Hardware-Software Co-simulation of Bus-based Reconfigurable Systems

Vikram K.N [*], V. Vasudevan

Department of Electrical Engineering, IIT Madras, Chennai – 600036, India

Abstract

One of the most flexible and modular approaches to reconfigurable systems is a bus-based approach. In order to get realistic performance estimates of these systems, detailed modeling of the processor as well as the bus and memory hierarchy is required. In addition, when coupling one or more reconfigurable units with a superscalar, out-of-order issue, load/store RISC CPU using the on-chip system bus, there are issues relating to cache coherency that need to be addressed. We have developed a cycle accurate co-simulator that uses a 'C' model of the processor and HDL models of the bus and reconfigurable units. We have also made modifications to the CPU pipeline to allow for non-cacheable accesses to the reconfigurable unit. This is reported in the paper. We have used this simulator to look at (a) The speedup obtained for two examples, namely, matrix multiplication and Lempel-Ziv compression, (b) The speedup obtained when there is a context switch from one application to the other and full reconfiguration is employed and (c) Speedup obtained with partial reconfiguration. These results are reported in the paper.

Key words: Bus-based, SoC, Embedded FPGA, Reconfiguration, Co-simulation

1 Introduction

Present day computing involves a variety of applications, which run different types of computation kernels. The computing machine therefore, needs to be as flexible as possible. The most flexible solution is to use a general purpose processor (GPP) in the system. GPPs have the flexibility in software, which can be changed by the user, for different applications. While GPPs

[*] Corresponding author
Email addresses: `vikramkn@ee.iitm.ernet.in` (Vikram K.N), `vinita@ee.iitm.ernet.in` (V. Vasudevan).

serve well with their inherent flexibility, the computation time incurred by them for many applications is very large. To overcome this shortcoming, the trend has been to extend the microprocessor architecture with reconfigurable hardware [1]. The literature reports several ways of coupling reconfigurable hardware to a processor [2]. OneChip [3] and Chimaera [4] incorporate the reconfigurable unit (RU) as a functional unit (FU) in the processor pipeline. They have custom instructions that are used for configuring the RU and starting RU execution. Pipeline interlocking schemes are employed to take care of data dependencies. The RU gets data either from the processor register file or directly from memory. Garp [5], REMARC [6], Morphosys [7] and Zippy [8] have the RU as co-processor. The co-processor interface provides a low latency, high bandwidth connection between the processor and the RU. Even in this case, custom instructions are required to be added to the CPU. The role of custom instructions here is to either send an opcode to the co-processor or to write data/control information to the co-processor. Some co-processors accept an opcode, decode it and execute the instruction [6,7]. In this case, pipeline interlocking has to be done. Others co-processors perform execution based on some control information written by the processor, into a co-processor register [5,8]. Here, either pipeline interlocking is done [5] or the user code explicitly polls a co-processor register to determine if execution has finished [8]. The co-processor either gets data from the processor, or it fetches data by itself, from the data cache or directly from memory.

Another model for coupling the RU to the CPU, is the bus-based approach. In the bus-based approach, the RU does not have direct access to the processor. No custom instructions are added to the processor, to control the RU. The RU is instead controlled over a bus, which until recently has been only on the same board as the CPU. The bus is used to connect the various components of the system, to the CPU. Traditionally, due to bandwidth and latency constraints imposed by the bus, attaching RUs on the bus has been feasible only if the computation to memory bandwidth ratio is high. So, the attached RUs are restricted to a small class of applications that satisfy this requirement. An example is stream-based multimedia processing, tackled by PipeRench [9].

Nowadays, with decreasing transistor size and increasing logic density on silicon, the CPU, hardware accelerators, peripherals and system memory exist on the same die, giving rise to the so-called “System-on-a-Chip” (SoC). In SoCs, the bus resides on-chip, alongwith the processor and other components. This bus does not have the bandwidth and latency constraints seen traditionally in the on-board bus. The bus-based approach for attaching RUs is the most flexible and modular approach for SoC systems. The processor resides as a master on the bus and the RU is attached to the bus as master or as a slave. All the RUs reside on a single, partially reconfigurable fabric (embedded FPGA or e-FPGA [10]). The RU interface to the bus is fixed, irrespective of the processor. In addition, there is no limitation on the number of RUs attached to the bus, other than the limitation imposed by the bus decoder and area of e-FPGA. This allows the number of active RUs sitting on the bus to be varied dynamically, by partial reconfiguration of the e-FPGA. Borgatti et al [10] have recently developed a reconfigurable system, featuring an e-FPGA, interfaced to the processor using the on-chip AMBA AHB [11] bus. Bondalapati and Prasanna [12] cite two recent commercial examples of RU coupled to the CPU using the bus-based approach. One is the Chameleon Systems reconfigurable processor. In the Chameleon architecture, a 32-bit ARC processor is linked to different components of the chip by the on-chip RoadRunner bus. The other example is that from Xilinx [13]. Xilinx has released the Virtex-II Pro FPGA, which houses an embedded PowerPC 405 core. The processor block is compliant with CoreConnect bus architecture. Like AMBA, this bus architecture comprises of a high performance bus and a peripheral bus.

In order to get realistic estimates of the performance of bus-based systems, we need a cycle accurate simulator along with detailed models of the bus/memory hierarchy. This is because, the estimation of cycle count in the bus based system is difficult since it depends on the number of modules attached to the bus. Also, if bus accesses are cached by the bus masters, the accesses cannot be accurately modeled at a high level of abstraction, due to the very nature of unpredictability introduced by caching. Several cycle accurate simulators have been developed for coprocessor/ functional unit based reconfigurable systems [3,8,4,5,7], but we have not seen detailed simulations of bus-based systems. A SystemC model of a bus-based reconfigurable system has been reported [14], but simulation has been done at a relatively high level with no detailed models of bus/memory hierarchy.

Besides the bus/memory hierararchy, there are also issues relating to the coupling of one or more RUs to a superscalar out-of-order processor that need to be addressed. Since the RUs are attached to the bus, they are memory mapped and all accesses from the processor to the slave will be cached. This is not desirable, since the slave can change the contents of the addressed locations, giving rise to the "stale data" problem that occurs when there are I/O systems or in multiprocessor systems [15]. The problem can be resolved either by using the OS or by adding extra hardware. Using the OS results in a performance penalty and adding extra hardware is not desirable.

In this paper we have addressed these two issues. To study the performance of bus based systems, we have developed a cycle accurate hardware-software co-simulator with detailed models of the bus/memory hierarchy. The CPU is modeled in 'C' and the bus and RUs are modeled in HDL. The cycle-accurate MASE simulator [16] has been modified and used to simulate the CPU. The AMBA [11] is a widely used open specification [17] bus for SoC systems. A synthesizable VHDL model is available for the AMBA bus. This was used to model the bus. The RUs can be modeled either in C or in HDL, within our framework. We model the RUs in HDL, since this allows us to easily attach off-the-shelf HDL models or IP cores to the bus. To avoid the "stale data" problem, the address range corresponding to the RUs is marked non-cacheable(NC). Since the processor is a superscalar out-of-order CPU, issues related to the out-of-order execution property arise. These problems and the modifications required in the CPU pipeline are detailed in this paper.

