–39.
[7] C. Ebeling, L. McMurchie, S. Hauck, and S. Burns, "Placement and routing tools for the triptych FPGA," *IEEE Trans. VLSI Syst.*, vol. 3, pp. 473–482, Dec. 1995.
[8] P. C. French and R. W. Taylor, "A self-reconfiguring processor," in *Proc. IEEE Workshop FPGA for Custom Computing Machines*, 1993, pp. 50–59.
[9] M. Gardner, "Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life"," *Sci. Amer.*, pp. 120–123, Oct. 1970.
[10] S. Hauck, G. Borriello, and C. Ebeling, "TRIPTYCH: An FPGA architecture with integrated logic and routing," in *Proc. 1992 Brown/MIT Conf. Advanced Research in VLSI and Parallel Systems*, Mar. 1992, pp. 26–43.
[11] S. Hauck, "Multi-FPGA systems," Ph.D. dissertation, Dept. Comp. Sci. & Eng., Univ. Washington, , Seattle, WA, 1995.
[12] S. Hauck, M. M. Hosler, and T. W. Fry, "High-performance carry chains for FPGAs," *IEEE Trans. VLSI Syst.*, vol. 8, pp. 138–147, Apr. 2000.
[13] H. S. Kim, A. K. Somani, and A. Tyagi, "A reconfigurable multi-function computing cache architecture," in *Proc. ACM/SIGDA Symp. Field-Programmable Gate Arrays*, 2000.
[14] LEDA, the Library of Efficient Data Types and Algorithms. Algorithmic Solutions. [Online]. Available: http://www.algorithmic-solutions.com/as_html/products/products.html
[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," Univ. California at Los Angeles, Los Angeles, CA, 1997.
[16] Z. Li, "Configuration management techniques for reconfigurable computing," Ph.D. dissertation, Dept. of ECE, Northwestern Univ., Evanston, IL, 2002.
[17] K. Nelson and S. Hauck, "Mapping methods for the Chimaera reconfigurable functional unit," Northwestern Univ., Evanston, IL, Dept. ECE Tech. Rep., 1997.
[18] S. Rajamani and P. Viswanath, "A quantitative analysis of processor-programmable logic interface," in *Proc. IEEE Symp. FPGA for Custom Computing Machines*, Apr. 1996, pp. 226–234.
[19] R. Razdan, "PRISC: programmable reduced instruction set computers," Ph.D. dissertation, Harvard Univ., Div. Appl. Sci., Cambridge, MA, 1994.
[20] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. Int. Symp. Microarchitecture*, 1994, pp. 172–180.
[21] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid, "Using on-chip configurable logic to reduce embedded system software energy," presented at the *IEEE Symp. FPGA for Custom Computing Machines*, 2002.
[22] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *Proc. IEEE Symp. FPGA for Custom Computing Machines*, Apr. 1995, pp. 99–107.
[23] R. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proc. IEEE Symp. FPGA for Custom Computing Machines*, Apr. 1996, pp. 126–135.
[24] Z. A. Ye, N. Shenoy, S. Hauck, P. Banerjee, and A. Moshovos, "CHIMAERA: A tightly-coupled reconfigurable unit/high-performance processor architecture," presented at the *Int. Symp. Computer Architecture*, 2000.
[25] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 1995.



Scott Hauck (S'94–M'95–SM'01) received the B.S. degree in computer science from the University of California, Berkeley, in 1990, and the M.S. and Ph.D. degrees in computer science from the University of Washington, Seattle, in 1992 and 1995, respectively.

He was an Assistant Professor at Northwestern University, Evanston, IL, from 1995–1999 before joining the University of Washington's Department of Electrical Engineering, where he is currently an Associate Professor. His research focuses on FPGAs and reconfigurable computing, including applications, architectures and CAD tools for these devices.

Dr. Hauck has received an NSF CAREER Award, an IEEE TRANSACTIONS ON VLSI Best Paper Award, and is a Sloan Research Fellow.

Thomas W. Fry received the B.S. degree in computer engineering from Northwestern University, Evanston, IL, in 1998, and the M.S. degree in electrical engineering from the University of Washington, Seattle, in 2001.

He was a Research Assistant at the University of Washington from 1999–2001, before joining IBM Microelectronics, Waltham, MA, in 2001, where he is a Physical Design Methodology Engineer. His research focuses on reconfigurable and adaptive logic design, high-performance clock distribution techniques, FPGA architectures, image processing, and compression techniques.

Matthew M. Hosler received the B.S. degree in electrical engineering from the University of Dayton, Dayton, OH, and the M.S. degree in computer engineering from Northwestern University, Evanston, IL.

He was with Motorola's Corporate Research Labs, Schaumburg, IL, where he developed new reconfigurable arrays optimized for digital signal processing. He is currently a Field Applications Engineer for Arrow Electronics, Dayton, OH.

Jeffrey P. Kao received the B.S. degree in computer engineering in 1997 from Northwestern University, Evanston, IL. He is currently working towards the MBA degree at the University of Michigan, Ann Arbor.

He was with Intel Corporation in Chandler, AZ, as a Technical Marketer and Senior Design Engineer from 1997–2003. His industry experience includes Xscale processor performance analysis and design work on three generations of the Itanium processor.