CHIMAERA: Integrating a Reconfigurable Unit into a High-Performance, Dynamically-Scheduled Superscalar Processor

Zhi Alex Ye, Andreas Moshovos, Scott Hauck*, Nagaraj Shenoy and Prithviraj Banerjee

Electrical and Computer Engineering
Northwestern University
{ye,moshovos,banerjee}@ece.northwestern.edu

*Electrical Engineering,
University of Washington,
hauck@ee.washington.edu

Abstract

We report our experience with Chimaera, a prototype system that integrates a small and fast reconfigurable functional unit (RFU) into the pipeline of an aggressive, dynamically-scheduled superscalar processor. We discuss the Chimaera C compiler that automatically maps computations for execution in the RFU.Using a set of multimedia and communication applications we show that even with simple optimizations, the Chimaera C compiler is able to map 22% of all instructions to the RFU on the average. Timing experiments demonstrate that for a 4-way out-of-order superscalar processor Chimaera results in average performance improvements of 21%. This assuming a very aggressive core processor design (most pessimistic RFU latency model) and communication overheads from and to the RFU.

1 Introduction

Conventional instruction set architectures (ISAs) provide primitives that facilitate low-cost and low-complexity implementations while offering high performance for a broad spectrum of applications. However, in some cases, offering specialized operations tailored toward specific application domains can significantly improve performance. Unfortunately, deciding what operations to support is challenging. These operations should be specialized enough to allow significant performance benefits, and at the same time, they should be general enough so that they are useful for a variety of applications. More importantly, we have to decide whether any of the current performance benefits justify the risks associated with introducing new instructions. Such instructions may become defunct as software evolves and may adversely impact future hardware implementations.

Reconfigurable hardware provides a convenient way of addressing most of the aforementioned concerns. It may significantly improve performance by tailoring its operation on a per application basis. Moreover, since the type of specialized operations is not fixed in the ISA, reconfigurable hardware has the potential to evolve with the applications. As increasingly higher levels of on-chip resources are anticipated, reconfigurable capable systems provide a potentially fruitful way of utilizing such resources. However, for this potential to materialize we also need methods of converting software so that we can exploit it. While it is possible to hand-map applications to exploit reconfigurable hardware, writing working software is already complicated enough. For this reason, an automated process is highly desirable. In this paper, we discuss our experience with designing Chimaera [9], a reconfigurable-hardware-based architecture and our experience with providing compiler support for it. In particular, in this paper we: (1) review the design of Chimaera, (2) explain how it can be integrated into a modern, dynamically-scheduled superscalar pipeline, (3) describe the compiler optimizations we used to exploit Chimaera, and (4) study the resulting performance tradeoffs.

Chimaera tightly couples a processor and a reconfigurable functional unit (RFU). This RFU is a small and fast field-programmable-gate-array-like (FPGA) device which can implement application specific operations. For example, an RFU operation (RFUOP) can efficiently compute several data-dependent operations (e.g., tmp=R2-R3; R5=tmp+R1), conditional evaluations (e.g., if (a>88) a=b+3), or multiple sub-word operations (e.g., "a = a + 3; b = c << 2", where a, b and c are half-word long). In Chimaera, the RFU is capable of performing computations that use up to 9 input registers and produce a single register result. The RFU is tightly integrated with the processor core to allow fast operation (in contrast to typical FPGAs which are build as discrete components and that are relatively slow). More information about the Chimaera architecture is given in Section 2.

Chimaera has the following advantages:

1. The RFU may reduce the execution time of dependent operations which can be reduced in a single, lower-latency RFU operation.
2. The RFU may reduce dynamic branch count by collapsing code containing control flow into an RFU operation. In this case the RFU speculatively executes all branch paths and internally selects the appropriate one.
3. The RFU may exploit sub-word parallelism. Using the bit-level flexibility of the RFU, several sub-word operations can be performed in parallel. While this is similar to what typical multimedia instruction set extensions do, the RFU-based approach is more general. Not only the operations that can be combined are not fixed in the ISA definition, but also, they do not have to be the same. For example, an RFU operation could combine 2 byte adds and 2 byte subtracts. Moreover, it could combine four byte-wide conditional moves.
4. The RFU may reduce resource contention as several instructions are replaced by a single one. These resources include instruction issue bandwidth, writeback bandwidth, reservation stations and functional units.

To exploit the aforementioned opportunities we have developed a C compiler for Chimaera. We found that even though our compiler uses very simple, first-cut optimizations, it can effectively map computations to the RFU. Moreover, the computations mapped are diverse.

In this paper, we study the performance of Chimaera under a variety of both timing and RFU mapping assumptions ranging from optimistic to very pessimistic. We demonstrate that, for most

programs, performance is sensitive to both the latency of the RFU and the aggressiveness of the synthesis process (in synthesis we map a set of instructions into an RFU operation and construct the RFU datapath). For some programs, Chimaera offers significant performance improvements even under pessimistic assumptions. Under models that approximate our current prototype of Chimaera's core RFU, we observe average speedups in between 31% (somewhat optimistic) and 21% (somewhat pessimistic).

The rest of this paper is organized as follows: In Section 2 we review the Chimaera RFU architecture and discuss how we integrate the RFU into a typical superscalar pipeline. In Section 3 we discuss the compiler support. In Section 4 we review related work. In Section 5 we present our experimental results. Finally, in Section 6 we summarize our findings.