Using the simulator, we have looked at the performance of two benchmarks -

matrix multiplication and Lempel-Ziv (LZ77) compression. In order to make it worthwhile to have dynamic reconfigurability, we need to look at the speedup that can be obtained with one/both RUs on the bus. Assuming that the configuration data is pre-loaded, we look at the speedup that can be obtained for (a) matrix multiplication with different sizes for the matrices and (b) LZ77 compression for different input data sizes. These results are reported. To study the performance of dynamically reconfigurable systems, we have looked at two cases - (a) The base configuration is LZ77 and the eFPGA is completely reconfigured to do matrix multiplication. (b) The base configuration is LZ77 and the e-FPGA is partially reconfigured to do matrix multiplication using a smaller RU. These results are also reported. Case (a) corresponds to the class of *multi-context* dynamically reconfigurable systems [18] whereas case (b) corresponds to *partially reconfigurable* [19] systems. The time required for reconfiguration is obtained using the configuration data sizes for Xilinx Virtex XCV-1000.

This paper is organized as follows. In Section 2, the processor model, bus model, processor interaction with the RU and the reconfiguration model are described. This section also details the issues involved in interfacing the processor to the bus. Section 3 describes the co-simulation environment, along with some details of the hardware-software synchronization. A brief review of various approaches for hardware-software co-simulation is also included. Section 4 presents simulation results and Section 5 concludes this paper.

2 System model

The system model is diagramatically depicted in Figure 1. It comprises of a superscalar CPU core as the bus master, a memory controller and multiple RUs interfaced as slaves on the bus. All the slaves on the bus are memory-mapped. The interaction between the CPU and the RU is through control and status registers. In addition, we have assumed that each RU has registers/memory to hold the data required for computation and the results of the computation. All these registers are memory mapped. In this section, we describe the various components of our system.

2.1 The processor model

Cycle-accurate ‘C’ simulators for processors are available: RSIM [20], SimpleScalar [21], MASE [16] etc. SimpleScalar has been used in the past for simulations of co-processor/FU based systems [3,4,8]. However, we have chosen MASE to model the processor. It is an extensible, parameterizable processor simulation tool suite that simulates a superscalar, out-of-order issue CPU with a Programmable Instruction Set Architecture (PISA), resembling MIPS. It supports multilevel, parameterizable caches. Compared to SimpleScalar, MASE has provisions to accomodate unknown/variable memory access la-

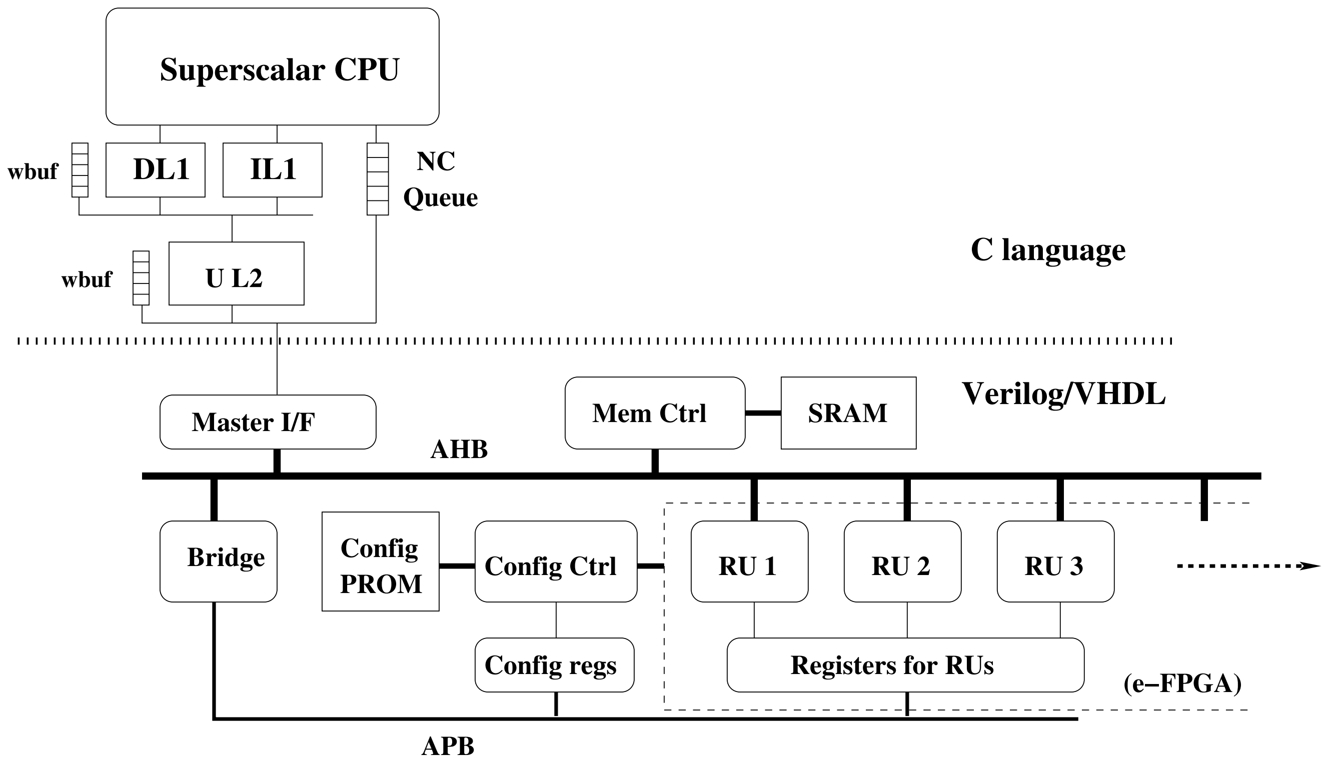

Fig. 1. System model of our simulation environment

tency. This makes it more amenable to interfacing it to a bus, since bus access times are not fixed. Moreover, in MASE, the pipeline operates on live data, rather than doing an in-order execution with just modeling the out-of-order execution latency. This feature is important for simulating non-cacheable accesses to the bus. Although both these features are present in RSIM, MASE also allows adding new instructions by way of annotation of existing instructions, without modifications to the compiler/assembler. This is an attractive feature, since reconfigurable systems generally require adding new intructions to the processor. In addition, the compiler chain used for MASE are the GNU tools, which are open-source.

2.1.1 Modifications required in the processor pipeline for non-cacheable access to the RUs

The CPU architecture is a load/store architecture, so all memory accesses are through load and store instructions only. All processor accesses to memory go through the cache system, first through the L1 cache and then through the L2 cache. The L2 cache interfaces to the AMBA bus. So all memory accesses are through the bus. Since the slaves are all memory mapped, all accesses from the processor to the slave will be cached. As explained in the introduction, this is not desirable. We have therefore, marked the address space corresponding to the RUs as non-cacheable. We have modified the operation of the pipeline, by giving the load/store unit direct access to the bus. This essentially allows non-cacheable (NC) access to the slaves. If the processor is a scalar CPU with in-order execution, the pipeline can stall till the NC access completes [22]. However, if the processor is a superscalar CPU, issues related to the out-of-order execution property arise. The different cases where the problem could arise are:

