The Effect of Reconfigurable Units in Superscalar Processors

Jorge E. Carrillo E.
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, ON, CANADA
jece@eecg.toronto.edu

Paul Chow
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, ON, CANADA
pc@eecg.toronto.edu

ABSTRACT

This paper describes OneChip, a third generation reconfigurable processor architecture that integrates a Reconfigurable Functional Unit (RFU) into a superscalar Reduced Instruction Set Computer (RISC) processor's pipeline. The architecture allows dynamic scheduling and dynamic reconfiguration. It also provides support for pre-loading configurations and for Least Recently Used (LRU) configuration management.

To evaluate the performance of the OneChip architecture, several off-the-shelf software applications were compiled and executed on Sim-OneChip, an architecture simulator for OneChip that includes a software environment for programming the system. The architecture is compared to a similar one but without dynamic scheduling and without an RFU. OneChip achieves a performance improvement and shows a speedup range from 2.16 up to 32 for the different applications and data sizes used. The results show that dynamic scheduling helps performance the most on average, and that the RFU will always improve performance the best when most of the execution is in the RFU.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures, Pipeline processors*

General Terms

Experimentation, Performance

Keywords

OneChip, reconfigurable processors, superscalar processors

1. INTRODUCTION

Recently, the idea of using reconfigurable resources along with a conventional processor has led to research in the area of reconfigurable computing. The main goal is to take advantage of the capabilities and features of both resources. While the processor takes care of all the general-purpose computation, the reconfigurable hardware acts as a specialized coprocessor that takes care of specialized applications. With such platforms, specific properties of applications, such as parallelism, regularity of computation, and data granularity can be exploited by creating custom operators, pipelines, and interconnection pathways.

There has been research done in the Department of Electrical and Computer Engineering at the University of Toronto on such reconfigurable processors, namely, the OneChip processor model has been developed. At first, this model tightly integrated reconfigurable logic resources and memory into a fixed-logic processor core. By using the reconfigurable units of this architecture, the execution time of specialized applications was reduced. The model was mapped into the Transmogrifier-1 field-programmable system. This work was done by Ralph Wittig [22].

A follow-on model, called OneChip-98, then integrated a memory-consistent interface. It is a hardware implementation that allows the processor and the reconfigurable array to operate concurrently. It also provides a scheme for specifying reconfigurable instructions that are suitable for typical programming models. This model was partially mapped into the Transmogrifier-2 field-programmable system. This work was done by Jeff Jacob [11].

OneChip's architecture has now been extended to a superscalar processor that allows multiple instructions to issue simultaneously and perform out-of-order execution. This leads to much better performance, since the processor and the reconfigurable logic can execute several instructions in parallel. Most of the performance improvement that this architecture shows comes from memory streaming applications, that is, those applications that read in a block of data from memory, perform some computation on it, and write it back to memory. Multimedia applications have this characteristic and are used to evaluate the architecture.

Previous subsets of the OneChip architecture[1] have been modeled by implementing them in hardware. The purpose of this work is to properly determine the feasibility of the architecture by building a full software model capable of simulating the execution of real applications.

[1]We will use the term *OneChip* from now on to refer to the latest version of the OneChip architecture.

FPGA 2001, February 11-13, 2001, Monterey, CA, USA.

1.1 Related Work

In general, a system that combines a general-purpose processor with reconfigurable logic is known as a Field-Programmable Custom Computing Machine (FCCM). Research on FCCMs done by other groups [2, 5, 7, 14, 16, 18, 19] has reported speedup obtained by combining these two techniques, however, most of the research in these groups is focused on aspects of the reconfigurable fabric and the compilation system. Much of the OneChip work is focused toward the interface between the two technologies. As a result, the applications are modified by hand; no modification was done to the compiler; and our simulations model only the functionality and latency of the reconfigurable fabric, not the specifics of the fabric architecture.

In our work, we study the effect of combining reconfigurability with an advanced technique to speedup processors, a superscalar pipeline, by focusing on the interplay between them. With the use of out-of-order issue and execution, one can further exploit instruction-level parallelism in applications, without incurring the overheads involved in reconfiguring a specialized hardware. Previously, performance reports by other groups were done using application kernels such as the DCT, FIR filters, or some small kernel-oriented applications. Only recently have some groups [2, 18, 23] reported on the performance using complete applications, which give more meaningful results. In this work, we are focused on the architecture's performance with full applications.

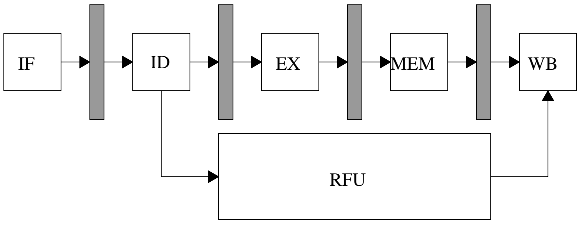

Figure 1: OneChip's Pipeline

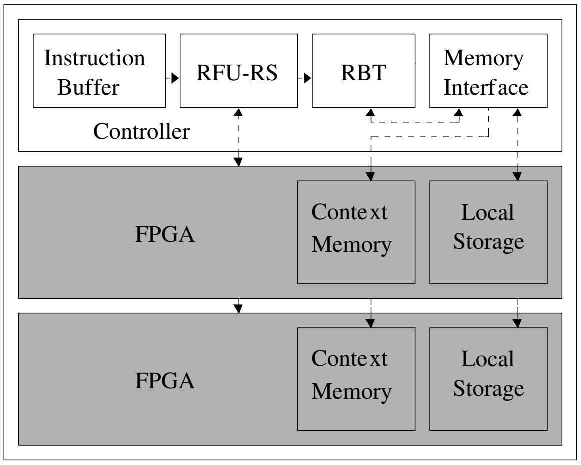

Figure 2: RFU Architecture

2. ONECHIP ARCHITECTURE

In this section we give a brief overview of the OneChip architecture, including the more recently added features.