2. The Chimaera Architecture

The Chimaera architecture, as we show in Figure 1 (more detailed information about the RFU can be found in [9,39]), comprises the following components: (1) The *reconfigurable array* (RA), (2) the *shadow register file* (SRF), (3) the *execution control unit* (ECU), and (4) the *configuration control and caching unit* (CCCU). The RA is where operations are executed. The ECU decodes the incoming instruction stream and directs execution. The ECU communicated with the control logic of the host processor for coordinating execution of RFU operations. The CCCU is responsible for loading and caching configuration data. Finally, the SRF provides input data to the RA for manipulation.

In the core of the RFU lies the RA. The RA is a collection of programmable logic blocks organized as interconnected rows (32 in our prototype). The logic blocks contain lookup tables and carry computation logic. Across a single row, all logic blocks share a fast-carry logic which is used to implement fast addition and subtraction operations. By using this organization, arithmetic operations such as addition, subtraction, comparison, and parity can be supported very efficiently. The routing structure of Chimaera is also optimized for such operations.

During program execution, the RA may contain configurations for multiple RFU operations (RFUOPs). A configuration is a collection of bits that when appropriately loaded in the RA implements a desired operation. Managing the RA-resident set of RFUOPs is the responsibility of the ECU and the CCCU. The CCCU loads configurations in the RA, provides fast access to recently evicted configurations through caching, and provides the interfaces necessary to communicate with the rest of the memory hierarchy. The ECU decodes the instruction stream. It detects RFUOPs and guides their execution through the RA and if necessary notifies the CCCU of currently unloaded configurations.

Each RFUOP instruction is associated with a configuration and a unique ID. The compiler generates configurations and their IDs. The linker places these configurations into the program's address-space and also generates a vector table pointing to the beginning of each generated configuration. At run-time, and upon detection of an RFUOP, the ECU initiates a trap to load the appropriate configuration in the RA. While the configuration is being loaded, execution is stalled. In our prototype implementation, each row requires 1674 bits of configuration.

Input data is supplied via the Shadow Register File (SRF) which is a physical, partial copy of the actual register file (only registers 0 through 8 are copied). It is organized as a single row containing copies of all logical registers. This allows single register write access from the host processor and allows the RA to read all registers at once.

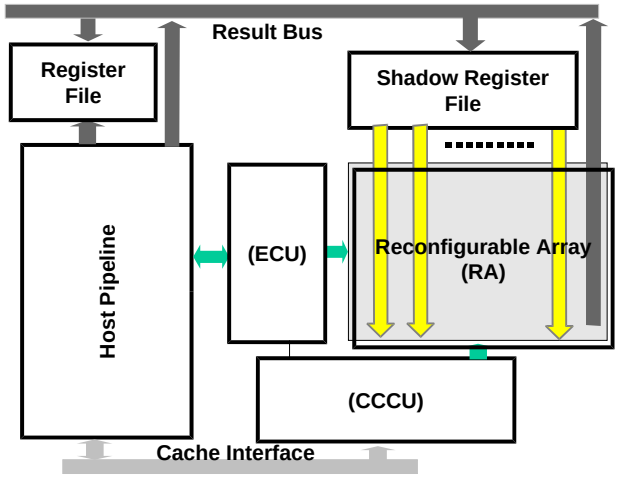

Figure 1: *Overview of the Chimaera Architecture*

To interface with the out-of-order core and to allow out-of-order execution of RFUOPs, we provide a separate, small RFUOP scheduler. This scheduler follows the RUU model [27]. Upon encountering an RFUOP, the ECU allocates a dummy entry in the scheduler of the OOO core. This entry is used to maintain in-order commits and to support control-speculative execution (the OOO notifies the RFUOP scheduler of miss-speculations using the dummy entry). Based on the input vector data (which is part of the configuration data), the ECU also allocates an entry in the RFUOP scheduler marking the location of all desired input registers (this is done by maintaining a shadow register renaming table that allows single cycle access to all entries). Moreover, the single target register of the RFUOP is renamed by the OOO core. RFUOP scheduling proceeds in the same fashion as regular instruction scheduling. In all experiments we assume a single-issue capable RFUOP scheduler since this significantly simplifies its design and allows easy integration with the current RA prototype.

A standalone prototype of the RA was fabricated and is tested. The chip was fabricated in a .5 um, 3-layer CMOS process using MOSIS. The worst case path through a single logic block in the current prototype consists of 23 transistor levels. Modern microprocessors exhibit great variety on the number of transistor levels operating within a single clock cycle. For example, an aggressive implementation allows up to 12 transistor levels per clock cycle [5] (six 2-input gates) while another design allows up to 24 transistors levels per clock cycle [11] (eight 3-input gates). By utilizing the Elmore delay model [34], we estimated the worst case delay through each RA row to be within 0.96 to 1.9 cycles for implementations with 24 and 12 transistor levels respectively.

3. Compiler Support

We have developed a C compiler for Chimaera to automatically map groups of instructions to RFUOPs. The compiler is built over the GCC framework, version 2.6.3. We introduced the following three RFUOP-specific optimizations: *Instruction Combination*, *Control Localization*, and *SIMD Within A Register* (SWAR).

4.1 Instruction Combination

The core optimization is Instruction Combination, which extracts RFUOPs from a basic block. It works by analyzing the DFG and by extracting subgraphs that consist of multiple RFU-eligible (RE) nodes. RE nodes correspond to instructions that can be mapped in the RFU (e.g., adds, logic operations and shifts). Each sub-graph must have a single register output (intermediate outputs are allowed provided that they are not used outside the sub-graph). Recall, that an RFUOP can produce only a single register result. Each of the subgraphs is mapped to an RFUOP.