(1) NC load after NC store to different address: As we shall see in section 2.3, the processor typically writes to a control register in a slave, followed by reading a status register, i.e, the program code will have a store to an NC address, followed by a load from another NC address. Due to out-of-order execution, it is possible for the NC load to occur before the NC store, since the addresses are unrelated. This is undesirable for NC accesses. We need consecutive NC store and NC load to execute in program order.
(2) NC load after NC store to same address: In case the program has a store to an address immediately followed by a load from the same address, the CPU in MASE prevents a read after write (RAW) hazard by performing a store forwarding, using the load-store queue (LSQ). In case the address is NC, it is possible for the addressed location contents to change between the store and load, since the location is in a slave. In such cases, the store forwarding should be prevented and the two instructions must be serialized, i.e, executed in program order. Also, the compiler should not make optimizations by storing a local copy of the stored value, to be used by the load.
(3) NC store after NC store to same address: In this case, we must ensure that the compiler does not optimize the two stores into a single store.
(4) NC load after NC load from different address: An NC load following an NC load can execute out of order. This normally does not pose any problem. But there is one case that needs to be taken care of. This generally happens in the case of branch misprediction. The user code might have a small loop that checks the status register in a slave, and remains in the loop till the status register gets a particular value, say `B`. Right after the loop, the program code might read a set of result registers. Only after the result registers have the correct value, the status register will change its value to `B`. But in case a branch misprediction occurs, the result register might be read speculatively, before the status register is set to `B`, so a wrong result value is read. But by the time the status register-read in the loop executes, the status register might be set to `B`. In that case, the CPU assumes that the the branch prediction was actually correct, and wrongly marks the speculatively executed instruction as non-speculative. As a result, a wrong result value is used. This should be prevented.
(5) NC load after NC load from same address: This is a case of Read after Read (RAR) hazard. Out of order execution normally does not cause any problem. We must ensure that the compiler does not optimize the two loads into a single load.

We have modified MASE to cater to these aspects. Any load or store operation falling in the address range meant for a slave, is recognized as an NC access. The NC request is then directly forwarded to the bus, through a queue, seen in figure 1. The queue helps in preventing the load/store units from stalling due to NC accesses. NC requests to the bus are given higher priority compared L2 cache requests to the bus, since data transfers sizes for NC transfers do not exceed eight bytes (corresponding to double word load/store instructions).

The user C code should have all NC addresses marked as `volatile`. This prevents compiler optimization of NC accesses and also ensures that all NC accesses are in the same order as present in the user code.

The changes to MASE (if any), regarding the ordering of the NC accesses are given below. No modifications in this regard are made to the L1 or L2 cache.

(1) NC load after NC store to different address: The processor pipeline has been modified so that no NC load instruction can issue till all outstanding NC stores commit, irrespective of the locations addressed by those instructions. This can affect the performance, but it is a necessary aspect to ensure correct operation.
(2) NC load after NC store to same address: Store forwarding is prevented when an NC load follows an NC store to the same address, by modification #1.
(3) NC store (load) after NC store (load) to (from) same address: All NC addresses are marked as `volatile`, in the user C code. This ensures that consecutive NC stores (loads) to (from) same address are not optimized to a single NC store (load).
(4) NC load after NC load from different address: We take care of this case in the user code itself, without changing the pipeline. In case an NC load following an NC load to a different address needs to be serialized, a dummy store to an NC location is inserted between the two loads, in the user program code. Due to modification #1, the second load then executes only after the dummy store executes. Since a store executes only in the commit stage while the store instruction retires, the dummy store will be executed only after it is marked as non-speculative. This ensures that the second load is also executed non-speculatively.

2.1.2 Memory Model

The MASE memory model employs two levels of caches, by default. It is possible to add any number of cache levels to the simulator. All the caches employ a write-back and write-allocate-on-miss caching strategy. The memory model in MASE has several limitations, especially when interfaced to a cycle-accurate bus. We have modified the memory model in MASE, to make it more realistic. The modifications are:

(1) Finite size write buffers are added for L1 data cache and L2 unified cache. The size of the write buffer is a command line parameter, in terms of the number of cache lines it can hold. Whenever a cache miss occurs, the write buffer is checked for a hit.
(2) A common bus is modeled between L1 and L2 cache levels. This common bus is shared by the L1 instruction and data caches, in accessing L2 cache. Data cache accesses are given higher priority. Within the data cache, read accesses normally have higher priority, to ensure lesser cache miss penalty. In case the write buffer gets full, the write access is given higher priority.
(3) The caches are made more realistic by modeling roughly the actual oper-

ations that occur in the cache. On a cache miss, cache line status is not updated by a latency value returned immediately from the lower level cache or memory. Instead, the cache data structures are updated every cycle, based on whether the lower level has completed the requested operation. Such a modeling is helpful while interfacing the L2 cache to a cycle-accurate bus. This takes care of unknown access times caused by the bus arbitration.

(4) The caches are converted, from being completely non-blocking, into partially blocking caches. They can sustain only one cache miss at a time, but can serve multiple hits under a miss.

2.2 The bus model

As an example for a bus, we have chosen the Advanced Microcontroller Bus Architecture (AMBA) [11]. This is an open specification for SoC systems [17]. A synthesizable VHDL model for the AMBA is available.

The AMBA comprises of two components. One is the Advanced High-performance Bus (AHB), meant to act as a high performance system backbone bus. This bus supports high bandwidth data transfer. System components attached to this bus can be either masters or slaves. Another component of the AMBA is the Advanced Peripheral Bus (APB). APB is optimized for mimized power consumption and reduced interface complexity. Typically, low bandwidth peripherals and registers are attached to the APB as slaves. The APB is interfaced to the backbone AHB, through an APB bridge. The APB bridge is a slave on the AHB and the only master on the APB.

The processor acts as a master on the AHB. For the processor, we have developed a master interface to the AHB, in HDL. This interface accepts bus requests from the processor model and translates them to AHB protocol signals.

2.3 Processor-RU interaction

As mentioned in the beginning of this section, we have modeled all RUs as slaves on the AHB bus. To enable interaction between the processor and the various RUs, the RUs have status and control registers, addressable by the processor. The RUs also hold input and output buffers or FIFOs, to hold the input data and the computed result, respectively. These buffers, as well as the status and control registers, are memory mapped. The processor takes care of transferring data to the RU, enabling the RU for computation and reading back the result data. The typical sequence of operations is outlined below:

(1) Processor writes data to the RU input buffer. The processor also writes other information into the RU, such as data size.

(2) Then the processor writes to a control register in the RU, to start RU execution.
(3) The processor polls the status register of the RU, waiting for completion of RU execution. When RU completes its execution, it changes the status register value.
(4) After RU execution, the processor reads back the result data from the output buffer of the RU.

Each RU is allocated an address range for access. This address range information is required in the processor model, for recognizing the slave accesses as non-cacheable. So the information is hardcoded in the processor model. This information is also hardcoded in the bus decoder. This hardcoded information determines the number of RUs that can be active on the bus at a given time. The user code also should have knowledge of the address range allocation to slaves.

2.4 Reconfiguration model

As is the case with most reconfigurable systems, we have assumed that the configuration data for all the RUs is present in a separate configuration PROM. Data transfer between this memory and the e-FPGA is via a separate 32-bit bus. This is shown in figure 1. Once reconfiguration is initiated, configuration data is transferred at the rate of one word every clock cycle. The clock for reconfiguration is assumed to be the same as the system clock. The processor is solely responsible for initiating any reconfiguration process, but actual data transfer is controlled by a separate configuration controller. The configuration controller is responsible for generating the address and control signals for the e-FPGA. It has status and control registers visible in the processor address space. A typical sequence of operations for full reconfiguration (Section 4.3) is given below. For purpose of illustration, two functions F-1 and F-2 are assumed to be separately mapped to the e-FPGA as RU-1 and RU-2, respectively.

