

CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit

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

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

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

Abstract

Reconfigurable hardware has the potential for significant performance improvements by providing support for application-specific operations. 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. Chimaera is capable of performing 9-input/1-output operations on integer data. We discuss the Chimaera C compiler that automatically maps computations for execution in the RFU. Chimaera is capable of: (1) collapsing a set of instructions into RFU operations, (2) converting control-flow into RFU operations, and (3) supporting a more powerful fine-grain data-parallel model than that supported by current multimedia extension instruction sets (for integer operations). 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. A variety of computations are mapped into RFU operations ranging from as simple as add/sub-shift pairs to operations of more than 10 instructions including several branches. Timing experiments demonstrate that for a 4-way out-of-order superscalar processor Chimaera results in average performance improvements of 21%, assuming a very aggressive core processor design (most pessimistic RFU latency model) and communication overheads from and to the RFU.

1 Introduction

Traditionally, instruction set architectures (ISAs) have been designed to 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 result in significant performance benefits. Unfortunately, this is easier said than done as deciding what operations to support is challenging. Such 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 in the ISA. Such instructions may become defunct as software evolves and may adversely impact future hardware implementations.

Reconfigurable hardware has the potential for providing a convenient way to address 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. Furthermore, the increasing popularity of multimedia applications provides an excellent target for reconfigurable hardware [8]. However, for this potential to materialize we need both the reconfigurable hardware and a way 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 potential advantages:

1. The RFU may reduce the execution time of dependent operations. By tailoring its datapath for specific operations, the RFU may perform several dependent operations in less

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ISCA 00 Vancouver, British Columbia Canada
Copyright (c) 2000 ACM 1-58113-287-5/00/06-225 \$5.00

time than it takes to execute each of the operations individually.

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]), 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 communicates 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). Each row contains a number of logic blocks, one per bit in the largest supported register data type (32 in our case). In Figure 2, we show the implementation of a logic

block. The logic block itself can be configured as a 4 look-up-table (LUT), two 3-LUTs, or a 3-LUT and a carry computation. 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.

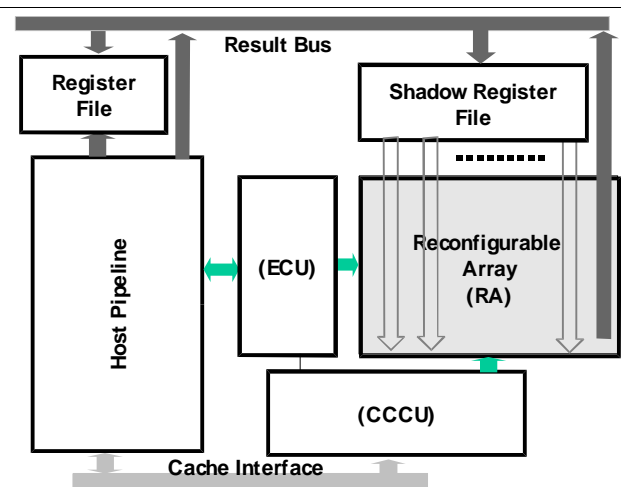


Figure 1: Overview of the Chimaera Architecture

Input data is supplied via the Shadow Register File (SRF) which is a physical, partial copy of the actual register file. 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. Physically, registers in the SRF are organized in a bit-interleaved fashion. This is because, all cells in each column of the RA have access to the corresponding bit of all registers. Which register(s) a cell accesses is determined by its configuration as we explain later on. Different cells within the array can choose which registers to access independently.

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. So long as there is sufficient space in the RA there is no need to reload an RFUOP configuration every time the corresponding RFUOP is executed. Managing the set of RFUOPs that are loaded in the RA 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. At any given point in time, multiple RFUOP configurations can be present in the RA (provided that there is enough space). We assume that a program cannot modify its own configuration data during execution. However, the CCCU snoops write traffic for writes to the configuration space. If such a write is detected, an exception is raised so that the RA can be flushed and updated.

Each RFUOP instruction is associated with a configuration and an ID. The ID serves to identify the appropriate configuration.