The algorithm *FindSequences* in Figure 2 identifies maximum RE instruction subgraphs. It works by coloring dataflow-graph nodes for a single basic block. Non-RE instructions are marked as BLACK. An RE instruction is marked as BROWN when its output must be written into a register, that is, the output is either alive after the basic-block or is being read by a non-RE instruction. Other RE instructions are marked as WHITE. Subgraphs consisting of BROWN and WHITE nodes are eligible for RFUOP generation.

The compiler currently changes all RFU-eligible subgraphs into RFUOPs. However, to control RFUOP generation we use profiling to identify the most frequently running functions. Moreover, instruction combination is limited to the basic blocks of the inner-most loops of these functions. **FIX: what policy is being used? Top 10, 90% of the time?**

4.2 Control Localization

To increase opportunities for instruction combination, the compiler first performs control localization. This optimization transforms branch containing sequences into temporary, aggregate instructions which can be treated as a single nodes by instruction combination [17]. This optimization works by first identifying RFU-eligible basic-blocks (REBB). These are basic-blocks whose instructions are all either RE (RFU-eligible) or branches. Two adjacent REBBs are combined into an aggregate instruction if the total number of input operands is less than nine and the total number or live-on-exit output operands is one. **DOUBLE CHECK THIS.**

4.3 SIMD Within A Register

The SWAR optimization identifies sub-word operations that can be executed in parallel. The current implementation attempts to pack several 8-bit operations into a single word operation. While this appears similar to existing SIMD ISA extensions, the RFU-supported SIMD model is much more general. Not only the operations that can be combined are not fixed in the ISA definition, but also, they do not have to be the same. For example, an RFU operation could combine 2 byte adds and 2 byte subtracts. Moreover, it could combine four byte-wide conditional moves. For example, four iterations of the following loops can be combined into a single RFUOP:

```
char out[100],in1[100],in2[100];
   for(i=0;i<100;i++)
          if ((in1[i]-in2[i])>10)
                 out[i]=in1[i]-in2[i];
          else
                 out[i]=10;
```

Unfortunately, due to the lack of an alias analysis phase, our current prototype cannot apply this optimization without endangering correctness. For this reason, we have disabled this optimization for all experiments.

```
ALGORITHM: FindSequences
INPUT: DFG G, Non-RE, RE, Live-on-exit registers R
OUTPUT: A set of RFU sequences S
begin
   S=∅
   for each node n∈G
      Color(n) ←WHITE
   end
   for each node n∈Non-RE
      Color(n) ←BLACK
      for each node p∈Pred(n)
         if p∈PE then
            Color(p) ←BROWN
         endif
      end
   end
   for each register r∈R
      n← the last node that updates r in G
      if n∈PE then
         Color(n) ←BROWN
      endif
   end
   for each node n∈G
      if Color(n)=BROWN then
         sequence=∅
         AddSequence(n, sequence)
         if sizeof(sequence)>1 then
            S=S∪{sequence}
         endif
      endif
   end
end

AddSequence(n, sequence)
begin
   if Color(n)=(BROWN or WHITE) then
      sequence←sequence∪{n}
      for each p∈Pred(n)
         AddSequence(p, sequence)
      end
   endif
end
```

Figure 2: *Instruction Combination Algorithm*

4.4 An Example of RFUOP Generation

Figure 3 shows an example of the compilation process on the adpcm_decoder function which is appear in the *adpcm.dec* benchmark (see Section 5 for a description of the benchmarks). Part of the original source code is shown in part (a). The code after control localization is shown in (b). The "d=tempx(s1,...,sn)" notation refers to a temporary instruction whose source operands are s1 to sn and destination is d. As shown, the three "if" statements are first converted into three temporary instructions, forming a single-entry/single-exit instruction sequence. The instruction combination phase then maps all three instructions into a single RFUOP, as shown in part (c).

4. Related Work

Numerous reconfigurable-hardware-based architectures have been proposed. We can roughly divide them into two categories, those that target coarse, loop-level optimizations and those that target fine-grain, instruction-level optimizations. The two approaches are complementary.

```
Adpcmdecoder()
{
int vpdiff, step, delta;
...
vpdiff = step >> 3;
if ( delta & 4 ) vpdiff += step;
if ( delta & 2 ) vpdiff += step>>1;
if ( delta & 1 ) vpdiff += step>>2;
...
}
```
(a)

```
Adpcmdecoder()
{
int vpdiff, step, delta;
...
vpdiff = step >> 3;
vpdiff=temp1(delta, vpdiff, step);
vpdiff=temp2(delta, vpdiff, step);
vpdiff=temp3(delta, vpdiff, step);
...
}
```
(b)

```
Adpcmdecoder()
{
int vpdiff, step, delta;
...
vpdiff=rfuop(delta,vpdiff, step)
...
}
```
(c)

Figure 3: *An example of the Chimaera optimizations. (a) Original code. (b) Code after control localization. (c) Code after instruction combination. The example is taken from the adpcm.dec Mediabench benchmark.*

The loop-level systems are capable of highly-optimized implementations (e.g., a pipeline) for whole loops. GARP [13], Napa [26], PipeRench [8], Rapid [4], Xputer [10], and RAW [33] are examples of such systems. Instruction-level systems target fine-grain specialization opportunities. Chimaera [9], PRISC [24, 25], DISC [32], and OneChip [30] are instruction-level systems. Chimaera differs from other systems primarily in that it supports a 9-input/1-output instruction model.