(1) Processor executes function F-1 on RU-1, as outlined in Section 2.3.
(2) Processor writes to control register of configuration controller, to initiate full reconfiguration of e-FPGA, into RU-2.
(3) The processor keeps polling the status register of the configuration controller, waiting for completion of reconfiguration. The controller changes the value of the status register, on completion of reconfiguration.
(4) Processor executes function F-2 on RU-2.

During full reconfiguration, the e-FPGA cannot perform any useful computation. Normally, any computation done by the processor is complemented by that done on the e-FPGA. As a result, during full reconfiguration, the processor is also not involved in performing useful computation. This is evident in the sequence of operations presented above. However, during partial reconfiguration, it is possible for part of the e-FPGA (and consequently the processor also) to be computationally active. A typical sequence of operations for partial reconfiguration is given below. As before, the functions F-1 and F-2 are mapped to hardware units RU-1 and RU-2. We assume that RU-1 is already present as the base configuration and the e-FPGA needs to be partially reconfigured to accomodate RU-2. The sequence of operations are

(1) Processor writes to control register of configuration controller, to initiate the partial reconfiguration of e-FPGA, to accomodate RU-2.
(2) Processor begins execution of F-1 on RU-1 (if it is not already running).
(3) During this period, the processor has to poll the status register of the configuration controller, to check if the reconfiguration is complete. It is also possible that RU-1 may require data from memory, which has to be supplied by the processor since the RUs are slaves. The processor may also read back the output buffer of RU-1, to prevent it from overflowing, and then perform some computation on the output data. All these operations are interleaved and they run in parallel with the reconfiguration process.
(4) The configuration controller updates its status register once the configuration data is transferred.
(5) The processor starts execution of F-2 on RU-2

We have an RTL model for the configuration data transfer in Verilog HDL. The configuration data size and control sequence is based on the Xilinx XCV-1000 device.

However, the number of cycles needed for configuration data transfer can be predicted accurately. This is unlike the main system bus, where there is an element of unpredictability in the number of bus cycles due to the memory hierarchy and a variable number of RUs on that bus. It is therefore possible to reduce the simulation time by using a simple blocking delay in the verilog module for configuration data transfer. The amount of delay depends on control signals needed to program the e-FPGA and the number of configuration words to be transferred. Doing this reduces the simulation time by about 5% for the examples simulated. The final results are not affected.

3 Co-simulation environment

A hardware-software (HW-SW) co-simulator is essential for accurate performance evaluation of bus-based reconfigurable systems (see Section 1). Simulators have been developed for the various co-processor and functional unit based reconfigurable architectures [3,8,4,5,7]. Most of them include ‘C’ models of the RU in the CPU simulator. We have only come across Zippy [8] as an example of a co-simulator for reconfigurable systems. OneChip [3] and Chimaera [4] have a cycle accurate C-model for the processor, similar to our own, but they model the attached reconfigurable fabric in ‘C’ itself. Garp has a cycle accurate simulator which models the MIPS processor and the reconfigurable co-processor [23]. Morphosys has simulators for the whole system, in C and separately in VHDL as well [7].

Commercial tools are available for performing hardware-software co-simulation.

These include Seamless from Mentor Graphics [24], CoSim from SGS-Thompson Microelectronics [25] and Celoxica's Nexus PDK [26]. The commercial tools offer instructions set simulators (ISS) for specific commercial processors for the purpose of codesign and co-simulation. While these tools are not meant for reconfigurable systems, they could be modified for our framework. A general simulation environment like Ptolemy can also be used to build a simulator conforming to our requirements [27]. However, we have found it convenient to build a co-simulation environment ourselves, using a C-model for the CPU. Since a VHDL model for the AMBA [11] is available, we have chosen to model the bus in HDL. The RUs are modeled in HDL, since it allows us to plug-in off-the-shelf HDL models into our simulation framework.

In a C-HDL co-simulator like ours, it is possible to run the 'C' and HDL as either a single process or two independent processes. Running them as two independent processes requires inter-process communication (IPC). Among the two approaches, viz., non-IPC and IPC based approaches, non-IPC based approach is fastest, since there is no communication overhead. However, non-IPC based approach has some problems:

(1) SW processor model may not be compatible with the HW simulator.
(2) Debugging in a combined binary is difficult.
(3) In a single-process simulation, the HDL should run as the master and the software model as a slave, being executed as a procedure call. Execution starts at the same point in the procedure, everytime it is called [28]. Processor simulators are not written that way.

This has motivated us to choose the IPC-based approach. The IPC method as well as the HW-SW synchronization details are presented in the following subsections.

3.1 Inter-process communication(IPC)

Inter-process communication under Unix based systems can be performed by various methods [29]. Among them, two approaches are: using sockets and using shared memory. In both these approaches, an application programmer interface (API) is available to the programmer, in the 'C' language. Socket based IPC is generally used when the communicating processes are on physically separate machines. Using socket is a very popular approach for HW-SW cosimulation [30–33]. Shared memory based IPC can be used only by processes running on the same machine. Access to the shared memory area is managed by using semaphores.

We have implemented both the methods of IPC. ModelSim 5.7 is used for the HDL simulation. The ModelSim HDL simulator runs as one process while the processor 'C' model runs as another process. The Verilog Procedural Interface (VPI) is used for calling some C routines from HDL. These C routines are necessary for using the inter-process communication API provided by Unix/Linux. The next subsection gives details about the synchronization mechanism between the communicating processes.

3.2 HW-SW synchronization

As mentioned in section 2.2, the processor bus-master interface is in HDL. The IPC is between the processor 'C' model and the HDL master interface. This communication between C and HDL needs to be synchronized. One approach is that C and HDL exchange messages at every clock tick. This is the approach adopted in [32], for HW-SW co-simulation. This approach incurs lot of synchronization overhead [33]. So, the HDL generally provides an estimate of the number of cycles for which the hardware resource will be busy, so that messages need not be exchanged during those cycles.

The simulation flow is as follows. The HDL simulator is started, and it runs as a "server", waiting for simulation requests from the 'C' model or processor. Simulation time in HDL does not proceed forward. However, 'C' keeps simulating, until a bus access is required. A bus access will be required either for L2 cache line read, L2 cache line write, non-cacheable read or non-cacheable write. When either of these conditions occur, a request is sent from 'C', the "client", to the HDL simulator. The HDL then estimates the number of cycles the bus will be busy, and sends back a conservative skip count `N`, which tells 'C' that it can simulate for the next `N` cycles, without performing any message transfer between C and HDL. (If the processor is the only master on the bus, then the processor has continuous ownership of the bus. In that case, the estimate `N` will be perfect, since the bus protocol is known). After `N` cycles, the HDL waits for message from C. Again, the HDL simulation time does not proceed forward, until a message from C arrives. In case the bus transaction is not yet completed, the HDL keeps ready another conservative estimate `N` to be sent back to C. After the message from C arrives, HDL sends `N` back to C. This goes on back and forth till the bus transaction completes. The HDL then goes into wait mode, waiting for another simulation request from C. The HDL simulator need not keep track of the actual system simulation time.

4 RESULTS

Using the simulator, we have looked at the performance of two applications - namely, matrix multiplication and Lempel-Ziv compression. First, the speedup that can be obtained for various matrix sizes/input data sizes is studied. This is necessary, because unless reasonable speedup is obtained, it is not worthwhile using reconfiguration. In this case, a base configuration is assumed to be loaded at startup, as it is normally the case for multicontext architectures.