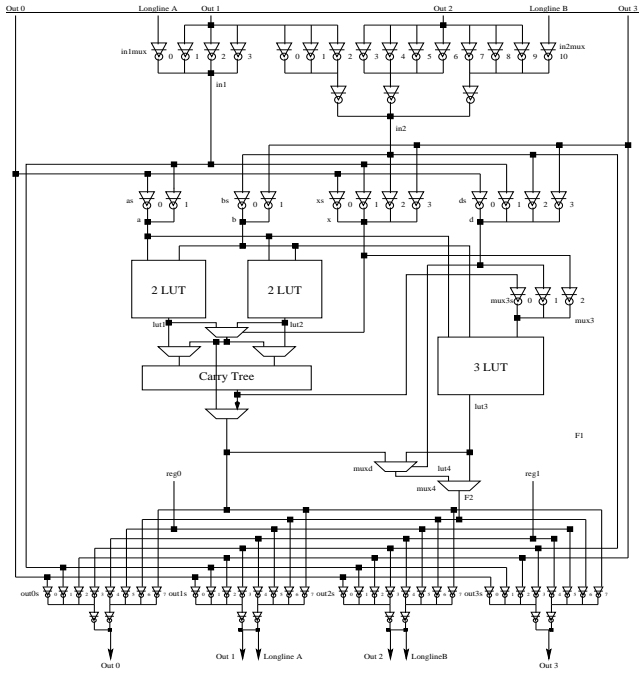


Figure 2: Logic Cell Structure

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 place. While the configuration is being loaded, execution is stalled. In our prototype implementation, each row requires 1674 bits of configuration. If the working set of RFUOPs is relatively small (which as we show is the case for the benchmarks studied), this configuration overhead can be amortized over multiple executions of the same RFUOP. Moreover, a cache of configurations is maintained to reduce configuration loading latency.

The RFUOP instruction format is shown in Figure 3(a). It consists of the RFUOP identifying opcode, of an ID, and of a destination register number. Notably, no input operand information is provided. Recall, that the configuration itself routes the appropriate input data. However, the identities of the input registers are required for out-of-order scheduling. In our implementation this information is provided in the configuration data whose layout is shown in Figure 3(b). It consists of a 32-bit vector that records the source operands, the number of rows required and the actual configuration bits. For out-of-order scheduling, the ECU maintains a record (up to 32 entries) of the input vectors for all RFUOPs currently loaded in the RA.

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 [28]. It operates as follows: 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, the ECU also allocates an entry in the RFUOP scheduler marking the location of all desired input register (this is done by maintaining a shadow register renaming table that allows single cycle access to all entries). Moreover,

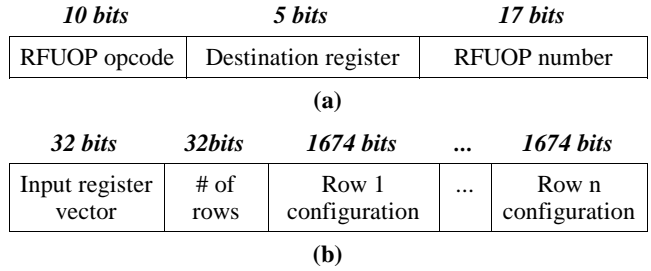


Figure 3: (a) RFUOP instruction format. (b) RFUOP configuration data layout.

the single target register of the RFUOP is renamed by the OOO core. Having marked all input dependences, and having renamed the single output register, 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. It should be noted that in an actual system, the RA will be implemented with the same technology as the processor core. 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 [35], 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. Each row is capable of implementing most single integer instructions in a typical ISA (e.g., addition, logic operations and shifts but not multiplication or loads and stores). However, logic blocks are capable of performing some complex computations. Table 1 shows the mapping and timing for a set of common 32-bit computations. The first row reports the critical path length through the RA in transistor levels. The second row shows the required number of RA rows. The third row reports the height of the equivalent dataflow graph. Finally, the fourth row shows the latency of this computation as a function of the processor’s cycle assuming an aggressive processor with only 12 transistor levels per clock cycle. It can be seen that the RFU introduces overheads when implementing a single instruction (e.g., r1 + r2). For combined operations, however, the RFU offers competitive or better latency. A detailed description of the RFUOP latency models we used is given in Section 5.1.1.

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 widely available GCC framework, version 2.6.3. We introduced the following three RFUOP-specific optimizations: *Instruction Combination*, *Control Localization*, and *SIMD Within A Register (SWAR)*. Here we provide an overview of these optimizations. More information is provided in [36].

The core optimization is Instruction Combination, which extracts RFUOPs from a sequence of instructions with no intermediate control flow. It works by analyzing the DFG and by extracting subgraphs that consist of multiple RFU-efficient

	$r1 + r2$	$r1 + r2 \ll 2$	$r1 + (r2 \& 5)$	if (r1>r2) r5=r3+r4
Critical path (transistors)	19	20	19	23
RA rows	1	1	1	2
Dataflow graph height	1	2	2	2
Latency (processor cycles)	1.58	1.67	1.58	1.96

Table 1: Critical path through the RFU’s RA for some operations assuming 12 transistor levels per processor cycle.

<pre> 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; ... } </pre> <p style="text-align: center;">(a)</p>	<pre> Adpcmdecoder() { int vpdiff, step, delta; ... vpdiff = step >> 3; vpdiff=temp1(delta, vpdiff, step); vpdiff=temp2(delta, vpdiff, step); vpdiff=temp3(delta, vpdiff, step); ... } </pre> <p style="text-align: center;">(b)</p>	<pre> Adpcmdecoder() { int vpdiff, step, delta; ... vpdiff=rfuop(delta, vpdiff, step) ... } </pre> <p style="text-align: center;">(c)</p>
---	---	--

Figure 4: 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.

nodes. RFU-efficient 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.