Restricted forms of optimizations similar to those we described can be found in several existing architectures. Many architectures provide support for collapsing a small number of data dependent operations into a single, combined operation. For example, many DSPs provide Multiply/Add instructions, e.g., [7,20]. More general multiple operation collapsing functional units have also been studied [22,28].

Most current ISAs have added support multimedia applications in the form of SIMD subword operations [14,18,19,21,23,29]. Chimaera provides a more general model subword-parallelism model as the operation itself is not restricted by the ISA.

Finally, Chimaera can map code containing control-flow into a single operation. A similar effect is possible with predicated execution (e.g., [1,2,6,16]). or with multiple-path execution models [15, 31].

5. Evaluation

In this section, we present our experimental analysis of a model of the Chimaera architecture. In Section 5.1, we discuss our methodology. There we also discuss the various RFUOP latency models we used in our experiments. In Section 5.2, we present an analysis of the RFUOPs generated. In Section 5.3, we measure the performance impact of our RFU optimizations under an aggressive, dynamically-scheduled superscalar environment.

5.1 Methodology

We used benchmarks from the Mediabench [19] and the Honeywell [12] benchmark suites. Table 2 provides a description of these benchmarks. The Honeywell benchmark suite has been used extensively in testing the performance of reconfigurable systems. For all benchmarks we have used the default input data set. While, in some cases the resulting instruction count appears relatively small, we note that due to their nature, even such short runs are indicative of the program's behavior. We have compiled these benchmarks, using the Chimaera C Compiler

For performance measurements we have used execution-driven timing simulation. We build our simulator over the Simplescalar simulation environment [3]. The instruction set architecture (ISA) is an extension of the MIPS ISA with embedded RFUOPs. By appropriately choosing the opcode and the Rd field of the RFUOP format, RFUOPs appear as NOOPs under the MIPS ISA.

For our experiments we have used the base configuration shown in Table 3. This models an aggressive, 4-way dynamically-scheduled superscalar processor. The RFU configuration we used is also shown in Table 3. When the RFU is in place the maximum number of instructions that can pass through decode, fetch, write-back and commit is still limited to 4 including any RFUOPs. Furthermore, only a single RFUOP can issue per cycle.

5.1.1 Modeling RFUOP Latency

To study performance it is necessary to express RFUOP execution latencies in terms of processor cycles. These latencies can be modeled accurately using a specific processor/RFU implementation and a synthesis (i.e., RFU configuration) algorithm. While valuable, the utility of such a model will be limited to the specific configuration and synthesis algorithm. Since our goal is to understand the performance tradeoffs that exist in the Chimaera architecture, we have experimented with several RFUOP latency models which are summarized in Table 4.

We use a two-tiered approach. First, we utilize latency models that are based on the original instruction sequence each RFUOP replaces. These models provide us with insight on the latencies the RFU should be able to sustain to make this a fruitful approach. These are reported as original-instruction-based models in Table 3. Models C, 2C and 3C model RFUOP latency as a function of the critical path "c" of the equivalent original program computation. To provide additional insight we also modeled fixed RFU latencies of 1, 2 and *n* cycles where *n* is the number of the original program instructions mapped in the RFUOP. The 1 and 2 cycle models offer upper bounds on the performance improvements possible with the current Chimaera compiler.

We also utilize transistor-level-based models. We first hand-mapped each RFUOP into an efficient RFU configuration and measured the number of transistor levels appearing in the critical path. We then calculated latencies for various base processor configurations. Using published data on the number of transistor levels per clock cycle for modern processors we developed the following four timing models: P24_0, P12_0, P24_1 and P12_1. Models P24_0 and P12_0 assume designs with 24 and 12 transistor levels per cycle. P24_0 corresponds to a design with eight 3-input gates per clock cycle such as the one in [11]. P12_0 assumes a more aggressive base processor pipeline with only six 2-input gates per clock cycle, such as the one as in [5]. To model the possibility of extra delays over the interconnect to and from the RFU we also include models P24_1 and P12_1 which include an additional cycle of latency over P24_0 and P12_0 respectively.

Component	Configuration
Superscalar Core	
Branch predictor	*64k GSHARE*
Scheduler	*Out-of-order issue of up to 4 operations per cycle, 128 entry re-order buffer (RUU), 32 entry load/store queue(LSQ)*
Functional units	*4 integer ALUs, 1 integer MULT, 4 FP adders, 1 FP mult/div*
Functional unit latencies	*Integer ALU 1, integer MULT 3, integer DIV 12, FP adder 12, FP MULT 4, FP DIV 12, load/store 1*
Instruction cache	*32kb Direct-Mapped, 32-byte block, 1 cycle hit latency*
Data cache	*32kb Direct-Mapped, write-back, write-allocate, non-blocking, 32-byte blocks, 1 cycle hit latency*
L2 cache	*Unified 4-way set associative, 128k byte, 12 cycles hit latency*
Main memory	*Infinite size, 100 cycles latency*
Fetch Mechanism	*Up to 4 instructions per cycle*
Reconfigurable Functional Unit	
Scheduler	*8 entries. Each entry corresponds to a single RFUOP* *Single Issue, Single Write-back per cycle.* *An RFUOP can issue if all its inputs are available and no other instance of the same RFUOP is currently executing.*
Functional Unit / RA	*32 rows. Each RFUOP occupies as many rows as instructions of the original program it replaced (pessimistic)* *Only a single instance of each RFUOP can be active at any given point in time.*
Configuration Loading	*1-st level configuration cache of 32 configuration rows (32 x 210 bytes).* *Configuration loading is modeled by injecting accesses to the rest of the memory hierarchy. Execution stalls for the duration of configuration loading.*
RFUOP Latency	*Various model simulated. See Section 5.1.1.*