The processor's main features, as proposed in [22, 11] are:

- MIPS-like RISC architecture — simple instruction encoding and pipelining.
- Dynamic scheduling — allows out-of-order issue and completion.
- Dynamic reconfiguration — can be reconfigured at run-time.
- Reconfigurable Functional Unit (RFU) integration — programmable logic in the processor's pipeline.

In addition, OneChip has now been extended to include:

- a Superscalar pipeline — allows multiple instructions to issue per cycle.
- Configuration pre-loading support — allows loading configurations ahead of time.
- Configuration compression support — reduces configuration size.
- LRU configuration management support — reduces number of reconfigurations.

2.1 Processor pipeline

The original OneChip pipeline described in [11] is based on the DLX RISC processor described by Hennessy & Patterson [9]. It consists of five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM) and Writeback (WB). A diagram of the pipeline is shown in Figure 1. The RFU is integrated in parallel with the EX and MEM stages. It performs computations as the EX stage does and has direct access to memory as the MEM stage does. The RFU contains structures such as the Memory Interface, an Instruction Buffer, a Reconfiguration Bits Table (RBT) and Reservation Stations.

OneChip is now capable of executing multiple instructions in parallel. The EX stage consists of multiple functional units of different types, such as integer units, floating point units and a reconfigurable unit. Due to the flexibility of the reconfigurable unit to implement a custom instruction, a programmer or a compiler can generate a configuration for the reconfigurable unit to be internally pipelined, parallelized or both. Dynamic scheduling of RFU instructions is implemented in OneChip. Data dependencies between RFU and CPU instructions are handled using RFU Reservation Stations.

2.2 RFU Architecture

The RFU in OneChip contains one or more FPGAs and an FPGA Controller as shown in Figure 2. The FPGAs have multiple contexts and are capable of holding more than one configuration for the programmable logic [4]. These configurations are stored in the Context Memory, which makes the FPGA capable of rapidly switching among configurations. Each context of the FPGAs is configured independently from the others and acts as a cache for configurations. Only one context may be active at any given time.

Instructions that target the RFU in OneChip are forwarded to the FPGA Controller, which contains the reservation stations and a Reconfiguration Bits Table (RBT).

The FPGA Controller is responsible for programming the FPGAs, the context switching and selecting configurations to be replaced when necessary. The FPGA Controller also contains a buffer for instructions and the memory interface. The RBT acts as the configuration manager that will keep track of where the FPGA configurations are located. The memory interface in the FPGA Controller consists of a DMA controller that is responsible for transferring configurations from memory into the context memory according to the values in the RBT. It also transfers the data that an FPGA will operate on into the local storage. The local storage may be considered as the FPGA data cache memory. The multiple FPGAs in the RFU share the same FPGA Controller and each FPGA has its own context memory and local storage.

OneChip has been enhanced to support configuration compression and reduce the overhead involved in configuring the FPGA. An algorithm for compressing configurations is proposed by Hauck et al. [8]. This feature has not been modeled in our simulator for these results since the internal architecture of the FPGA fabric is not yet defined, therefore the actual size of the configuration bitstreams is unknown. Futhermore, our benchmarks only use one configuration and the effect of the overhead can be easily managed by preloading the configuration.

The architecture has also been extended to support configuration management. Although the FPGAs can hold multiple configurations, there is a hardware limit on the number of configurations it can hold. OneChip uses the Least Recently Used (LRU) algorithm as a mechanism for swapping configurations in and out of the FPGA. LRU is implemented in OneChip by using a table of configuration reference bits. The approach is similar to the Additional-Reference-Bits Algorithm described by Silberschatz & Galvin in [20]. A fixed-width shift register is used to keep track of each loaded configuration's history. On every context switch, all shift registers are shifted 1 bit to the right. On the high-order bit of each register, a 0 is placed for all inactive configurations and a 1 for the active one. If the shift register contains 00000000, it means that it hasn't been used in a long time. If it contains 10101010, it means that it has been used every other context switch. A configuration with a history register value of 01010000 has been used more recently than another with the value of 00101010, and this later one was used more recently than one with a value of 00000100. Therefore, the configuration that should be selected for replacement is the one that has the smallest value in the history register. Notice that the overall behavior of these registers is to keep track of the location of configurations in a queue, where a recently used configuration will come to the front and the last one will be the one to be replaced. Our simulator does not have this feature implemented at this time as it was not required in the benchmarks.

The Reconfiguration Bits Table (RBT) acts as the configuration manager that will keep track of where the FPGA configurations are located. The information in this table includes the address of each configuration and flags that keep track of loaded and active configurations. The RBT described in [11] has been enhanced to support the algorithm for configuration management [3]. The history of each configuration is also stored in the table to allow LRU configuration management and to select configurations to be replaced.

Table 1: Memory Consistency Scheme

Hazard Number	Hazard Type	Actions Taken
1	RFU rd after CPU wr	1. Flush RFU source addresses from CPU cache when instruction issues. 2. Prevent RFU reads while pending CPU store instructions are outstanding.
2	CPU rd after RFU wr	3. Invalidate RFU destination addresses in CPU cache when RFU instruction issues. 4. Prevent CPU reads from RFU destination addresses until RFU writes its destination block.
3	RFU wr after CPU rd	5. Prevent RFU writes while pending CPU load instructions are outstanding.
4	CPU wr after RFU rd	6. Prevent CPU writes to RFU source addresses until RFU reads its source block.
5	RFU wr after CPU wr	7. Prevent RFU writes while pending CPU store instructions are outstanding.
6	CPU wr after RFU wr	8. Prevent CPU writes to RFU destination addresses until RFU writes its destination block.
7	RFU rd after RFU wr	9. Prevent RFU reads from locked RFU destination addresses.
8	RFU wr after RFU rd	10. Prevent RFU writes to locked RFU source addresses.
9	RFU wr after RFU wr	11. Prevent RFU writes to locked RFU destination addresses

2.3 Instruction specification

OneChip is designed to obtain speedup mainly from memory streaming applications in the same way vector coprocessors do. In general, RFU instructions take a block of data that is stored in memory, perform a custom operation on the data and store it back to memory.