Then a scenario is studied when both the example operations may be required. Two cases are considered. In the first case, the entire e-FPGA is dynamically reconfigured. In the second, the e-FPGA is partially reconfigured, while one

Simulation Parameter	Value
Fetch width	4 instructions / cycle
Decode width	4 instructions / cycle
Issue width	4 instructions / cycle
Commit width	4 instructions / cycle
Reorder buffer size	16 entries
Reservation table size	16 entries
Load-Store Q size	8 entries
Instruction size	32-bit
Functional units	4 intALUs, 1 intMUL, 2 memory ports, no FP units, 2 mem ports
L1 I-cache	16KB, direct mapped, 512 sets, 32-bytes per cache line, LRU
L1 D-cache	16KB, 4-way assoc., 128 sets, 32-bytes per cache line, LRU
L2 unified	256KB, 4-way assoc., 1024 sets, 64 bytes per cache line, LRU
Write buffer size	4 cache lines for L2 and L1 D-cache
L1/L2 bus	Width L1 cache line, 1-cycle thoughput, 4 cycle latency
L1 hit latency	1 cycle
L2 hit latency	6 cycles

Table 1
MASE simulation parameters

application is in progress. Table 1 shows the MASE simulation parameters that we have used for these simulations.

4.1 Matrix multiplication

In matrix multiplication A·B, the core computation is performed in the RU, attached to the AHB bus as a slave. The processor takes care of sending input data to the RU and reading back the result data from the RU and storing it in memory. Here it is assumed that the configuration is pre-loaded in the e-FPGA at startup.

To have dynamic reconfiguration and partial reconfigurability, it is advantageous to have architectures that exhibit a high degree of regularity and scalability. We have therefore designed the matrix multiplication unit, so that it can be easily scaled to accomodate various matrix sizes. This architecture is shown in Figure 2. Each attached RU can store a partial row of A and a partial column of B. As seen in Figure 2, each RU can do a 16x16 multiplication. Within each RU there are four MAC units, each shared by four

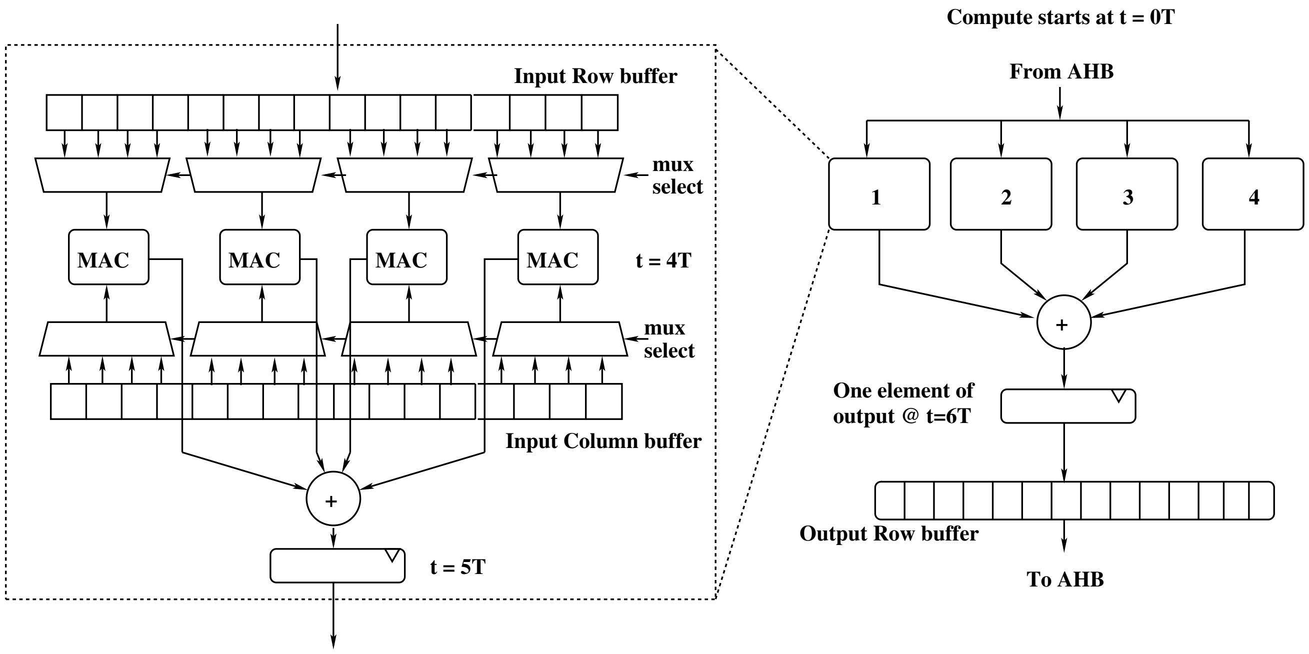

Fig. 2. Architecture of Matrix Multiplication RU. The numbered boxes are RU modules capable of mulitplying two vectors of length 16 each. The box numbered 1 is enough for 16x16 matrix multiplication, while all the four boxes are required for 64x64 multiplication.

elements. Hence, four clock cycles are required to complete the multiplication. The results of the four MAC units are added in the fifth clock cycle and an additional cycle is required to add the results from various RUs. As shown in figure 2, the RUs are capable of producing one element of result matrix in six clock cycles.

The row and column buffer are mapped to the processor address space. The number of RUs required depends on the row size of A or the column size of B. Another column buffer (not shown) is provided to pre-fetch the next column during computation. The control and status registers for the RU are attached to the APB bus. It can been that the RU architecture shows a high degree of regularity and scalability. This architecture can be easily mapped to a reconfigurable fabric with one column of logic as the basic unit of reconfigurable logic (one example of such a fabric is the Virtex device from Xilinx [13]). This allows us to reconfigure the e-FPGA for different input matrix sizes.

The simulation results are presented in Figure 3, for different sizes of matrices A and B. It is clear that considerable speedup can be obtained, especially for larger sized matrices. Table 2 shows simulation time for a 32x32 matrix. This table gives the simulation time for both the IPC approaches that we have used. It can be seen that the simulation using shared memory IPC is faster. Socket based IPC is especially slow when a lot of bus transactions are present, as for the case using RU for 32x32 multiplication.

Application	No. of cycles	Speedup w.r.to SW	Simulation Time (secs)	
			TCP sockets	Shared Memory
Matmul (SW) 32x32	754244	–	49	40
Matmul (HW) 32x32	443913	1.7	1581	248
LZ77 (SW) 1100 bytes	2723745	–	125	73
LZ77 (HW) 1100 bytes	251805	10.8	2768	846
Combined(SW)	3480462	–	–	–

Table 2
Results for simulation run on the co-simulator. All simulations were run on a single, Pentium-3 866MHz computer. Applications marked as SW do not use the RU, while those marked HW use the RU for computation. Simulation times are shown for both socket based IPC and shared memory based IPC