To increase opportunities for instruction combination, the compiler first performs the other two optimizations. Control localization transforms branch containing sequences into temporary, aggregate instructions, which can be treated as a single unit by instruction combination [18].

The SWAR optimization identifies sub-word operations that can be executed in parallel. In the current implementation, it searches for opportunities to pack several 8-bit operations into a single word operation. In most cases, this pattern exists in loops with unit stride. 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.

Figure 4 shows an example of the compilation process on the *adpcm_decoder* function which appears 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.

The loop-level systems are capable of highly-optimized implementations (e.g., a pipeline) for whole loops. GARP [13], Napa [27], PipeRench [8], Rapid [4], Xputer [10], and RAW [34] are examples of such systems. The success of this approach lies on the compiler’s ability to perform extensive loop and memory disambiguation analysis which is typically required to decide whether and how a loop can be pipelined or parallelized. Nevertheless, such systems can utilize large amounts of parallelism (coarse-grain) provided that such parallelism exists in the target application.

Instruction-level systems target fine-grain specialization opportunities. They are capable of building functional units that can implement the operation of several instructions. Chimaera [9], PRISC [25, 26], DISC [33], and OneChip [31] are instruction-level systems. Besides implementation details, Chimaera differs from other systems primarily in that it supports a 9-input/1-output instruction model.

Restricted forms of optimizations similar to those Chimaera is capable of 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. So does IBM’s Power architecture [7, 21] and several other ISAs. Phillips and Vassiliadis [23] proposed a 3-1 interlock collapsing ALU, capable of 3-input complex expressions. Sazeides, Vassiliadis, and Smith [29] analyzed the performance potential of collapsing several data-dependent operations into single-cycle equivalents.

Most current ISAs have added support for SIMD subword operations and for supporting operations tailored to multimedia applications (e.g., saturating arithmetic) [15,19,22,24,30]. Chimaera provides a more general model subword-parallelism model as the operation itself is not restricted by the ISA. Moreover, Chimaera can combine several subword operations into a single word-wide operation, even when the operations are different (e.g., two adds and two xors). Strictly speaking, this is not SIMD, rather, it is MIMD within the aggregate instruction. Finally, the number of input registers can be as large as 9.

Component	Configuration
<i>Superscalar Core</i>	
Branch predictor	<i>64k GSHARE</i>
Scheduler	<i>Out-of-order issue of up to 4 operations per cycle, 128 entry re-order buffer (RUU), 32 entry load/store queue(LSQ)</i>
Functional units	<i>4 integer ALUs, 1 integer MULT, 4 FP adders, 1 FP mult/div</i>
Functional unit latencies	<i>Integer ALU 1, integer MULT 3, integer DIV 12, FP adder 12, FP MULT 4, FP DIV 12, load/store 1</i>
Instruction cache	<i>32kb Direct-Mapped, 32-byte block, 1 cycle hit latency</i>
Data cache	<i>32kb Direct-Mapped, write-back, write-allocate, non-blocking, 32-byte blocks, 1 cycle hit latency</i>
L2 cache	<i>Unified 4-way set associative, 128k byte, 12 cycles hit latency</i>
Main memory	<i>Infinite size, 100 cycles latency</i>
Fetch Mechanism	<i>Up to 4 instructions per cycle</i>
<i>Reconfigurable Functional Unit</i>	
Scheduler	<i>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.</i>
Functional Unit / RA	<i>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.</i>
Configuration Loading	<i>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.</i>
RFUOP Latency	<i>Various model simulated. See Section 5.1.1.</i>

Table 2: Base configuration for timing experiments.

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, 17]). Internally, the RFU computes all possible paths and at the end selects only the appropriate one. A similar effect is possible with the more general multiple-path execution models [14,16, 32].

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 provide statistics on the working set of RFUOPs. This is significant, as execution has to be stalled while loading the configuration of a newly encountered RFUOP. Finally, 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 [20] and the Honeywell [12] benchmark suites. Table 3 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, a modified version of GCC version 2.6.3. We used profiling to identify candidate functions for

optimization.

For performance measurements we have used execution-driven timing simulation. We build our simulator over the widely available 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

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

Table 3: Benchmark characteristics.

used the base configuration shown in Table 2. This models an aggressive, 4-way dynamically-scheduled superscalar processor. In some experiments we also use a model of an 8-way processor which is derived for the 4-way configuration by doubling issue width (and all other appropriate stages) and instruction window size. The RFU configuration we used is also shown in Table 2. 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.

Original 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	$\lceil t/24 \rceil$	$\lceil t/24 \rceil + 1$	$\lceil t/12 \rceil$	$\lceil t/12 \rceil + 1$		

Table 4: RFUOP latency models. "c" is the critical path length of the original dataflow graph an RFUOP replaces. "n" is the number of the original instructions replaced by each RFUOP. "t" is the number of transistor levels in an RFUOP.