Previously, OneChip supported only a two-operand RFU instruction. To have more flexibility for a wider range of applications, it has now been extended to support a three-operand RFU instruction. In the two-operand instruction, one can specify the opcode, the FPGA function, one source and one destination register that hold the respective memory addresses, and the block sizes. In this instruction, the source and destination block sizes can be different. In the three-operand instruction, one of the block sizes is replaced by another source register. This allows the RFU to get source data from two different memory locations, which need not be continuous. In this instruction, all three blocks should be the same size.

In OneChip, there are two configuration instructions. One of them is the *Configure Address* instruction, which is used for assigning memory addresses in the RBT. The other configuration instruction is the *Pre-load* instruction, which is used for pre-fetching instructions into the FPGA and reducing configuration overhead. Some compiler prefetching techniques have been previously published for other reconfigurable systems [6, 21].

2.4 Memory controller

OneChip allows superscalar dynamic scheduling, hence instructions with different latencies may be executed in parallel. The RFU in OneChip has direct access to memory and is also allowed to execute in parallel with the CPU. When there are no data dependencies between the RFU and the CPU, the system will act as a multiprocessor system, providing speed up. However, when data dependencies exist between them, there is a potential for memory inconsistency that must be prevented.

The memory consistency scheme previously proposed for OneChip, as described in [11], allows parallel execution between one FPGA and the CPU. The scheme has now been extended to support more than one FPGA in the RFU. The nine possible hazards that OneChip may experience along with the actions taken to prevent them, are listed in Table 1. This scheme preserves memory consistency when the CPU and an FPGA, or when two or more FPGAs, are allowed to execute concurrently.

OneChip implements the memory consistency scheme by using a Block Lock Table (BLT). The BLT is a structure that contains four fields for each entry and locks memory blocks to prevent undesired accesses. The information stored in the table includes the *block address*, *block size*, *instruction tag* and a *src/dst* flag.

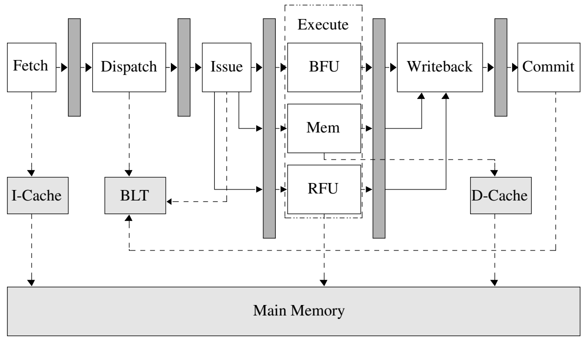

Figure 3: Sim-OneChip's Pipeline

3. SIM-ONECHIP

This section will describe the implementation of *sim-onechip*, the simulator that models the architecture of OneChip. It is a functional, execution-driven simulator derived from *sim-outorder* from the simplescalar tool set [1].

To model the behaviour of OneChip, we needed an already existing simulator capable of doing out-of-order execution and that was easily cutomizable to be used a basis to add OneChip's features. Two existing architecture simulators [1, 17] were considered for modification and SimpleScalar was the chosen platform. Besides being a complete set of tools, the annotations capability was an attractive feature, since it would allow the addition of new instructions in a very simple manner.

3.1 Modifications to sim-outorder

Modifications were done to *sim-outorder* to model OneChip's reservation stations, reconfiguration bits table, block lock table and the reconfigurable unit. The overall functionality of *sim-outorder* was preserved.

The reservation stations for Sim-OneChip were implemented as a queue. Besides the already existing scheduler queues for Basic Functional Unit (BFU) instructions and for Memory (MEM) instructions, a third scheduler queue was implemented to hold RFU instructions. This queue is referred to as the Reconfigurable Instructions Queue (RecQ). The dispatch stage detects instructions that target the RFU and places them in the RecQ for future issuing.

The RBT is implemented as a linked list. The RBT models the FGPA controller by performing dynamic reconfiguration and configuration management. Functions are provided for assigning configuration addresses, loading configurations and to perform context switching.

The BLT is implemented as a linked list. Each entry holds the fields for the two sources and the destination memory blocks for each RFU instruction. It ensures the OneChip memory consistency scheme by modeling the actions taken for each of the hazards presented. By keeping track of the memory locations currently blocked, conflicting instructions are properly stalled.

The RFU was included with the rest of the functional units and in the resource pool in the functional unit resource configuration.

3.2 Pipeline description

To be able to adapt OneChip to the SimpleScalar architecture, several modifications were done to the original pipeline. Sim-OneChip's pipeline, as in *sim-outorder*, consists of six stages: *fetch*, *dispatch*, *issue*, *execute*, *writeback* and *commit*. This section will describe the modifications done to each stage in *sim-outorder* and the places where each of OneChip's structures were included. Sim-OneChip's pipeline is shown in Figure 3.

The *fetch* stage remained unmodified and fetches instructions from the I-cache into the dispatch queue.

The *dispatch* stage decodes instructions and performs register renaming. It moves instructions from the dispatch queue into the reservation stations in the three scheduler queues: the *Register Update Unit* (RUU), the *Load Store Queue* (LSQ) and the *Reconfigurable Instructions Queue* (RecQ). This stage adds entries in the BLT to lock memory blocks when RFU instructions are dispatched.

The *issue* stage identifies ready instructions from the scheduler queues (RUU, LSQ and RecQ) and allows them to proceed in the pipeline. This stage also checks the BLT to keep memory consistency and stalls the corresponding instructions.

The *execute* stage is where instructions are executed in the corresponding functional units. Completed instructions are scheduled on the event queue as writeback events. This stage is divided into three parallel stages: BFU stage, MEM stage and RFU stage. The BFU stage is where all operations that require basic functional units, such as integer and floating point are executed; the MEM stage is where all memory access operations are executed and has access to the D-cache, and; the RFU stage is where RFU instructions are executed.

The *writeback* stage remained unmodified and moves completed operation results from the functional units to the RUU. Dependency chains of completing instructions are also scanned to wake up any dependent instructions.