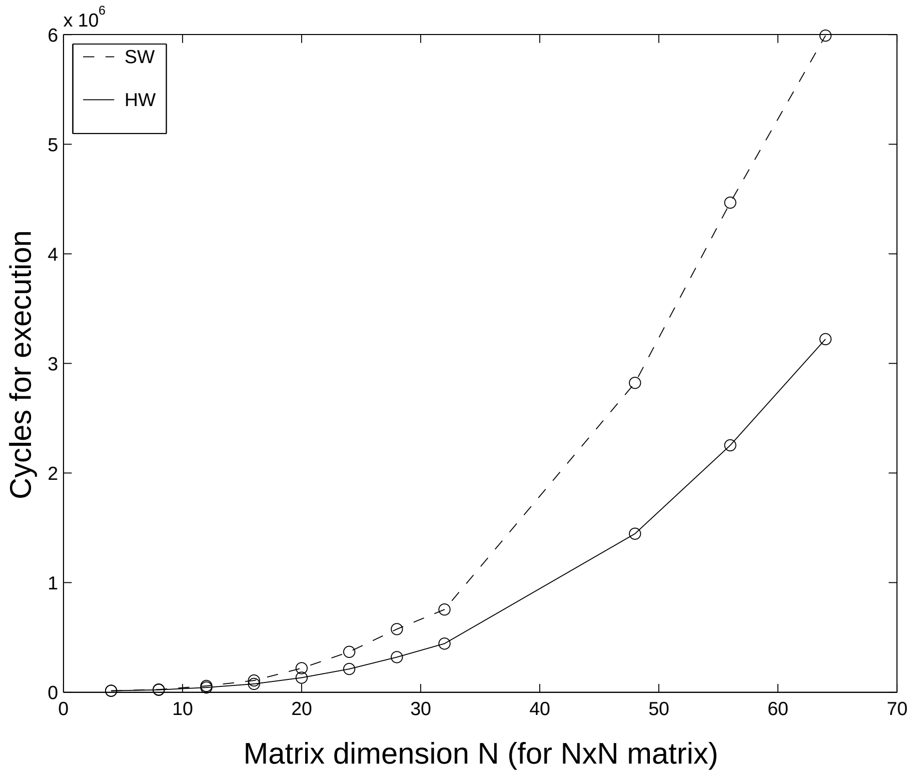

Fig. 3. Simulation results for matrix multiplication example. For ‘SW’, the RU is not used for multiplication - only memory accesses appear on the bus. For ‘HW’, the RU is used for computation.

4.2 Lempel-Ziv Compression

The Lempel-Ziv algorithm is a universal algorithm for lossless compression of sequential data. Proposed in 1977 [34], it is commonly referred to as LZ77. LZ77 operates by maintaining a buffer of symbols from the input data/symbol stream. One part of the buffer holds a set of recently encoded symbols. This part of the buffer is known as the search window. The other part of the buffer holds the new symbols received from the data stream, which have not been

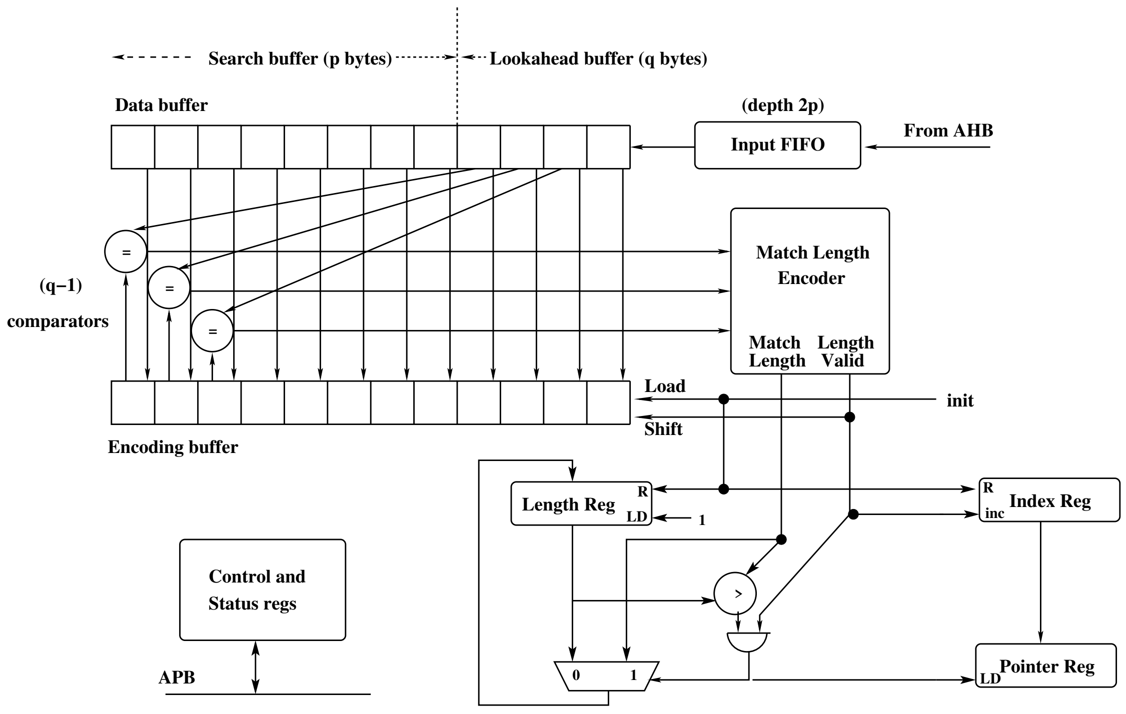

Fig. 4. Architecture of LZ77 compression RU. It is based on the architecture in [35]

encoded. This is the lookahead window. LZ77 achieves compression by locating within the search window, the longest matching string from the lookahead window. The compressed codeword consists of the length of the longest matching string and the pointer to the starting position of the string, in the search window.

LZ77 compression thus involves three key steps: reading data into a buffer, matching data in the buffer and generating a codeword. Since data matching is the most time consuming operation, it is performed in the RU hardware. The other two steps are performed in software. The RU is attached to the AHB bus as a slave. The processor takes care of sending input data to the RU, reading back the result data from the RU and generating the codeword. It is assumed that the configuration for LZ77 is pre-loaded in the e-FPGA at startup.

The architecture of the RU is shown in figure 4. It is based on the architecture presented in [35]. We have added an input FIFO buffer, to facilitate data transfer to the RU, while the RU is executing. The search window size (p) is set to 256 bytes and the lookahead window size (q) is chosen to be 16 bytes. The FIFO size is set to 512 bytes. The FIFO input port is mapped to the processor address space. The status and control registers, attached to the APB bus, are also part of the processor address space. The RU also consists of two internal buffers, each of size (p+q) bytes. One is the data buffer, which consists of the aforementioned search window and lookahead window. The other buffer is the encoding buffer, which is initialized to the data buffer contents before string matching begins. The encoding buffer is shifted left as the string matching operation parses the search window, searching for the longest matching string. (q-1) comparators are used to compare the lookahead window contents against the encoding buffer contents. The match length encoder encodes the comparator outputs into a binary value indicating the string match length. The largest value of the match length is maintained in the 'length register', which is mapped to the processor address space. The 'index register' indicates the starting position of the string currently being compared, in the search window. The index register is incremented every time the encoding buffer is shifted. The 'pointer register' holds that value of the index register which corresponds to the largest length. The length and pointer registers are accessible to the processor.

The following sequence of operations are repeatedly performed till the input data is exhausted:

(1) Processor writes data to the input FIFO of the RU, then initiates string matching by writing to the control register of the RU.
(2) Data is read sequentially from the input FIFO into the data buffer (figure 4)
(3) The data buffer contents are loaded into the encoding buffer.
(4) The string matching operation begins. The encoding buffer is shifted left, till the whole search window is exhausted. The processor continues to write data to the FIFO in the meanwhile.
(5) String matching finishes. The length register and pointer register have the required values. The status register value indicates that operation has completed.
(6) Processor reads the length and pointer values from the RU and then encodes them into a codeword.