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 4. 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. 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. Our goal is to provide insight on the type of computations our current prototype of Chimaera is servicing. We measure the total number of instructions mapped to RFUOPs and the distribution of RFUOP sizes in terms of the number and type of original instructions they replace. We also present measurements of the critical path of the computation RFUOPs replace. Then, 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 5 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 2). 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 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.

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

Table 5: Global Instruction Count Statistics.

We next take a closer look at the computations mapped to RFUOPs. We measure their distribution in terms of the number of original instructions they replace and of the height of the original dataflow graph (i.e., critical path) they implement. These measurements are shown in Table 6 and Table 7 respectively. All measurements are weighted by the dynamic execution count of each RFUOP. Focusing on Table 6 we observe that with the exception of *mpegenc*, at least half and up

	1	2	3	4	5	6	7	...	16	17
<i>adpcmenc</i>	0%	60%	27%	0%	0%	13%	0%	0%	0%	0%
<i>adpcmdec</i>	0%	50%	0%	0%	0%	25%	0%	0%	0%	25%
<i>mpegenc</i>	0%	0%	0%	32%	0%	31%	0%	0%	37%	0%
<i>g721enc</i>	0%	52%	0%	16%	0%	32%	0%	0%	0%	0%
<i>g721dec</i>	0%	55%	0%	15%	0%	30%	0%	0%	0%	0%
<i>pegwitkey</i>	0%	57%	20%	0%	4%	0%	20%	0%	0%	0%
<i>pegwitenc</i>	0%	55%	21%	0%	2%	0%	21%	0%	0%	0%
<i>pegwitdec</i>	0%	57%	20%	0%	4%	0%	20%	0%	0%	0%
<i>honeyenc</i>	0%	70%	30%	0%	0%	0%	0%	0%	0%	0%
<i>honeydec</i>	0%	58%	33%	9%	0%	0%	0%	0%	0%	0%
Average	0%	51%	15%	7%	1%	13%	6%	0%	4%	3%

Table 6: RFUOP distribution in terms of original instruction count. Range shown is 1 to 17 instructions (columns omitted have 0% in all rows).

	1	2	3	4	5	6	7	8
<i>adpcmenc</i>	39%	47%	0%	13%	0%	0%	0%	0%
<i>adpcmdec</i>	0%	50%	0%	25%	0%	0%	0%	25%
<i>mpegenc</i>	0%	0%	0%	32%	0%	69%	0%	0%
<i>g721enc</i>	0%	68%	0%	16%	0%	16%	0%	0%
<i>g721dec</i>	0%	70%	0%	15%	0%	15%	0%	0%
<i>pegwitkey</i>	0%	57%	24%	20%	0%	0%	0%	0%
<i>pegwitenc</i>	0%	57%	23%	21%	0%	0%	0%	0%
<i>pegwitdec</i>	0%	57%	23%	20%	0%	0%	0%	0%
<i>honeyenc</i>	0%	83%	17%	0%	0%	0%	0%	0%
<i>honeydec</i>	0%	58%	36%	6%	0%	0%	0%	0%
Average	4%	55%	12%	17%	0%	10%	0%	3%

Table 7: RFUOP distribution in terms of the critical path of the original dataflow graph. Range shown is 1 to 8 instructions.

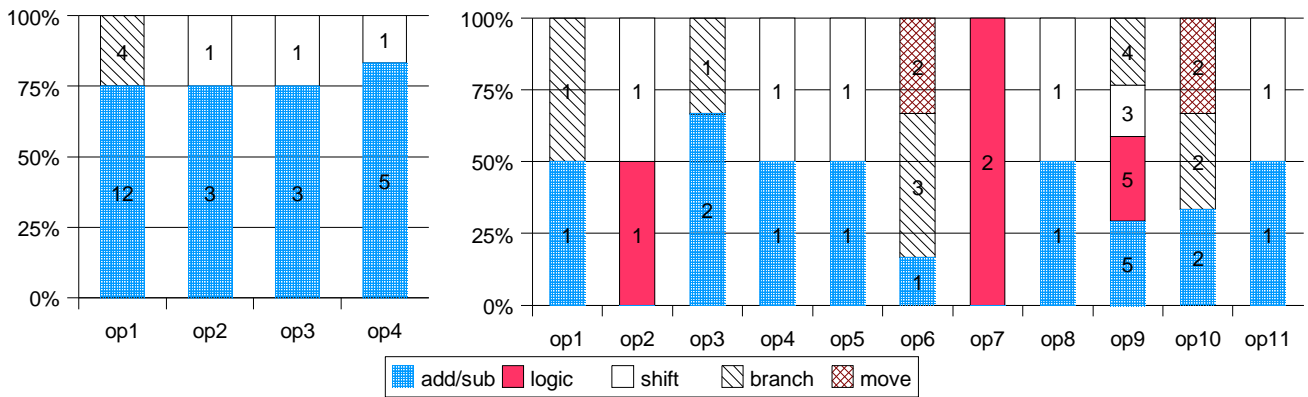


Figure 5: RFUOP instruction type composition. Left: *mpegenc*. Right: *adpcmenc* (RFUOPs 1 through 7) and *adpcmdec* (RFUOPs 8 through 11). Instruction types shown are addition/subtraction, logical operations, shifts, branches and moves.