The *commit* stage retires instructions in-order and frees up the resources used by the instructions. It commits the results of completed instructions in the RUU to the register file and stores in the LSQ will commit their result data to the data cache. This stage clears BLT entries to remove memory locks once the corresponding RFU instruction is committed.

The BLT is accessed by the dispatch, issue and commit stages. The memory consistency scheme requires that instructions are entered in the BLT and removed from it in program order. In the pipeline, the issue, execute and writeback stages do not necessarily follow program order since out-of order issue, execution and completion is allowed. Hence, memory block locks and the corresponding entries in the BLT need to be entered when an RFU instruction is dispatched, since dispatching is done in program order. Like-

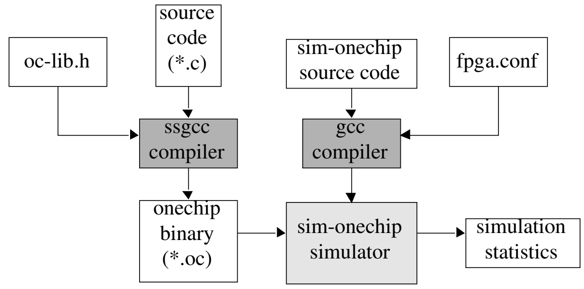

Figure 4: Sim-OneChip's Simulation Process

wise, entries from the BLT need to be removed when RFU instructions commit, since committing is also performed in program order.

All actions in the memory consistency scheme are taken in the issue stage. The issue stage is allowed to probe the BLT for memory locks. Instructions that conflict with locked memory blocks are prevented from issuing at this point. All others are allowed to proceed provided there are no dependencies.

3.3 RFU instructions

Annotation of instructions in SimpleScalar are useful for creating new instructions. They are attached to the opcode in assembly files for the assembler to translate them and append them in the annotation field of assembled instructions.

Taking advantage of this feature, new instructions can be created without the need to modify the assembler. OneChip's RFU instructions will be disguised as already existing, but annotated instructions that the simulator will recognize as an RFU instruction and model the corresponding operation. Without the annotation, instructions are treated as regular ones; with the annotation they become instructions that target the reconfigurable unit.

The four instructions defined for OneChip (i.e. two RFU operation instructions and two configuration instructions) were created for Sim-OneChip. Macros are used to translate from a C specification to the corresponding annotated assembly instruction.

3.4 Programming model

Currently, the programming model for OneChip is the use of circuit libraries. Programming for Sim-OneChip is done in C. The user may use existing configurations from a library of configurations, or create custom ones. Configurations are defined in C and several macros are available for accessing memory or instruction fields.

The complete simulation process is shown in Figure 4. A C program that includes calls to RFU instructions is compiled by the simplescalar gcc compiler *ssgcc* along with the OneChip Library `oc-lib.h`. This will produce a binary file that can be executed by the simulator *sim-onechip*. All the program configurations specified in `fpga.conf` must be previously compiled by gcc along with the simulator source code to produce the simulator. Once both binaries are ready, the simulator can simulate the execution of the binary and produce the corresponding statistics.

Sim-OneChip's processor specification can be defined as command-line arguments. One can specify the processor core parameters, such as fetch and decode bandwidth, internal queues sizes and number of execution units. The memory hierarchy and the branch predictor can also be modified.

3.4.1 OneChip library

The library defines the following five macros:

oc_configAddress(func, addr) is used for specifying the configuration address for a specified function. It will associate the function `func` with the address `addr` where the FPGA configuration bits will be taken from and will enter the corresponding entry in the BLT.

oc_preLoad(func) is used for pre-loading the configuration associated with the specified function `func` into an available FPGA context.

rec_2addr(func, src_addr, dst_addr, src_size, dst_size) is the two-operand reconfigurable instruction. `func` is the FPGA function number, `src_addr` and `dst_addr` are the source and destination block addresses, `src_size` and `dst_size` are source and destination block sizes encoded.

rec_3addr(func, src1_addr, src2_addr, dst_addr, blk_size) is the three-operand reconfigurable instruction. `func` is the FPGA function number, `src1_addr`, `src2_addr` and `dst_addr` are the source-1, source-2 and destination block addresses, and `blk_size` is the block size encoded.

Both reconfigurable instructions will perform the context switch to activate function `func` and will execute the corresponding operation associated with it. They will also lock their respective source and destination blocks of memory by entering the corresponding fields in the BLT for as long as the function takes to execute. When finished, the BLT entries corresponding to the instruction will be cleared.

oc_encodeSize(size) is a macro used for encoding the size of memory blocks. It obtains the encoded value from a table that is defined by the function $log_2(size) - 1$. This macro should be used to encode block sizes in reconfigurable instructions.

For example,

```
rec_3addr(2,&a,&b,&c,oc_encodeSize(16));
```

where a, b and c are defined as

```
unsigned char a[16], b[16], c[16];
```

will activate function 2 and perform the operation with arrays `a` and `b` as source data and array `c` as destination data. The encoded size passed to the reconfigurable instruction will be $log_2(16) - 1 = 3$.

3.4.2 Configuration definition

The behavior of the RFU is modeled with a high-level functional simulation. It is given some inputs, and a function produces the corresponding outputs without performing a detailed micro-architecture simulation of the programmable logic.

Configurations are defined as follows,

```
DEFCONF(<addr>, <oplat>, <issuelat>,
        {
          <EXPR>
        }
)
```

where

`<addr>`	Configuration address (i.e. the location of the configuration bits in memory).

`<oplat>`	Operation latency (i.e. the number of cycles until result is ready for use).
`<issuelat>`	Issue latency (i.e. the number of cycles before another operation can be issued on the same resource).
`<EXPR>`	Expression that describes the reconfigurable function.

The separation of the instruction latency into operation and issue latencies, allows the specification of pipelined configurations. For example, assume one configuration takes 20 cycles to complete one instruction, but the configuration is pipelined and one instruction can be started every 4 cycles. In this case, the operation latency will be 20 and the issue latency will be 4. Hence, the configuration can have $\frac{20}{4} = 5$ executing instructions at a time and the throughput for the configuration is implied as $\frac{20}{5} = 4$ cycles per instruction.