We see that string matching in LZ77 involves byte level operations. Doing it in software is an inefficient use of the 32-bit processor. Further, performing the string matching on the processor entails repeated fetching of data, as the search window is parsed. This repeated data fetch is avoided when string matching is performed in the RU. Therefore we expect significant speedup when LZ77 is performed using the RU. The simulation results are presented in Figure 5, for different input data sizes. As expected, large amounts of speedup are displayed, when the RU is used. In fact, the speedup obtained for LZ77 is much greater than the speedup obtained for matrix multiplication.

Table 2 shows simulation time for data size of 1100 bytes. It shows the simulation time for both the IPC approaches we have used. Again, it can be seen that simulation using shared memory IPC is faster.

4.3 Combined matrix multiplication and LZ77

We simulated both LZ77 compression and 32x32 matrix multiplication combined together as a single application. We have taken the base configuration in the e-FPGA to be that of LZ77 RU. The simulations were done assuming that the e-FPGA is similar to Xilinx Virtex XCV-1000. The area of this device is not sufficient to hold both the 32x32 multiplication and the LZ77 RUs at the same time. The following two cases were simulated.

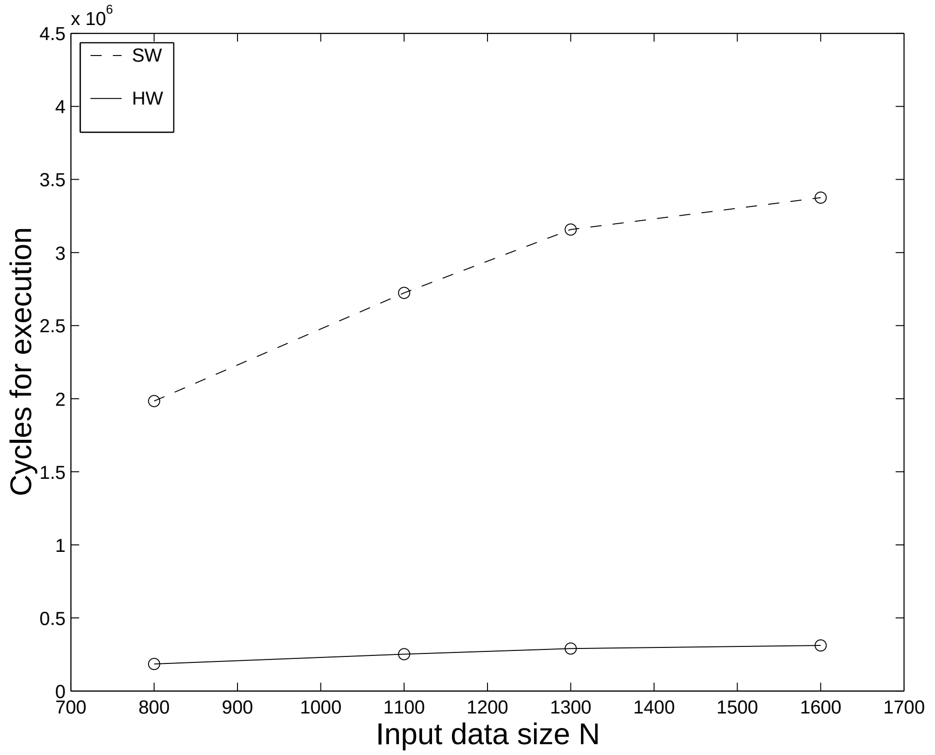

Fig. 5. Simulation results for LZ77 example. For 'SW', the RU is not used for LZ77 compression - only memory accesses appear on the bus. For 'HW', the RU is used for LZ77.

(1) Perform LZ77 compression completely, followed by 32x32 multiplication, using the following sequence of steps
    (a) Perform LZ77 compression (base configuration is for LZ77). When the requirement for matrix multiplication arises in-between, register the request.
    (b) Do a context-switch, i.e, completely reconfigure the e-FPGA to accomodate the 32x32 multiplication RU.
    (c) Perform 32x32 multiplication
(2) Partially reconfigure the e-FPGA to accomodate 16x16 multiplication RU, while executing LZ77. The steps are given below:
    (a) Start performing LZ77 compression (base configuration is for LZ77)
    (b) After a short while, when requirement for matrix multiplication arises, initiate partial reconfiguration of the e-FPGA to accomodate the 16x16 multiplication RU. Continue LZ77 compression in parallel.
    (c) As soon as partial reconfiguration completes, do both LZ77 and matrix multiplication in an interleaved manner. The 32x32 matrix multiplication is done using the 16x16 multiplication RU.

It may be noted that a third possibility exists, wherein the base configuration itself holds both LZ77 RU as well as the 16x16 multiplication RU, in anticipation that multiplication may be required later. In that case there is no need for reconfiguration itself, and in itself is an example of static configuration rather than dynamic reconfiguration.

In case 1, reconfiguration latency is high due to full reconfiguration, although the matrix multiplication takes lesser time since it has a large RU allocated. In case 2 , the reconfiguration time is low and LZ77 computation can proceed during the partial reconfiguration process, but matrix multiplication takes longer time since it has a smaller RU allocated to it.

Xilinx Core Generator was used to obtain estimates of the number of CLBs used by the different RUs. XCV-1000 requires 6,127,744 bits to configure the whole device. The 32x32 multiplication RU occupies almost the full device, 16x16 multiplication occupies about 50% of the the device, while the LZ77 RU requires 5% of the device. Table 3 gives the results. We see that best performance in terms of total execution time is obtained, when we employ partial reconfiguration.

We have repeated the simulations for case 1 and case 2, for performing LZ77 compression and 64x64 matrix multiplication. We have appropriately scaled up the e-FPGA size to correspond to double of Xilinx Virtex XCV-1000, in terms of number of CLBs. The area of this hypothetical device is not sufficient to hold both the 64x64 multiplication and the LZ77 RUs at the same time. However, it can accomodate the RUs for 32x32 multiplication and LZ77. The results obtained are given in Table 4. Here we see that doing a full reconfiguration is a better alternative to partial reconfiguration, as regards the total number of cycles for execution.

The result in Table 3 indicates that it is possible to obtain better performance, by employing partial reconfiguration. Table 4 however warns us that, this need not always be the case. We see that one of the important factors is the reconfiguration time relative to the actual application execution time and the associated overhead. The overhead consists of both bus cycles as well as the additional cycles required for operations to be performed in software when a smaller RU is used. When the 64x64 matrix multiplication is performed using 32x32 multiplication RU, the overhead due to using the smaller RU turns out to be much larger. So in this case, full reconfiguration takes approximately 400,000 cycles less than partial reconfiguration and is the better option.

It may be noted that in the literature, there exist examples of multi-context architectures with smaller reconfiguration times than the representative values considered here. For example, Zippy [8] has a reconfigurable fabric that allows for a single-cycle context switch.

5 Conclusions

We have presented the issues involved in coupling reconfigurable units to a superscalar processor, in a bus-based system. We have developed a cycle-accurate co-simulator for such a system. Simulation results have been presented for three examples, viz, matrix multiplication, Lempel Ziv compression and a combination of both, employing full/partial reconfiguration. In all the cases,

Case	Reconfig cycles	Total cycles	Speedup w.r.to SW	Bus cycles
Pure SW	–	3480462	–	9435
Full reconfig	191492	879768	3.96	401732
Partial reconfig	95746	819635	4.25	261869

Table 3
Combined LZ77 and 32x32 matrix multiplication, with reconfiguration. Full reconfiguration requires 6,127,744 bits or 191492 words of configuration data. A 50% reconfiguration correspondingly requires around 95746 words. Configuration data is transferred one word per clock cycle

Case	Reconfig cycles	Total cycles	Speedup w.r.to SW	Bus cycles
Pure SW	–	8726493	–	19227
Full reconfig	382984	3864819	2.26	1061409
Partial reconfig	191492	4275367	2.04	805219

Table 4
Combined LZ77 and 64x64 matrix multiplication, with reconfiguration. Full reconfiguration requires 12,255,488 bits or 382984 words of configuration data. A 50% reconfiguration correspondingly requires around 191492 words. Configuration data is transferred one word per clock cycle

speedup compared to pure software implementation has been demonstrated.