to 81% of all executed RFUOPs replace two instructions. These usually correspond to 3-input/1-output operations of the form "dest=src1 op src2 op src3" (e.g., $y=a+b>>2$). RFUOPs that replace three original instructions are also quite common. These are either 4-input/1-output ALU-type operations (e.g., $y=a+b+c>>2$), or statements including hammock-like control-flow behavior (e.g., "if ($x>y$) $y=3$ "). In some cases, we find RFUOPs that replaced six original instructions. In most cases these correspond to two chained branch sequences. In two extreme cases, our compiler has been able to generate RFUOPs that replace as many as 17 instructions. These RFUOP operations comprise multiple branch statements and a series of operations that result in multiple-input/single-output statements.

The critical path measurements shown in Table 7 provide additional insight on the computations mapped to RFUOPs. Given enough resources, a typical out-of-order processor would be limited primarily by the computation's critical path. Most RFUOPs have a critical path of only 1 to 2 equivalent original instructions. This includes those RFUOPs that replace two original instructions (the critical path is 1 when the first instruction is a branch, or is 2 otherwise). It also includes 8% of other RFUOPs that map more than two original instructions. About 87% of all RFUOPs have a critical path of 4 equivalent instructions or less. Also, the RFUOPs in *adpcmdec* and *mpegenc* that map 16 and 17 original instructions respectively, demonstrate significantly shorter critical paths of 8 and 6

instructions respectively. (Note that some small differences between tables 5 and 6 are the result of rounding errors.)

For better understanding, 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*. Figure 5 shows these measurements. We chose these benchmarks as they contain a small number of RFUOPs. 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 Chimera 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. In *adpcmdec*, *op9* computes the equivalent of 4 branches, 3 shifts, 5 logic and 5 add/sub instructions.

Finally, we report statistics on the number of transistor levels used when RFUOPs are mapped into actual RA configurations. For the purposes of this experiment, we hand-mapped all RFUOPs. In the current RA implementation, add/sub operations are relatively slow requiring 18 transistor levels. Other operations are much more efficient requiring about 4 transistor

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 8: RFUOP transistor level statistics.

levels. Furthermore, it is possible to collapse several original instructions in much more efficient transistor level implementations. For example, an "add-shift" operation requires only 20 transistor levels. Table 8 reports RFUOP transistor level statistics for all benchmarks. For clarity, we use aggregate metrics. 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 much as 90 transistor levels, while the most simple require only 7. While these number 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.

5.3 Working Set of RFUOPs

Before we measure the performance impact of various models of the Chimaera architecture, we provide measurements on the working set of RFUOPs. This is important since execution is stalled during configuration loading. Having a large working set may adversely impact performance as it would result in thrashing in the configuration array (RA) requiring frequent accesses to memory. We measured both the working set of RFUOPs and the amount of storage required to avoid excessive accesses to memory for configuration loading. Figure 6 reports working set measurements. Shown is the miss rate of a cache that contains the last n most recently encountered RFUOPs. We vary n from 2 to 16. For most programs, maintaining a record of the last 4 most recent RFUOPs results in virtually no misses. Even in the worst case, 8 entries are sufficient. In Figure 7, we measure the miss rate assuming that each RFUOP requires K configuration rows (each of 1674 bits) where K is the number of original instructions it replaces. This is a pessimistic assumption as in an actual implementation a single row could be used to map several instructions. We simulated caches of various sizes. The range shown is 4 to 32 configuration rows (.8K to 6.7K bytes). For most programs, just 16 rows are sufficient. In few

cases, 32 rows are required. The prototype RA is capable of holding up to 32 rows simultaneously.

The results of this experiment indicate that the working set of RFUOPs is relatively small and that the amount of configuration storage in the RFU is sufficient enough to prevent thrashing.

5.4 Performance Measurements

In this section, we study how performance varies when Chimaera is introduced into an aggressive, dynamically scheduled superscalar processor. We first consider a 4-way base configuration for both the original-instruction-based timing models and the transistor-level-based models. Then we consider an 8-way base configuration.

