Hybrid Hardware/Software Co-execution using Continuations

Isaac Gelado*,1, Enric Morancho*,1, Nacho Navarro*,1

* Computer Architecture Department, Universitat Politècnica de Catalunya, Jordi Girona 1-3, 08038 Barcelona, Spain

1 Introduction

Upcoming computer architectures will include additional computational elements besides one or more general purpose processors. For instance, IBM, Sony and Toshiba have developed the Cell processor, composed of a PowerPC processor and up to eight Synergistic Processing Elements (SPEs). Xilinx offers the Virtex-II Pro, a chip which includes two PowerPC cores surrounded by a Field Programmable Gate Array (FPGA). Application-Specific Accelerators (ACCs) can be instantiated into the reconfigurable logic or into the SPEs to perform customized computations [Gela06].

This poster presents our experimental work using fine grain accelerators on current reconfigurable architectures. It shows that currently there are three major overheads on these architectures. First, the process model does not allow transferring the control flow from the processor to ACCs. ACCs are not able to call another ACC or piece of code, so the processor must wait until one ACC finishes to call the next one. This requires implementing synchronization mechanisms between the processor and the ACCs, which introduce a large overhead. Second, applications copy huge amounts of data between the main memory and the accelerators and vice versa. On our prototype implementation, the data movement time is about two orders of magnitude bigger than the ACC computation time. Finally, there is a huge contention on the interconnection network when ACC are being used. It leads to increasing up to 100 cycles the main memory access time.

The poster sketches our incoming research on the three previous problems.

1. A new execution model for ACC based on continuations. This model enables ACCs to call other ACCs or code into the processor. Using continuations the processor transfers the program control flow to an ACC, so synchronization mechanisms are not needed any more.

2. A streaming data flow. The results from one computation (whether it has been performed by an ACC or the processor) are passed directly as the input for the next one.

E-mail: {igelado,nacho}@ac.upc.edu
It avoids copying data from the main memory to ACC and vice versa, so the data movement overhead is reduced.

3. A reconfigurable interconnection network for ACCs. It provides support to the continuation model and the streaming data flow. This network is able to dynamically switch the local memory each accelerator is using. Therefore, data is passed without using the bus that interconnects the processor and the main memory.

This poster also shows experimental reconfiguration times on the Virtex-II pro. On the basis of these results the poster explores new hardware and software scheduling policies and new features the operating system will offer.

2 Experimental Work

The experimental work has been done using a prototype platform based on the Xilinx University Program board. A dithering accelerator has been developed to be used by audio applications. This accelerator is able to process a stereo input (two samples) in one 25MHz cycle. The madplayer application has been modified to use the dithering accelerator. The accelerator substitutes to calls to the same function inside a loop. When playing a 10.2 seconds MP3 file, the accelerator is called 452,742 times.

The software function implementing the dithering takes 340 cycles when CPU is running at 100MHz and 362 cycles when it runs at 300MHz. Therefore, the accelerator provides an speed-up of 85x and 30x respectively. However, when the cost of moving the data from the processor to the accelerator is considered, the speed-up becomes 1.78x at 100Mhz and 1.41x at 300Mhz. The data movement overhead offsets the accelerator speed-up.

The cost of getting the data from the main memory to the processor has been also measured. Bringing the data from memory to the processor takes about 110 more cycles (independently of the CPU clock) when using the hardware accelerator. This overhead is due to the contention on the interconnection network, which is shared between the main memory and the accelerator.

3 Continuations: Operating System and Programming Model

A continuation is a representation of the execution state of a program at a certain point in time. The control flow of a program can be transfered by calling to a continuation, which restores its state and starts executing. Programming languages have used continuations during the last forty years [Land65]. They also have been proposed to improve communications in operating systems.

Continuations can be extended to include hardware accelerators. An application creates as many continuations as ACCs it will use. ACCs are invoked by saving the state and calling the correspondent continuation. For instance, in the code shown in Figure 1, the function filter() could be implemented by an ACC.

Parallel execution can be accomplished by spawning a continuation instead of calling it. In the code shown in Figure 1(b), continue(filter) is substituted by spawn(filter). Therefore, the processor will start processing the next block while the previous one is being filtering by the ACC.
Figure 1: (a) traditional call/return code. (b) how this code can be transformed to use continuations.

The compiler will transform sequential call/return code (as those shown on figure (a)). Then it will determine which functionalities are suitable to be implemented by ACCs and it will extract the available parallelism. Finally, it will transform the code to make use of continuations and allow parallel execution.

4 Reconfigurable Interconnection Architecture

A continuation based execution model could be implemented on current reconfigurable architectures, but it would not offer any performance gain due to the big data movement overhead between the processor and the ACCs. However, the continuation model allows implementing different reconfigurable architectures which do not require moving big amounts of data among the ACCs.

The data movement from the main memory to ACC and from one ACC to another introduces a huge overhead which can overcome the ACC speed-up [Gela06]. High-bandwidth and low-delay interconnection networks reduces this overhead. However it is not always feasible to implement such a network. This poster presents a completely different approach: avoiding the data movement. When an ACC finishes its computation, it calls a continuation to change the control flow to another ACC or to the main processor. All the data the continuation requires is its state, which is stored on the caller ACC local memory, and the caller will not longer use this data. Therefore, the interconnection architecture just disconnects the local memory from the caller and connects it to the ACC implementing the continuation. It means, the interconnection network reconfigures itself.

Figure 2 shows a prototype continuation-aware interconnection architecture. The central component is the cross-bar which interconnects the local memory, the accelerators and the main processor. A local memory is connected to only one accelerator or to the processor at the same time. The cross-bar implements a connection table where this connections are mapped.

Support to continuations is provided by a forward table. When an ACC finishes a computation, it asserts a signal. The cross-bar determines the next component where the memory will be connected and adds this information to the forward table. In parallel, the cross-bar also looks into this same table if there is any local memory waiting for this ACC. Then, the cross-bar modifies the the connection table to reconfigure the interconnection network.
Figure 2: Reconfigurable interconnection architecture. The cross-bar allows switching the local storage use by the ACCs and the processor.

5 Reconfiguration

Partial reconfiguration on the Virtex-II Pro platform is based on Partial Reconfigurable Regions (PRR) and Partial Reconfigurable Modules (PRM) [Doli05]. A prototype base configuration with a single PRR and two different 112,640 bytes PRMs that fit that PRM have been developed. This configuration includes an ICAP peripheral, which does the actual reconfiguration.

Partial reconfiguration is carried out in three stages: first, a block of the partial bitstream is moved from the main memory to the ICAP. Second, some commands are sent to the ICAP. Third, the ICAP performs the actual reconfiguration. This process has to be repeated as many times as 4,096 bytes blocks are in the partial bitstream.

The total time required to reconfigure a PRM is 24.23ms. About 93% of this time is consumed on moving data from the main memory to the ICAP internal buffer. The actual reconfiguration takes only 4.66% of the total time.

Partial reconfiguration takes about 10% of a typical scheduler quantum time on our prototype. However, other applications will use larger PRMs so this percentage will increase. Therefore, the operating system will use reconfiguration-aware scheduling policies. The scheduler should not select a process from the run queue unless all the hardware accelerators it uses are present on the system. It also pre-fetches the hardware accelerators required by the next process in the run queue, so when the process is selected, the accelerators will be already loaded into the logic.

References