In our simulator, the processor is modeled in 'C' and the bus and reconfigurable units are modeled in HDL. This has some advantages and limitations. Modeling everything in 'C' will make the simulator faster. However, we have chosen the given approach so that off-the-shelf HDL models can be directly integrated into our simulator. We see that there is a trade-off between the simulation speed and the flexibility of adding HDL models. Ideally, we need a hierarchy of models within the simulation environment so that user can explore design trade offs at various levels. This requires further work. As regards simulation time, our results indicate that HW-SW cosimulation based on shared memory IPC is better. Although socket based IPC has been used previously in several simulators [30–33], our results show that it is not worthwhile, unless simulations need the 'C' and HDL processes to run on different machines.

In a bus-based reconfigurable system, our results indicate that the best results are not always obtained with partial reconfiguration. Speedup obtained depends on the reconfiguration overhead and software overhead that occurs when a smaller reconfigurable unit is used. Predicting which type of reconfiguration gives better results requires further study.

6 Acknowledgement

The authors would like to thank the editors of the Microprocessors and Microsystems Special Issue on FPGAs, for providing an opportunity to submit this paper and encouraging them to submit the manuscript to the regular issue with modifications. The authors also thank the anonymous reviewers of the special issue, for their critical comments and suggestions.

References

[1] A. DeHon, DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century, in: D. A. Buell, K. L. Pocek (Eds.), IEEE Workshop on FPGAs for Custom Computing Machines, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 31–39.

[2] K. Compton, S. Hauck, Reconfigurable Computing: A Survey of Systems and Software, ACM Computing Surveys 34 (2) (2002) 171–210.

[3] J. E. Carrillo, P. Chow, The Effect of Reconfigurable Units in Superscalar Processors, in: Ninth ACM International Symposium on Field-Programmable Gate Arrays (FPGA'01), ACM/SIGDA, 2001, pp. 141–150.

[4] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, The Chimaera Reconfigurable Functional Unit, in: K. L. Pocek, J. Arnold (Eds.), IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 87–96.

[5] J. R. Hauser, J. Wawrzynek, Garp: A MIPS Processor with a Reconfigurable Coprocessor, in: K. L. Pocek, J. Arnold (Eds.), IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 12–21.

[6] T. Miyamori, K. Olukotun, REMARC: Reconfigurable Multimedia Array Coprocessor (abstract), in: 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, 1998, p. 261.

[7] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, E. M. C. Filho, Morphosys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications, IEEE Transactions on Computers 49 (5) (2000) 465–481.

[8] R. Enzler, C. Plessl, M. Platzner, Co-simulation of a Hybrid Multi-Context Architecture, in: 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), CSREA Press, 2003, pp. 174–180.

[9] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer, Piperench: A Coprocessor for Streaming Multimedia Acceleration, in: Proceedings of the 26th International Symposium on Computer Architecture, 1999, pp. 28–39.

[10] M. Borgatti, F. Lertora, B. Forêt, L. Calf, A Reconfigurable System Featuring Dynamically Extensible Embedded Microprocessor, FPGA and Customizable I/O, IEEE Journal of Solid-State Circuits 38 (3) (2003) 521–529.

[11] ARM Ltd., AMBA Specification (Rev 2.0), ARM IHI 0011A (May 1999).

[12] K. Bondalapati, V. K. Prasanna, Reconfigurable Computing Systems, Proceedings of the IEEE 90 (7) (2002) 1201–1217.

[13] Xilinx Programmable Logic Devices, FPGA and CPLD, World Wide Web, `http://www.xilinx.com` (2003).

[14] A. Pelkonen, K. Masselos, M. Cupák, System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC, in: International Parallel and Distributed Processing Symposium (IPDPS 03), 2003, pp. 174–181.

[15] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, 1990.

[16] E. Larson, S. Chatterjee, T. Austin, MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling, in: IEEE International Symposium on Performance Analysis of Systems and Software, 2001, pp. 1–9.

[17] AMBA 2.0 specification on the ARM website, World Wide Web, `http://www.arm.com/products/solutions/AMBA_Spec.html`.

[18] R. Maestre, F. Kurdahl, M. Fernandez, R. Hermida, N. Bagherzadeh, H. Singh, A formal approach to context scheduling for multicontext reconfigurable architectures, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9 (1) (2001) 173–185.

[19] Xilinx Inc., XAPP151: Virtex Series Configuration Architecture User Guide, application Note: Virtex Series (March 2003).

[20] C. J. Hughes, V. S. Pai, P. Ranganathan, S. V. Adve, RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors, IEEE Computer 35 (2) (2002) 40–49.

[21] D. Burger, T.Austin, The SimpleScalar Toolset Version 2.0, Tech. Rep. 1342, University of Wisconsin, Computer Sciences, `http://www.simplescalar.com` (June 1997).

[22] The Leon2 Processor Users Manual, World Wide Web, `www.gaisler.com` (2003).

[23] J. R. Hauser, Augmenting a Processor with Reconfigurable Hardware, Ph.D. thesis, University of California, Berkeley (2000).

[24] Seamless Hardware/Software Co-Verification Datasheet, Mentor Graphics Corporation, `www.mentor.com/seamless/datasheets/index.html` (2003).

[25] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, A. Jerraya, System-on-a-chip Cosimulation and Compilation, IEEE Design and Test of Computers 14 (2) (1997) 16–25.

[26] Nexus PDK Co-Design and Co-Verification Tool Suite Data Sheet, Celoxica Ltd., `http://www.celoxica.com/products/tools/nexus-pdk.asp` (2003).

[27] J. Liu, M. Lajolo, A. Sangiovanni-Vincentelli, Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator (1998).

[28] H. Hubert, A Survey of HW/SW Cosimulation Techniques and Tools, Master's thesis, Royal Inst. of Tech., Stockholm, Sweden (June 1998).

[29] W. R. Stevens, Unix Network Programming, Prentice-Hall Inc., 1990.

[30] D. Becker, R. K. Singh, S. G. Tell, An engineering environment for hardware/software co-simulation, in: 29th ACM/IEEE Design Automation Conference, 1994, pp. 129–134.

[31] S. L. Coumeri, D. Thomas, A simulation environment for hardware-software codesign, in: 1995 IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD'95), 1995, pp. 58–63.

[32] M. Pirvu, L. Bhuyan, R. Mahapatra, Hierarchical Simulation of a Multiprocessor Architecture, in: 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD'00), 2000, pp. 585–588.

[33] S. Yoo, K. Choi, Synchronization Overhead Reduction in Timed Cosimulation, in: Int. High Level Design Validation and Test Workshop (HLDVT97), 1997.

[34] J. Ziv, A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory 23 (3) (1977) 337–343.

[35] Y. J. Kim, K. S. Kim, K. Y. Choi, Efficient VLSI Architecture for Lossless Data Compression, IEE Electronics Letters 31 (13) (1995) 1053–1054.