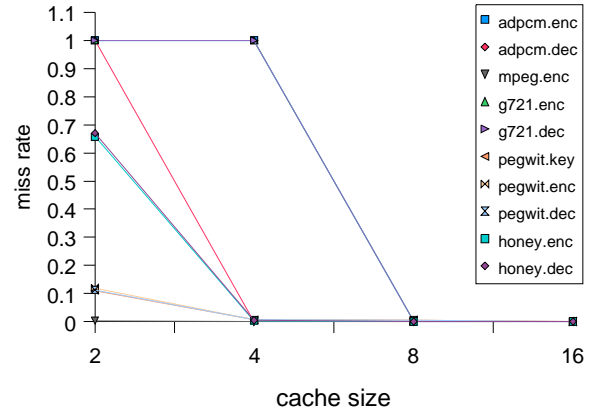


Figure 6: RFUOP working set. Cache size is the number of rfuops that can coexist in the RFU.

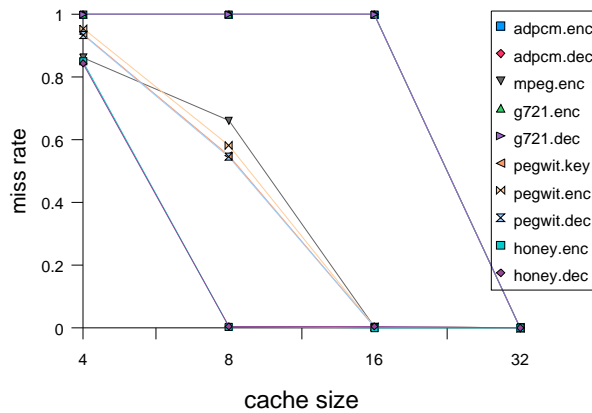


Figure 7: RFUOP configuration size working set. Cache size is the number of rows in the RFU.

Figure 8 shows how performance varies over the 4-way configuration that does not include an RFU. 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 8, 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%. 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

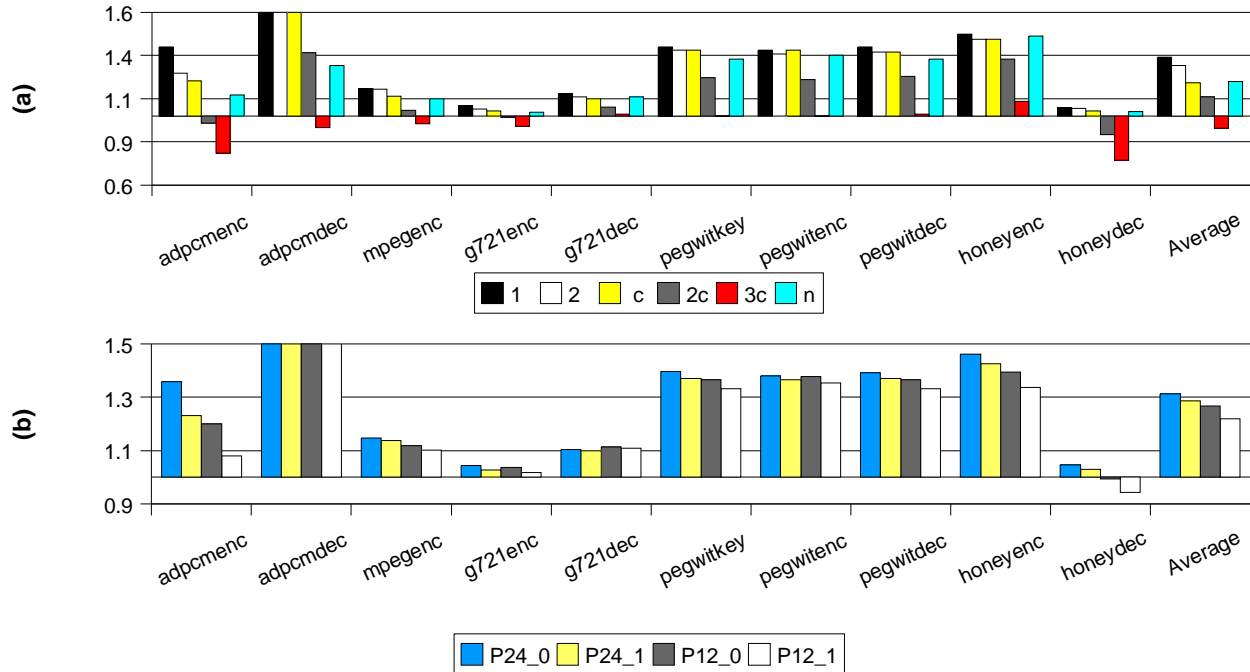


Figure 8: Relative performance over the 4-way base configuration. (a) Original-instruction-based timing models (The *adpcm.dec* bars for 1, 2 and C are truncated. The measurements are 3.52, 2.51, and 2.55). (b) Transistor-level-based timing models (The *adpcm.dec* bars are truncated. The measurements are 2.88, 2.97, 2.56, 2.31 from left to right).

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 in the context of highly-aggressive assumptions about the base processor cycle. With P12_1 which is the most conservative model, we observe an improvement of 21% on the average. As shown by the P12_0 model, performance can improve by 26% 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. Performance in these models is close to the upper bound as measured with the 1-cycle model in part (a) of Figure 8.