The expression field is where the semantics of the configuration will be specified. It is a C expression that implements the configuration being defined, the expression must modify all the processor state affected by the instruction's execution.

All memory accesses in the DEFCONF() expression must be done through the memory interface. There are macros available for doing memory reads and writes; for accessing general purpose registers, floating point and miscellaneous registers; for accessing the value of the RFU instruction operand field values, and; for creating a block mask and decoding a block size. Some configuration examples are included in [3].

4. PROGRAMMING FOR SIM-ONECHIP

This section will present an example of how to port an application so that it uses the RFU in OneChip to get speedup. The application to be implemented is an 8-tap FIR filter.

Consider that you have this C code in a file called `fir.c`.

```
 1: /* FILE: fir.c */
 2:
 3: #include <stdio.h>
 4:
 5: #define TAPS 8
 6: #define MAX_INPUTS 1024
 7:
 8: int coef[TAPS] = {1,2,3,4,5,6,7,8};
 9: int inputs[MAX_INPUTS];
10:
11: void main(){
12:   int i, j;
13:   int *x;
14:   int y[MAX_INPUTS];
15:
16:   /* Set the inputs to some random numbers */
17:   for (i = 0; i < MAX_INPUTS; i++){
18:     inputs[i] = 3 * (i % 1000) - MAX_INPUTS % 123;
19:     y[i] = 0;
20:   }
21:   x = inputs;
22:
23:   /* FIR Filter kernel */
24:   for (i = 0; i < MAX_INPUTS-TAPS; i++){
25:     for (j = 0; j < TAPS; j++){
26:       y[i] += coef[j] * x[j];
27:     }
28:     x++;
29:   }
30:
31:   printf("\nFIR filter done!\n");
32: }
```

The inner loop in the FIR filter kernel on lines 25–27 of `fir.c` can be ported to be executed entirely on the OneChip RFU. For that, we need to do some modifications to the C code. The file `fir.oc.c` that reflects this changes is shown below.

```
 1: /* FILE: fir.oc.c */
 2:
 3: #include <stdio.h>
 4: #include "oc-lib.h"
 5:
 6: #define TAPS 8
 7: #define MAX_INPUTS 1024
 8:
 9: int coef[TAPS] = {1,2,3,4,5,6,7,8};
10: int inputs[MAX_INPUTS];
11:
12: void main(){
13:   int i, j;
14:   int *x;
15:   int y[MAX_INPUTS];
16:
17:   oc_configAddress(0, 0x7FFFC000);
18:   oc_preLoad(0);
19:
20:   /* Set the inputs to some random numbers */
21:   for (i = 0; i < MAX_INPUTS; i++){
22:     inputs[i] = 3 * (i % 1000) - MAX_INPUTS % 123;
23:     y[i] = 0;
24:   }
25:   x = inputs;
26:
27:   /* FIR Filter kernel */
28:   for (i = 0; i < MAX_INPUTS-TAPS; i++){
29:     rec_3addr(0, x, coef, &y[i], oc_encodeSize(8));
30:     x++;
31:   }
32:
33:   printf("\nFIR filter done!\n");
34: }
```

The first step was including the OneChip library in the code as shown on line 4 in `fir.oc.c`. The second step was defining the address of the configuration bitstream for the FIR filter. In this case, we are using configuration #0 and the memory address is `0x7FFFC000`, as shown on line 17. As a third step, notice that lines 25–27 on `fir.c` have been removed and replaced by a 3-operand RFU instruction in line 29 on `fir.oc.c`. This instruction is using configuration #0 and is passing the address of the two source memory blocks, `x` and `coef`, which are pointers, as well as the address of destination memory block, which for each iteration will be `&y[i]`. The block size, 8, is passed using the function `oc_encodeSize`.

The previous three changes are necessary. Furthermore, if we want to reduce configuration overhead, we would introduce a pre-load instruction as in line 18. This instruction tells the processor that configuration #0 will be used soon. This way, by the time it gets to execute the RFU instruction, the configuration is already loaded and no time is spent waiting for the configuration to be loaded. This instruction is not necessary, because if the configuration is not loaded in the FPGA, the processor will automatically load it.

Now that the C code has been modified to use the RFU, we need to define the FPGA configuration that will perform the FIR filter. Configurations are defined in `fpga.conf`. The fir filter definition used is shown below.

```
1: /* This configuration is for a 3-operand instruction.
2:    It is used for a fir filter program. */
3:
4: DEFCONF(0x7FFFC000, 17, 17,
5:  {
6:   int oc_index;             /* for indexing */
7:   unsigned int oc_word;     /* for storing words */
8:   unsigned int oc_result;   /* for storing result */
9:
10:   oc_result = 0;
11:   for(oc_index = 0;
12:       oc_index <= OC_MASK(OC_3A_BS);
13:       oc_index++)
14:    {
15:     oc_word = READ_WORD(GPR(OC_3A_S1R)+(4*oc_index));
16:     oc_word*= READ_WORD(GPR(OC_3A_S2R)+(4*oc_index));
17:     oc_result += oc_word;
18:    }
19:   WRITE_WORD(oc_result, GPR(OC_3A_DR));
20:  }
21: )
```

This configuration is the equivalent of the inner loop in the FIR filter kernel on lines 25–27 in `fir.c`. Note that in the configuration, each memory access is done through the memory interface. Line 4 defines the configuration address `0x7FFFC000` and the operation and issue latencies of 17. Lines 11–13 define the iteration loop for the FIR filter. Line 15 reads a word from the memory location defined by the address stored in the general purpose register that contains one source address plus the corresponding memory offset. In the same way, line 16 reads a word from the other source block and multiplies it with the data previously read and stored in the `oc_word variable`. Line 17 simply accumulates the multiplied values across loop iterations. When the loop is finished, line 19 writes the result into the memory location defined by the address stored in the general purpose register that has the destination block address.

The simulator will generate statistics for the number of instructions executed in each program. The speedup obtained with Sim-OneChip can be verified.

5. APPLICATIONS

To evaluate the performance of the OneChip architecture, several benchmark applications were compiled and executed on Sim-OneChip.