Table 1: *Base configuration for timing experiments.*

Benchmark	***Description***	***Inst. Count***
MediaBench Benchmarks		
Mpegenc	Mpeg encoder	1139.0 M
G721enc	CCITT G.721 voice encoder	309.0 M
G721dec	CCITT G.721 voice decoder	294.0 M
Adpcm enc	Speech compression	6.6 M
Adpcm dec	Speech decompression	5.6 M
Pegwitkey	Pehwit key generation. Pegwit is a public key encryption and authentication application.	12.3 M
Pegwitenc	Pegwit encryption	23.9 M
Pegwitdec	Pegwit decryption	12.5 M
Honeywell Benchmarks		
Comp	Image compression	34.1 M
Decomp	Image decompression	32.7 M

Table 2: *Benchmark characteristics.*

Model P24_0 is the most optimistic while model P12_1 is the most pessimistic.

5.2 RFUOP Analysis

In this section we present an analysis of RFUOP characteristics. We measure the total number of instructions mapped to RFUOPs and we take a close look at the internals of some of the RFUOPs. Finally, we present results on the number of transistor levels used to implement RFUOPs in the RA.

Table 4 shows statistics on the number of instructions mapped to RFUOPs. Under the "IC" columns we report the dynamic instruction count of the Chimaera optimized program. This is expressed as a percentage of the original instruction count (shown in Table 3). We also report the fraction of the original instructions that were mapped to RFUOPs ("Red." column). The remaining eight columns provide a per instruction type breakdown of the mapped instructions. Shown is the percentage of instructions of each type in the original program ("Orig." columns) and the portion of this percentage ("Opt." columns) that was mapped to RFUOPs in the Chimaera optimized program. For example, for *adpcmenc*, 34% of all instructions was mapped to RFUOPs resulting in a reduction of 19% in dynamic instruction count. The original program had 27% branches and 37% of them (i.e., 9.9% of all instructions) was mapped to RFUOPs. We can observe that

Instruction-based Models						
Model	C	2C	3C	1	2	N
CPU cycles	c	2*c	3*c	1	2	n
Transistor-Level-based Models						
Model	P24_0		P24_1	P12_0	P12_1	
CPU cycles	⌈t/24⌉		⌈t/24⌉+1	⌈t/12⌉	⌈t/12⌉+1	

Table 3: *Timing Models.*

a significant fraction of instructions is mapped to RFUOPs (22% on the average). The actual percentage varies from as little as 8% to as much as 58%. More importantly, a significant fraction of branches is eliminated (18% on the average). Some of these branches foil the GSHARE predictor. Also, relatively large fractions of shift operations are mapped to RFUOPs as compared to other instruction types.

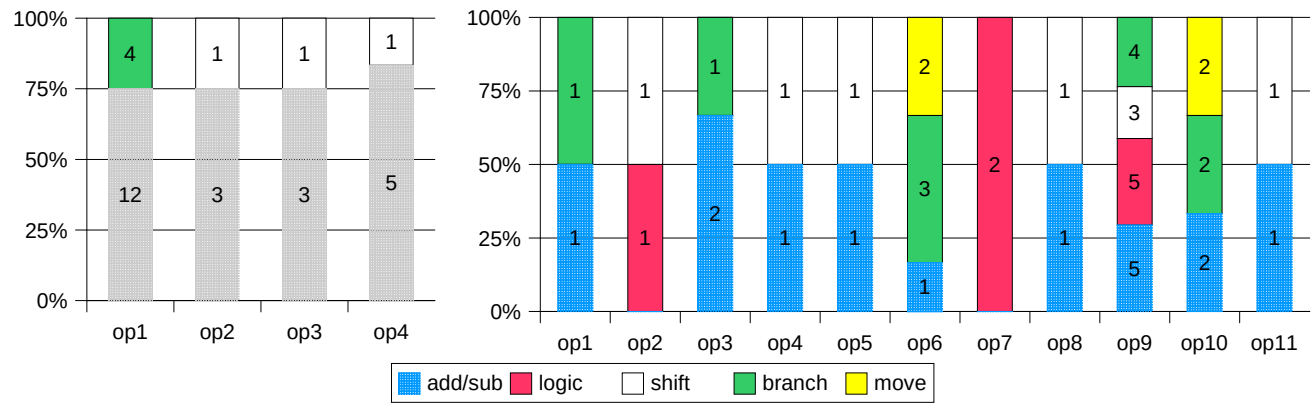

Figure 4: *RFUOP instruction type composition. Left: mpegenc. Right: adpcmenc (RFUOPs 1 through 7) and adpcmdec (RFUOPs 8 through 11).Instruction types shown are addition/subtraction, logical operations, shifts, branches and moves.*

We take a closer look at the internal composition of individual RFUOPS. For clarity, we restrict our attention to three applications: *mpegenc*, *adpcmenc* and *adpcmdec*. We chose these benchmarks as they contain a small number of RFUOPs. Figure 4 shows these measurements. One bar per RFUOP is shown (X-axis). Each bar is divided into sub-bars (Y-axis) representing the per instruction type breakdown of the original instructions. We split instructions into those that do addition/subtraction, bit logic operations, shifts, branches and moves. All RFUOPs are included in this graph. The actual instruction count per type is also shown (numbers within the sub-bars). It can be seen that the Chimaera compiler is capable of mapping a variety of computations. While addition/subtraction and shift operations are quite common, other types of operations are also mapped. For example, in mpegenc, a computation comprising 12 additions/subtractions and 4 branches has been mapped into a single RFUOP.