Figure 9 reports performance variations over the 8-way base configuration. In part (a) we report experiments with the original-instruction-based timing models, while in part (b) we report performance with the transistor-level timing models. With the original-instruction-based models performance improves only under the C, 1 and 2 and N models. Moreover, the relative improvements are smaller as compared to the 4-way processor. Under the 2C model slowdowns of as much as 25% are observed. These results suggest that in an 8-way host processor RFUOP latencies better than the 2C model are

required to improve performance. Two factors explain the relatively lower performance impact of Chimaera over the 8-way base processor configuration as compared to the 4-way configuration. First, we limit RFUOP execution into a single instance of each RFUOP at any given time (different RFUOPs can be active simultaneously). This limits the amount of inter-RFUOP parallelism that can be exploited. This impacts performance more in the 8-way configuration than it does in the 4-way configuration as the former has a twice as large instruction window than the latter. As a result, the inability to exploit inter-RFUOP parallelism for instances of the same RFUOP. Second, RFUOPs map several original instructions each of which would consume an issue slot in the base processor. As a result, a single RFUOP effectively allows us to issue its corresponding original instructions using a single issue slot. This can greatly improve performance when issue resources are limited (4-way vs. 8-way configuration).

Part (b) of Figure 9 reports performance variations with the transistor-level-based models. As it was the case for the 4-way configuration, performance improves significantly even under the most pessimistic model (P12_1). However, the performance differences among models are greater than they were under the 4-way configuration.

A strong correlation exists between performance and the fraction of branches replaced by RFUOPs. As shown in Table 5, *honeydec*, *g721enc*, and *g721dec* demonstrate the lowest fractions of reduced branches and the lowest performance improvements. In comparison, performance improves the most in *adpcmdec* which also has the largest amount of removed branches.

The results of this section suggest that even under relatively pessimistic assumptions about RFU latency Chimaera results in significant performance improvements over both a 4-way and an 8-way highly-aggressive superscalar processors.

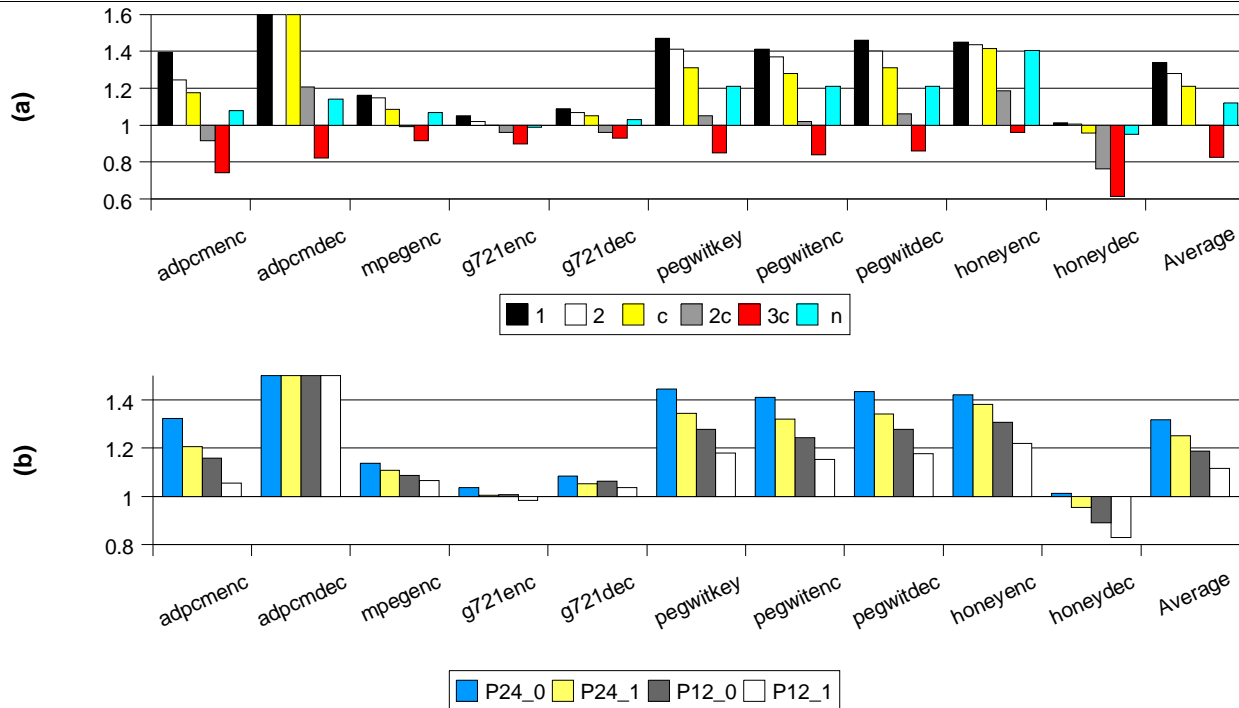


Figure 9: Relative performance over the 8-way base configuration. (a) Original-instruction-based timing models (The adpcm.dec bars for 1, 2 and C are truncated. The measurements are 4.23, 2.77 and 2.26 respectively. The 3c bar for honeydec is also truncated. The number is 0.62). (b) Transistor-level-based timing models (The adpcm.dec bars are truncated. The measurements are 3.59, 3.35, 2.26 and 2.04 from left to right).