5.1 Experimental Setup

To do the experiments, four steps were performed for each application. Step one is the identification of which parts of each application are suitable for implementation in hardware. Step two is modeling the hardware implementation of the identified parts of the code. Step three is the replacement of the identified code in the application, with the corresponding hardware function call. And step four is the execution and verification of both the original and the ported versions of the application.

The pipeline configuration used for both simulations was the default used in SimpleScalar. Among the most relevant characteristics are an instruction fetch queue size of 4 instructions; instruction decode, issue and commit bandwidths of 4 instructions per cycle; a 16-entry register update unit (RUU) and an 8-entry load/store queue (LSQ). The number of execution units available in the pipeline are 4 integer ALU's, 1 integer multiplier/divider, 2 memory system ports available (to CPU), 4 floating-point ALU's, 1 floating-point multiplier/divider. Also, in the case of sim-onechip, 1 reconfigurable functional unit (RFU), an 8-entry RBT and a 32-entry BLT were used. The branch predictor and cache configuration remained unmodified as well.

5.2 Benchmark applications

There is currently no standard benchmark suite for reconfigurable processors. C. Lee et al.[13] from the University of California at Los Angeles have proposed a set of benchmarks for evaluating multimedia and communication systems, which is called MediaBench. Since current reconfigurable processors available are used mostly for communications applications, MediaBench was taken as the suite for evaluating OneChip. Not all of the applications were used for the evaluation. Some of them could not be ported to SimpleScalar, due to the complexity of the makefiles or due to some missing libraries. However, the rest of the applications can provide good feedback on the architecture's performance.

5.3 Profiles

Profiling the execution of an application helps to identify the parts of the application take a lot of time to execute and hence, being candidates for rewriting to make it execute faster. Profiling of the applications was performed using GNU's profiler *gprof* included in GNU's *binutils 2.9.1* package.

From the profiling information, we identified specific functions in each application that are worth improving by executing them in specialized hardware implemented in the OneChip reconfigurable unit. To port an application to OneChip, a piece of code must have a long execution time and perform memory accesses in a regular manner, as in applications suitable for vector processors. In general, any application that can be sped up by a vector processor, will be also suitable for OneChip.

5.4 Analysis and modifications

Four applications met our requirements and were ported to OneChip [3]. JPEG Image compression, ADPCM Audio coding, PEGWIT Data encryption and MPEG2 Video encoding. The encoder and the decoder for each one was ported. The modifications to the applications are done by hand (i.e. no compiler technologies are used). For the RFU timing in each of the applications, we assume that memory accesses dominate the computational logic and that our bottleneck is the memory bandwidth. If we also assume that one memory access is perfomed in one cycle, the latency of an operation will be obtained from counting the total number of memory accesses performed by the operation. This timing approach may not be precise for highly compute intensive operations, but it is not the case on these applications.

6. RESULTS

The original and the modified versions of the eight chosen applications were executed on the simulator. Each application was tested with three different sizes of data, one small, one medium and one large. Four experiments were done for each application. Our first experiment was executing the original applications with *in-order* issue **(A)** to verify how many cycles each one takes to execute. As a second experiment, we executed the OneChip version of each

Table 2: Speedup

Application	Data size	Onechip inorder (A/B)	OneChip outorder (C/D)	Outorder original (A/C)	Outorder OneChip (B/D)	Total (A/D)
JPEG encode	Small	1.37X	1.34X	2.29X	2.25X	3.08X
	Medium	1.36X	1.33X	2.29X	2.24X	3.05X
	Large	1.38X	1.35X	2.33X	2.29X	3.15X
JPEG decode	Small	1.29X	1.20X	2.47X	2.29X	2.96X
	Medium	1.29X	1.19X	2.52X	2.34X	3.01X
	Large	1.25X	1.16X	2.53X	2.35X	2.93X
ADPCM encode	Small	22.38X	17.04X	1.54X	1.18X	26.31X
	Medium	26.25X	17.85X	1.62X	1.10X	28.94X
	Large	29.92X	20.57X	1.56X	1.07X	32.02X
ADPCM decode	Small	18.32X	13.47X	1.60X	1.18X	21.55X
	Medium	21.79X	14.81X	1.62X	1.10X	24.02X
	Large	24.43X	16.27X	1.61X	1.07X	26.13X
PEGWIT encrypt	Small	1.46X	1.43X	2.09X	2.06X	3.00X
	Medium	1.33X	1.36X	2.20X	2.26X	3.00X
	Large	1.16X	1.24X	2.48X	2.65X	3.07X
PEGWIT decrypt	Small	1.40X	1.42X	2.08X	2.11X	2.95X
	Medium	1.28X	1.32X	2.27X	2.35X	3.00X
	Large	1.13X	1.18X	2.62X	2.72X	3.08X
MPEG2 decode	Small	4.69X	5.44X	2.02X	2.34X	11.00X
	Medium	5.07X	5.70X	2.07X	2.33X	11.82X
	Large	5.23X	5.91X	2.08X	2.36X	12.33X
MPEG2 encode	Small	1.16X	1.14X	1.90X	1.87X	2.16X
	Medium	1.30X	1.26X	1.86X	1.81X	2.34X
	Large	1.28X	1.24X	1.87X	1.81X	2.32X

application also with *in-order* issue **(B)**. The third experiment was executing the original version of the applications with *out-of-order* issue **(C)**. And the fourth and last experiment was executing again the OneChip version, but now with *out-of-order* issue **(D)**. This way we could verify the speedup obtained by using both features, the reconfigurable unit and the *out-of order* issue, in the OneChip pipeline. All output files were verified to have the correct data after being encoded and decoded with the simulator.

The speedup obtained from the experiments is shown in Table 2. The first column **(A/B)** shows the speedup obtained by only using the reconfigurable unit. The second column **(C/D)** shows the speedup obtained by introducing a reconfigurable unit to an out-of-order issue pipeline. The third column **(A/C)** shows the speedup obtained by only using out-of-order issue. The fourth column **(B/D)** shows the speedup obtained by introducing out-of-order issue to a pipeline with a reconfigurable unit as OneChip. The fifth column **(A/D)** shows the total speedup obtained by using the reconfigurable unit and out-of order issue at the same time.