Finally, we report statistics on the number of transistor levels used by RFUOPs. For the purposes of this experiment, we hand-mapped all RFUOPs. Table 5 reports RFUOP transistor level statistics for all benchmarks. We report the average, minimum and maximum number of transistor levels per RFUOP per benchmark (three rightmost columns). The variation on the average number of levels is relatively large. The most complex operations require as many as 90 transistor levels, while the most simple ones require only 7. While these numbers may seem discouraging, it is important to also pay attention to the original instruction sequence they replace. Accordingly, we report the average, minimum and maximum number of transistor levels per RFUOP amortized over the critical path of the original instruction sequence replaced by the RFUOP. From this perspective and in the worst case, only 20 transistor levels are required per level of the original dataflow graph. While we hand-optimized the configurations shown we expect that it should be possible to generate comparable or better results using an automated method.

5.4 Performance Measurements

Figure 8 shows how performance varies over the base configuration. Note that when the RFU is included, overall issue, write-back and commit bandwidth are each still limited at 4 instructions per cycle including RFUOPs. Furthermore, only a single instance of an RFUOP can be active in the RFU at any given point in time.

It can be seen from Figure 5, part (a) that with the 2C model, Chimaera offers speedups of about 11% on the average over the 4-way base configuration. In two cases, speedups exceed 30%.

Bench	IC	Red.	Branch		Add/Sub		Logic		Shift	
	Opt.		Orig.	Opt.	Orig.	Opt.	Orig.	Opt.	Orig.	Opt.
Adpcmenc	81%	34%	27%	37%	41%	31%	10%	46%	15%	46%
Adpcmdec	53%	58%	30%	59%	29%	57%	18%	77%	14%	72%
Mpegenc	90%	12%	17%	13%	47%	19%	0%	0%	3%	31%
G721enc	94%	8%	22%	4%	41%	5%	3%	32%	12%	35%
G721dec	92%	9%	23%	5%	41%	5%	3%	32%	11%	41%
Pegwitkey	85%	22%	15%	16%	37%	33%	13%	3%	11%	67%
Pegwitenc	85%	22%	15%	16%	37%	33%	12%	2%	10%	67%
Pegwitdec	85%	22%	15%	16%	37%	33%	13%	3%	11%	67%
Honeyenc	83%	28%	13%	18%	51%	36%	1%	0%	9%	88%
Honeydec	88%	21%	13%	0%	47%	27%	0%	51%	10%	82%
Average	84%	22%	12%	18%	41%	28%	7%	25%	10%	60%

Table 4 *Global Instruction Count Statistics.*

On the other hand, we observe slowdowns in 3 benchmarks. With the C model, performance improvements almost double (about 20% on the average). Note that for *adpcmdec* the speedup under the C model is 155%. With the 3C model, performance improves only for one benchmark. As expected the 1-cycle model shows radical performance improvements for those benchmarks having RFUOPs that replaced several original instructions. Notably, even under the N model performance improves over all benchmarks. In this case, it is the decreased branches and reduced resource contention that primarily impact performance. For most programs studied the branches mapped into RFUOPs foil the GSHARE predictor. The results of this experiment suggest that in a 4-way superscalar processor and for most of the programs studied, Chimaera can offer performance improvements even if RFUOP latencies are in the order of 2C.

Figure 8, part (b) shows performance variation with the transistor-level-based models. Notably, Chimaera performs well even we assume few transistor levels per processor cycle. With the most conservative model, P12_1, we observe an improvement of 21% on the average. As shown by the P12_0 model, performance can improve by 26% o on the average in the absence of additional communication overheads. As expected, the other two models, P24_0 and P24_1, show even greater improvements, 31% in P24_0 and 29% in P24_1.

The results of this section suggest that even under relatively pessimistic assumptions about RFU latency Chimaera results in