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 also studied the performance of Chimaera under a variety of configurations. 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). The actual performance variation range was -5% to 131%. For an 8-way superscalar processor we observed speedups of 11% on the average. The actual performance variation range was -19% to 104%. With a different timing model (P24_1) that matches existing high-performance processor designs, Chimaera

improved performance by 28% and 25% on the average over the 4-way and 8-way base configurations respectively. Performance varied from 5% to 197% and from -5% to 235% respectively.

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.

7. Acknowledgments

We would like to thank the reviewers for their helpful comments. 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," Proceedings of the 25th Annual International 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, Feb. 1998.
- [3] Doug Burger, Todd 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", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, 1998.
- [5] Vivek De and Shekhar Borkar, "Low Power and High Performance Design Challenges in Future Technologies", Proceedings of the 10th Great Lakes Symposium on VLSI, Mar. 2000.
- [6] K. Ebcioglu and E. R. Altman, "Daisy: Dynamic Compilation for 100% Architectural Compatibility", Proceedings of the 24th Annual IEEE/ACM International 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", Proceedings of the 26th Annual ACM/IEEE International 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, and M. Weber, "The Machine Paradigm of Xputers and its Application to Digital Signal Processing Acceleration", International Conference on Parallel Processing, 1990.
- [11] Tim Horel and Gary Lauterbach, "UltraSparc-III: Designing Third-Generation 64-Bit Performance", IEEE MICRO, May/June 1999.
- [12] Honeywell technology center, Adaptive Computing System Benchmarking, <http://www.htc.honeywell.com/projects/acsbench/>, 1998.
- [13] John R. Hauser and John Wawrzyniek, "GARP: A MIPS processor with a reconfigurable coprocessor", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Apr. 1997.
- [14] T. Heil and J. Smith, "Selected Dual Path Execution", University of Madison - Wisconsin Technical Report, May 1997.
- [15] E. Killian, "MIPS Extension for Digital Media with 3D", Microprocessor Forum, Oct. 1996.
- [16] A. Klauser, A. Paithankar, and D. Grunwald, "Selective Eager Execution on the Polypath Architecture", Proceedings of the 25th Annual International Symposium on Computer Architecture, Jun. 1998.
- [17] 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.
- [18] 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", Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1998.
- [19] R. B. Lee, "Subword Parallelism with MAX-2", IEEE Micro vol 16(4), Aug. 1996.
- [20] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture, Dec. 1997.
- [21] 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.
- [22] Stuart Oberman, Greg Favor, and Fred Weber, "AMD 3Dnow! Technology: Architecture and implementations", IEEE MICRO, Mar./Apr. 1999.
- [23] J. Phillips and S. Vassiliadis, "High Performance 3-1 Interlock Collapsing ALU's", IEEE Transactions on Computers, Mar. 1994.
- [24] A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for Multimedia PCs", Communications of the ACM, Jan. 1997.
- [25] R. Razdan and M. D. Smith, "High-Performance Microarchitectures with Hardware-Programmable Functional Units," Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture, Nov. 1994.
- [26] R. Razdan, "PRISC: Programmable Reduced Instruction Set Computers", Ph.D. Thesis, Harvard University, Division of Applied Sciences, 1994.
- [27] 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.
- [28] G. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers", IEEE Transactions on Computers, Mar. 1990.
- [29] Yiannakis Sazeides, S. Vassiliadis, and James E. Smith. "The Performance Potential of Data Dependence Speculation & Collapsing", Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, Dec. 1996.
- [30] M. Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He, "VIS Speeds New Media Processing", In IEEE MICRO, vol 16(4), Aug. 1996.
- [31] R. D. Wittig and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, Apr. 1996.
- [32] S. Wallace, B. Calder, and D. Tulsen, "Threaded Multipath Execution", Proceedings of the 25th Annual IEEE/ACM International Symposium on Computer Architecture, Dec. 1998.
- [33] Michael J. Wirthlin and Brad L. Hutchings, "A Dynamic Instruction Set Computer", IEEE Symposium on FPGAs for Custom Computing Machines, Apr. 1995.
- [34] 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.
- [35] N. H. E. Weste and K. Eshraghian, "Principles of CMOS VLSI Design, A Systems Perspective", Addison-Wesley Publishing Company, 1993.
- [36] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee, "A C Compiler for a Processor/FPGA Architecture", Proceedings of the 8th ACM International Symposium on Field-Programmable Gate Arrays, Feb. 2000.