Further analyzing the simulation statistics, we note that there are no BLT instruction stalls (i.e. instructions stalled due to to memory locks) in the applications, except for JPEG. This means that either the RFU is fast enough to keep up with the program execution or there are no memory accesses performed in the proximity of the RFU instruction execution. The second one is the actual case. It is important not to confuse BLT stalls, which prevent data hazards, with stalls due to unavailable resources, which are structural hazards. If there are two consecutive RFU instructions with no reads or writes in between, there will most likely be a structural hazard. Since there is only one RFU, the trailing RFU instruction will be stalled until the RFU is available. This is not considered a BLT stall.

In the case of JPEG, there are CPU reads and writes performed in the proximity of RFU instructions. These are shown in Table 3. *RFU instructions* shows the total dynamic count of RFU instructions in the program, *BLT instruction stalls* is the number of CPU reads and writes stalled after an RFU write is executing (this was the only type of hazard present). The next column shows the *Stalls per RFU instruction* and the last one shows the average RFU instruction overlap with CPU execution. Note that 128 is the operation latency for JPEG. We can see that for the JPEG encoder there is an overlap of approximately 11 instructions, and for the JPEG decoder an overlap of approximately 5 instructions.

Table 3: JPEG RFU instructions

Application	Data size	RFU instructions (X)	BLT instruction stalls (Y)	Stalls per RFU instruction (Y/X)	RFU Overlapping (128 - Y/X)
JPEG encode	Small	851	99531	116.96	11.04
	Medium	5720	669204	116.99	11.01
	Large	18432	2156508	117.00	11.00
JPEG decode	Small	851	104422	122.71	5.29
	Medium	5720	702466	122.81	5.19
	Large	18432	2264375	122.85	5.15

6.1 Discussion

In Table 3 we can see that there is an approximate overlap of 11 instructions for the JPEG decoder. This means that when an RFU instruction is issued, 11 following instructions are also allowed to issue out-of-order because there are no data dependencies. Then, even if the RFU issue and operation latencies are improved (i.e. reduced) by new hardware technologies, the maximum improvement for this application will be observed if the configuration has a latency of 11 cycles. That is, any latency lower than 11 will not improve performance because the other 11 overlapping instructions will still need to be executed and the RFU will need to wait for them. The same will be observed for the JPEG decoder with a latency of 5 cycles. For the rest of the applications there is no overlap, so any improvement in the RFU latency will be reflected in the overall performance.

ADPCM shows a fairly large speedup from OneChip. This is because the application does not perform any data validation or other operations besides calling the encoder kernel. The data is simply read from standard input, encoded on blocks of 1000 bytes at a time, and written to standard output, so the behaviour of the application is more like that of a kernel. It is expected that ADPCM is the application with the most speedup due to Amdahl's Law [9], which states that the performance improvement to be gained from using a faster mode of execution is limited by the fraction of time the faster mode can be used. ADPCM's performance clearly depends on the size of the data. The larger the data, the less time the application reads and writes data, and most of the time the RFU executes instructions. There are no BLT instruction stalls in this application.

PEGWIT's performance also shows a dependence on the data size. However, different behavior is observed for the RFU and the out-of-order issue features. The RFU shows better performance with small data, while out-of-order issue shows a better performance improvement with larger data. The overall speedup with both features is greater as the input data is larger. This is because for the decoder, the application makes a number of RFU instruction calls independent of the data size, and even if the data size for each call is different, the latency is the same for every call.

With the encoder, almost the same thing happens. No BLT instruction stalls were originated in this application.

MPEG2's performance is also data size dependent. In the case of the decoder, the larger the data, the greater the performance improvement. This applies for both, the RFU feature and the out-of-order issue feature. In the case of the encoder, the performance improvement is shown to be larger, as the frame sizes get larger. There is a a higher performance improvement between the tests with small and medium input data, which have a different frame size and almost the same number of frames, than between the tests with medium and large data, which have the same frame size and different number of frames. In the application, there are no BLT instruction stalls.

For all the applications, we can see that out-of-order issue by itself produces a big gain **(A/C)**. Using an RFU still adds more speedup to the application. Speedup obtained from dynamic scheduling ranges from 1.60 up to 2.53. Speedup obtained from an RFU **(A/B)**ranges from 1.13 up to 29. When using both at the same time, even when each technique limits the potential gain that the other can produce, the overall speedup is increased. Dynamic scheduling seems to be more effective with the applications, except for ADPCM, where the biggest gain comes from using the RFU. This leads us to think that for kernel-oriented applications, it is better to use an RFU without the complexity of out-of-order issue and for the other applications it is better to use dynamic scheduling possibly augmented with an RFU.

7. CONCLUSION AND FUTURE WORK

In this work, the behavior of the OneChip architecture model was studied. Its performance was measured by executing several off-the-shelf software applications on a software model of the system. The results obtained confirm the performance improvement by the architecture on DSP-type applications.

From the work, a question arises whether the additional hardware cost of a complex structure, such as the Block Lock Table, is really necessary in reconfigurable processors. It has been shown that the concept of the BLT does accomplish its purpose, which is maintaining memory consistency when closely linking reconfigurable logic with memory and when parallel execution is desired between the CPU and the PFU. However, considering that only one of the four applications (i.e. JPEG) used in this research actually uses the BLT and takes advantage of it, we conclude that by removing it and simply making the CPU stall when any memory access occurs while the RFU is executing, will not degrade performance significantly on the types of benchmarks studied. In JPEG there is an average of 11 overlapping instructions, which is only 8.6% of the configuration operation latency slot of 128. If the RFU is used approximately 20% of the time in the JPEG encoder, the performance improvement by the overlapping is only 1.72%. This is a small amount compared to the performance improvement of dynamic scheduling, which is approximately 56% (i.e. 2.29 speedup). Hence, dynamic scheduling improves performance significantly only when used with relatively short operation delay instructions, as opposed to OneChip's RFU instructions, which have large operation delays.