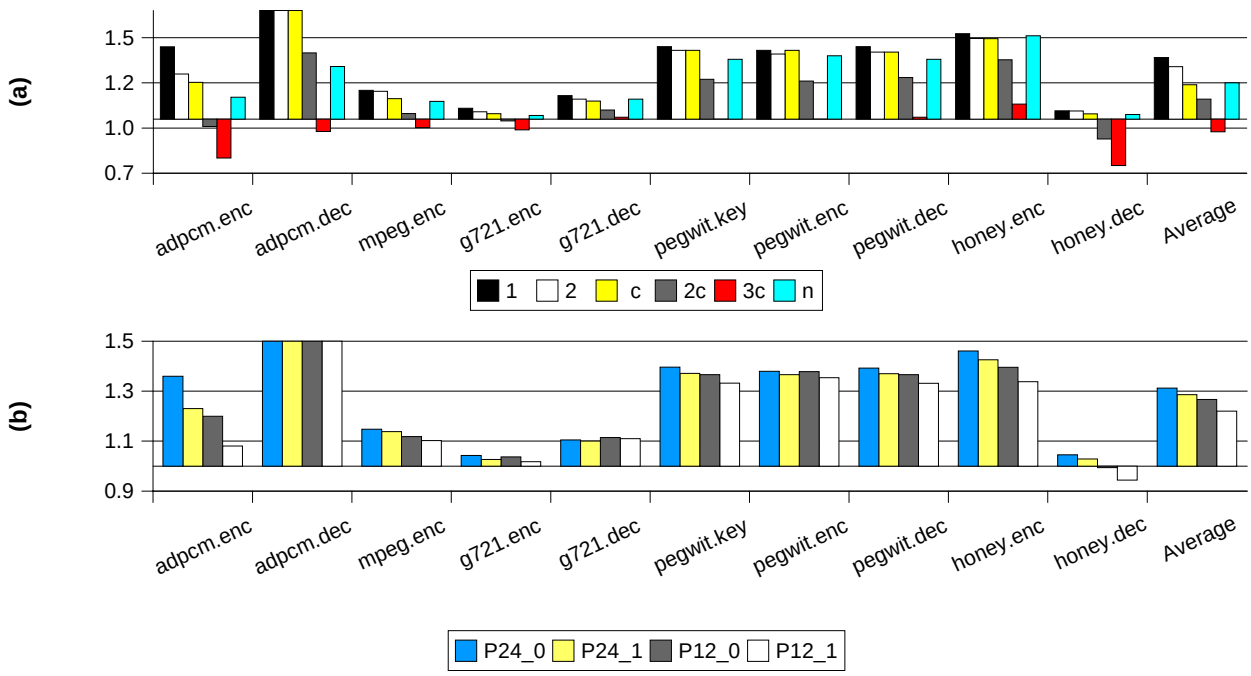

Figure 5: Chimaera Performance. *(a) Original-instruction-based timing models (The adpcm.dec bars for 1, 2 and C are truncated. The measurements are 3.52, 2.51, and 2.55). (b) Transistor-level-based timing models (The adpcm.dec bars are truncated. The measurements are 2.88, 2.97, 2.56, 2.31 from left to right).*

Benchmark	***Transistor Levels / Critical Path Inst.***			***Transistor Levels/ RFUOP***		
	Avg.	***Min***	***Max***	***Avg.***	***Min***	***Max***
adpcmenc	13.9	4	22	21.2	7	38
adpcmdec	10.4	10	12	43.5	20	96
mpegenc	11.8	10	15	64.3	40	90
g721enc	8.7	5	15	26.6	10	55
g721dec	8.5	5	15	25.5	10	55
pegwitkey	10.5	8	19	27.8	20	40
pegwitenc	10.6	8	19	28.3	20	40
pegwitdec	10.5	8	19	27.8	20	40
honeyenc	12.8	10	20	27.9	20	54
honeydec	12	10	20	29.9	19	50

Table 5: *RFUOP transistor level statistics.*

significant performance improvements over both a 4-way and a 8-way highly-aggressive superscalar processors.

6. Summary

We have described Chimaera, a micro-architecture that integrates a reconfigurable functional unit into the pipeline of an aggressive, dynamically-scheduled superscalar processor. We also described the Chimaera C compiler that automatically generates binaries for RFU execution. The Chimaera micro-architecture is capable of mapping a sequence of instructions into a single RFU operation provided that the aggregate operation reads up to 9 input registers and generates a single register output. Chimaera is also capable of eliminating control flow instructions in a way similar to that possible with predicated execution. Finally, Chimaera is capable of a more general sub-word data-parallel model than that offered by current, multimedia-oriented ISA extensions.

Using a set of multimedia and communication applications we have found the even with simple optimizations, the Chimaera C compiler is able to map 22% of all instructions on the average. A variety of computations were mapped into RFU operations, from as simple as add/sub-shift pairs to operations of more than 10 instructions including several branch statements. We studied Chimaera's performance under a number of timing models, ranging from pessimistic to optimistic. Our experiments demonstrate that for a 4-way out-of-order superscalar processor performance our approach results in average performance improvements of 21% under the most pessimistic transistor-level-based timing model (P12_1). With a different timing model (P24_1) that matches existing high-performance processor designs, Chimaera improved performance by 28% on the average. Performance varied from 5% to 197%.

Our results demonstrate the potential of the Chimaera approach, even under very pessimistic RFU latency assumptions. It is encouraging that the performance improvements were obtained using automatic compilation. While similar or higher performance improvements have been observed in multimedia applications using specialized instruction set extensions, these were in most cases the result of careful hand optimizations.

Acknowledgments

This project was supported by DARPA under Contract DABT-63-97-0035.

References

[1] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-M. W. Hwu, "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture". In Proc. 25th Intl. Symposium on Computer Architecture, 1998.

[2] D. I. August, K. M. Crozier, J. W. Sias, P. R. Eaton, Q. B. Olaniran, D. A. Connors and W.-M. Hwu, "The IMPACT EPIC 1.0 Architecture and Instruction Set Reference Manual", Technical report IMPACT-98-04, University of Illinois Urbana, IL, 1998.

[3] D. Burger, T. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set", Computer Sciences Technical Report CS-TR-96-1308, University of Wisconsin-Madison, 1996.

[4] D. Cronquist, P. Franklin, S. Berg, and C. Ebling, "Specifying and compiling applications for RaPiD", Proc. of Workshop on FPGAs for Custom Computing Machines, 1998.

[5] V. De and S. Borkar. "Low Power and High Performance Design Challenges in Future Technologies", Proc. of the 10th Great Lakes Symposium on VLSI, Mar. 2000.

[6] K. Ebcioglu and E. R. Altman, "Daisy: Dynamic Compilation for 100% Architectural Compatibility", Proc. of the 24th Intl. Symposium on Computer Architecture, Jun. 1997.

[7] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000", Journal of Research of Development, Vol. 34, No 1., Jan 1990.

[8] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. "PipeRench: A Coprocessor for Streaming Multimedia Acceleration", in Proc. Intl. Symposium on Computer Architecture, May 1999.

[9] S. Hauck, T. W. Fry, M. Hosler, and J. P. Kao. "The Chimaera Reconfigurable Functional Unit", IEEE Symposium on FPGA for Custom Computing Machines, Apr. 1997.

[10] R.W. Hartenstein, A.G. Hirschbiel, M.Weber, "The Machine Paradigm of Xputers and its Application to Digital Signal Processing Acceleration", Intl. Conference on Parallel Processing,, 1990.

[11] T. Horel and G. Lauterbach. " UltraSparc-III: Designing Third-Generation 64-Bit Performance", IEEE MICRO, May/June 1999.

[12] Honeywell Technology Center, Adaptive Computing System Benchmarking, 1998.

[13] J. R. Hauser and J. Wawrzynek. "GARP: A MIPS processor with a reconfigurable coprocessor", in Proc. of Workshop on FPGAs for Custom Computing Machines, Apr. 1997.

[14] E. Killian, "MIPS Extension for Digital Media with 3D", Microprocessor Forum, Oct. 1996.

[15] A. Klauser, A. Paithankar and D. Grunwald, Selective Eager Execution on the Polypath Architecture, in Proc. of the 25th Intl. Symposium on Computer Architecture, Jun. 1998.

[16] V. Kathail, M. S. Schlansker and B. R. Rau, HPL PlayDoh architecture specification: Version 1.0, Technical Report HPC-93-80, Hewlett-Packard Laboratories, Feb. 1994.

[17] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. "Space-Time Scheduling of Instruction-level parallelism on a Raw machine", Proc. of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, 1998.

[18] R. B. Lee, "Subword Parallelism with MAX-2", IEEE Micro vol 16(4), Aug. 1996.

[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", Proc. of the 30th Annual IEEE/ACM International Symposium on Microarchitecture, Dec. 1997.

[20] D. Levitan, T. Thomas, and P. Tu, "The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Processor", COMPCON '95: Technology for the Information Superhighway , Mar. 1995.

[21] S. Oberman, G. Favor, and F. Weber, "AMD 3Dnow! Technology: Architecture and implementations", IEEE Micro, Mar./Apr. 1999.

[22] J. Phillips and S. Vassiliadis, "High Performance 3-1 Interlock Collapsing ALU's", IEEE Transactions on Computers, Mar. 1994.

[23] A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for Multimedia PCs", Communications of the ACM, Jan. 1997.

[24] R. Razdan and M. D. Smith. "High-Performance Microarchitectures with Hardware-Programmable Functional Units," Proc. of the. 27th Intl.l Symposium on Microarchitecture, Nov. 1994

[25] R. Razdan. "PRISC: Programmable Reduced Instruction Set Computers", Ph.D. Thesis, Harvard University, Division of Applied Sciences, 1994.

[26] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H.Holt, J.Arnold and M. Gokhale, "The NAPA Adaptive Processing Architecture", IEEE Symposium on FPGAs for Custom Computing Machines, Apr. 1998.

[27] G. Sohi. Instruction issue logic for high-performance, interruptible, "multiple functional unit, pipelined computers", IEEE Transactions on Computers, Mar. 1990.

[28] Y. Sazeides, S. Vassiliadis, and J. E. Smith. "The Performance Potential of Data Dependence Speculation & Collapsing", Proc. of the 29th Intl. Symposium on Microarchitecture, Dec. 1996.

[29] M. Tremblay J. Michael O'Connor, Venkatesh Narayanan, and Liang He, "VIS Speeds New Media Processing", In IEEE MICRO, vol 16(4), Aug. 1996.

[30] R. D. Wittig and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", Proc. of Symposium on FPGAs for Custom Computing Machines, Apr. 1996.

[31] S. Wallace, B. Calder, D. Tulsen, Threaded Multipath Execution, in Proc. of the 25th Intl. Symposium on Computer Architecture, Jun. 1998.

[32] M. J. Wirthlin and B. L. Hutchings, "A Dynamic Instruction Set Computer", in Proc. Symposium on FPGAs for Custom Computing Machines, Apr. 1995.

[33] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R.Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring It All To Software:Raw Machines", IEEE Computer, Sep. 1997.

[34] N. H. E. Weste and K. Eshraghian, "Principles of CMOS VLSI Design, A Systems Perspective", Addison-Wesley Publishing Company, 1993.

[35] Z. A. Ye, N. Shenoy, and P. Banerjee. "A C Compiler for a Processor/FPGA Architecture", Proceedings of the 8th ACM Internation Symposium on Field-Programmable Gate Arrays, Feb. 2000,

[36] Z. A. Ye, A. Moshovos, S. Hauck and P. Banerjee "Chimaera: A High-Performance Architecture with a tightly-coupled Reconfigurable Unit", in Proc. 27th Intl. Symposium on Computer Architecture, Jun. 2000.