Based on the four applications in this work, it appears to be that the number of contexts does not need to be large to achieve good performance improvement with an RFU. In these applications, only one context was used for each application and a considerable speedup was obtained. For some applications, a second context could have provided an increase in this improvement, but not as much as for the first context. This is because, based on our profiles, we could implement a different routine in a second context in the same way the first one was, but it would not be so frequently used.

Another question that arises from this work is whether the configurations are small enough to fit on today's reconfigurable hardware, or if they can be even implemented. Hardware implementations of DSP structures done in our and other groups [22, 11, 12], and which have even been shown to outperform digital signal processors, have been proven to fit on existing Altera and Xilinx devices with a maximum of 36,000 logic gates. Today's FPGAs have more than 1 million system gates available.

To estimate the silicon area of this version of OneChip, we can start with the area of the processor that is required. It will be much larger than the simple processor used in the previous version of OneChip. A similar processor is the MIPS R10000 processor core [15], which is a 4-way superscalar processor that supports out-of-order execution and includes a 32KB instruction cache and a 32KB data cache. Using a CMOS 0.35μm process, the die area is approximately 297mm^2. We can estimate that as fabrication technology approaches a 0.13μm process, the size of the processor core would be approximately 41mm^2. The OneChip-98 processor [10] includes a small processor core of insignificant area, an eight-context FPGA structure with about 85K gates of logic, and 8 MBytes of SRAM. In a 0.18μm process, this was estimated to take about 550mm^2. Scaling to 0.13μm brings this to 287mm^2 to which we can add the 41mm^2 for the processor. The complete OneChip device would be about 328mm^2, which is quite manufacturable. Obviously, it would be desirable to add more gates of FPGA logic and as the process technology continues to shrink, this would be easy to do.

We also conclude that dynamic scheduling is important to achieve good performance. By itself it produces a big gain for a number of applications. With kernel oriented applications, the gain obtained by an RFU is bigger, but with complete applications, the biggest gain is obtained from out-of-order issue and execution.

Further investigation is necessary in the area of compilers for reconfigurable processors. Specifically, a compiler designed for the OneChip architecture is needed to fully exploit it and to better estimate the advantages and disadvantages of the architecture's features. Developing a compilation system that allows automatic detection of structures suitable for the OneChip RFU, as well as generating the corresponding configuration and replacing the structure in the program, will allow further investigation of the optimal number of contexts for the RFU. The compiler should be able to pre-load configurations to reduce delays, and should also make an optimal use of the BLT by scheduling as many instructions in the RFU delay slot.

Also, future work should investigate the architecture of the RFU. In our work we have assumed an optimal RFU. No work has been done on what type of logic blocks or interconnection resources should be used in the FPGA. The simulator should be extended to properly simulate the FPGA fabric and any configuration latency issues. Ye et. al. [23] have modeled RFU execution latencies using simple instruction-level and transistor-level models. However, their architecture target fine-grain instructions, while OneChip targets coarse-grain instructions.

At this point, it becomes difficult to make a detailed comparison between OneChip's performance and other current reconfigurable systems. This is because there are no standard application benchmarks available for reconfigurable processors. However, other groups have reported performance improvement results similar to the ones presented in this paper, using Mediabench and SPEC benchmarks [2, 23]. Although Onechip shares certain similarities with other systems [2, 19] that target memory-streaming applications and focus on loop-level code optimizations, a standardized set of benchmarks and metrics for reconfigurable processors is needed to properly evaluate the differences between them.

8. ACKNOWLEDGEMENTS

We would like to akcnowlegde Chameleon Systems Inc. for financially supporting the OneChip project. Jorge Carrillo was also supported by a UofT Open Fellowship. We would also like to thank the reviewers for their helpful comments.

9. REFERENCES

[1] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison, Computer Sciences Department, 1997.

[2] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, Apr. 2000.

[3] J. E. Carrillo Esparza. Evaluation of the OneChip reconfigurable processor. Master's thesis, University of Toronto, 2000.

[4] A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In *Proceedings IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 31–39, Apr. 1994.

[5] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Roe, and R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, Apr. 2000.

[6] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 65–74, Feb. 1998.

[7] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 87–96, Apr. 1997.

[8] S. Hauck, Z. Li, and E. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 123–130, Apr. 1998.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, second edition edition, 1996.

[10] J. A. Jacob. Memory interfacing for the OneChip reconfigurable processor. Master's thesis, University of Toronto, 1998.

[11] J. A. Jacob and P. Chow. Memory interfacing and instruction specification for reconfigurable processors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 145–154, Feb. 1999.

[12] D. Lau, A. Schneider, M. D. Ercegovac, and J. Villasenor. FPGA-based structures for on-line FFT and DCT. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 310–311, Apr. 1999.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Procedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO 30)*, pages 330–335, Dec. 1997.

[14] G. Lu, H. Singh, M.-H. Lee, N. Bagherzadeh, F. Kurdahi, and E. M. C. Filho. The MorphoSys parallel reconfigurable system. In *Procedings of Euro-Par 99, Toulouse, France*, Sept. 1999.

[15] MIPS Technologies, Incorporated. *MIPS R10000 (T5) Superscalar Microprocessor, technical brief*, Oct. 1994.

[16] T. Miyamori and K. Olukotun. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 2–11, Apr. 1998.

[17] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, Feb. 1997.

[18] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, Nov. 1994.

[19] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. A. Arnold, and M. Gokhale. The NAPA adaptive processing architecture. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 28–37, Apr. 1998.

[20] A. Silberschatz and P. B. Galvin. *Operating System Concepts.* Addison Wesley Longman, Inc., USA, fifth edition edition, 1998.

[21] X. Tang, M. Aalsma, and R. Jou. A compiler directed approach to hiding configuration latency in chameleon processors. In *Proceedings of the the 10th International Conference on Field-Programmable Logic and Applications*, Aug. 2000.

[22] R. D. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 126–135, Mar. 1996.

[23] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable unit. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 225–235, June 2000.