A Process Model for Hardware Modules in Reconfigurable System-on-Chip

Neil W. Bergmann, John A. Williams, Jie Han, Yi Chen

School of ITee, University of Queensland,
Brisbane Q 4072 Australia
{n.bergmann, jwilliams,hanjie,chenyi}@itee.uq.edu.au

Abstract: Reconfigurable System-on-Chip is a powerful method to harness the power of FPGA technology. However, there is a very limited pool of designers who can build hardware-software designs. This paper describes a way to simplify rSoC design, by making hardware coprocessors appear like software processes in a Linux development environment. We explain both the ways that hardware processes are controlled by software ghost processes, and also how hardware and software processes communicate.

1 Background & Motivation

An interesting new alternative to expensive ASIC System-on-Chip design is reconfigurable System-on-Chip (rSoC) [Ly02], where the implementation target is an FPGA. Here a CPU, hardware co-processors, peripheral controllers, cache memories, and external memory controllers are integrated onto a single FPGA. 32-bit softcore processors, such as Xilinx’s Microblaze [Xi06] and Altera’s Nios [Al06], consume less than 10% of the total gate count on a large modern FPGA, so rSoC is now a real design option for many embedded systems.

The current focus of our research group is examination of this new rSoC methodology with particular application to embedded systems. We are interested in system architectures, operating systems, and specific rSOC applications [BW03], [WB05]. This paper looks at our work in integrating hardware and software design, so that hardware appears as similar to software as possible to applications engineers, which we hope will allow a larger pool of designers to access rSoC technology.
2 Embedded Linux

Linux [SF03] is an open-source variant of the UNIX operating system, which is now a common choice for desktop platforms. As even small processors become more powerful, Linux is also gaining significant penetration into the embedded devices arena. Linux provides a well-understood, reliable and royalty-free operating system, which considerably reduces the risk involved in new complex embedded systems designs.

As part of our rSoC architecture [BW03], we have ported Linux to Xilinx’s Microblaze processor, and Linux is already available on PowerPC versions of Xilinx chips. The use of a standard bus (IBM CoreConnect for Xilinx), emergence of both commercial and open-source peripheral controller cores for FPGAs, plus the wide availability of device drivers for Linux means that conventional processor + peripheral systems can be constructed quickly. The ability to debug hardware (rather than the “must-be-right-first-time” constraint of ASICs) further enhances the attractiveness of these rSoC systems.

However, high-performance embedded systems, especially those for computationally complex systems such as multimedia data processing and software-defined radio, typically require multiple general-purpose and special-purpose processors to meet required performance. A major system design issue is how to coordinate control and data communications between multiple parallel processors in a way that is easy for programmers to understand.

3 Linux Processes

In a conventional Linux operating system, there are many different processes running concurrently. In mainframes and desktops, different processes correspond to different applications competing for system resources. However, in many embedded applications, there is often only one application running, and even in the case of multiple applications, these applications are cooperating (eg. an MP3 player on a mobile phone will be muted if an incoming call is received). In embedded systems, a major task of the operating system is to coordinate communications between processes. Linux encourages applications to be composed by joining together existing small programs, with the output from one program being “piped” into the input of another.

The task of communicating between different modules is a central component of any operating system. One of UNIX’s founding principles was the highly modular nature of the computing system, with system utilities providing a large number of computational building blocks (such as list a directory, create, delete, copy or list a file, or match a regular expression). Because many of these programs are written as general tools without knowledge of the many different ways in which they might be used, it is appropriate that each of these programs be run as a separate process, with inter-process communication happening implicitly through input and output streams.
In other complex applications, a single program might be composed as a set of interdependent processes, with explicit communication streams set up between the processes as part of the application execution. Such multi-process implementations allow a program to take advantage of concurrent execution on multiple CPUs within a SMP architecture. For even broader concurrency, applications can be designed with a message passing interface between processes, so that the processes do not even need to execute on the same machine – computer networking delivers messages between processes on remote machines.

Processes provide a very neat model of concurrency, since communications between processes are explicit, and limited to specific mechanisms. Threads are a lightweight form of process, where threads all share access to the same memory space. Whilst these are suitable for SMP architectures where all processors share a single memory space, threads cannot easily be shared amongst remote machines. Additionally, interactions between threads can occur through side effects: one thread can change a variable which interferes with another thread’s operation. If we are interested in splitting an application into hardware and software modules, then it is unlikely (in the general case) that hardware will have access to the same logical address space as a software task, and so threads do not appear to be a suitable model. Rather, we believe that a process, with a separate address space, can be a suitable encapsulation of a hardware or a software task, and the rest of this paper explores this theme.

Of course, the idea of communicating processes as a general model of computation, regardless of whether the processes are hardware or software ones, is widespread. Formalisms such as Hoare’s CSP [Ho85] and Milner’s CCS[Mi89] are well known. However, in this paper we particularly explore the process as implemented in the Linux operating system as our fundamental building block for hardware-software codesign.

An initial problem specification is first decomposed into a number of communicating tasks, represented with a notation such as task graphs. Each task can be mapped to either a software process, or a hardware “process”, and standard Linux IPC mechanisms are used for inter-task communications.

4 Ghost Processes

A difficulty with integrating reconfigurable hardware processes into a conventional software operating system is that operations like starting and stopping the operation of a process, and communications with a hardware process are inherently different to those of communicating with a software process. Previous approaches have typically solved this problem by adding additional capabilities into the underlying operating system kernel, or by adding special run-time software to manage hardware-software interactions [UH04]. This has the great disadvantage of running and maintaining a non-standard operating system. Since one of our aims is to leverage off existing commodity operating systems, we wish to avoid this if possible. Instead, our approach is to make as few changes to the Linux operating system kernel as possible. Instead, changes occur at higher levels of the system, such as device drivers, and server daemons.
To do this, each hardware process has an associated software process, which acts as a proxy or ghost process (so-called because it appears to the operating system as a full process, but its body is elsewhere, in hardware).

To start a hardware process, the operating system simply starts the associated software ghost process and sets up communications with it. The first task of the ghost process is to load the associated reconfigurable hardware module, perhaps by calls to a reconfiguration server. The ghost process similarly sets up communication channels to and from the hardware module. Such channels should not need every transaction to be relayed through the software ghost process – this would be too inefficient, especially for hardware-hardware communications. Stopping a hardware process would similarly be done via a kill signal to the ghost process, which would send appropriate commands to the reconfiguration server. If the ghost process is involved only at process start and end, then for the rest of the time it will simply be an inactive software process – Linux already has lots of them!

In some cases, a single process may need to combine control-based computations suitable for software, and data-flow operations suitable for hardware. If the hardware module is used only for the algorithm kernel, then the ghost process may do some more complex data marshalling.

Overall, the ghost process hides the hardware from the rest of the system, so that other software processes see the hardware as another software component.

5 Design Flow

Our proposed system design flow leverages off well-understood Linux development methodologies. The first step is to implement an application, as represented by a task graph or similar formalism, as a set of communicating software processes on a desktop Linux system. This is a well-understood development environment. In later stages, these processes will be allocated to hardware or software implementations, so it is important that this be kept in mind during the partitioning. Because hardware and software processes are unlikely to have access to the same process states, it is not appropriate to use threads as the units of partitioning. These processes will communicate with standard Linux Inter-Process Communications (IPC) mechanisms, as described in section 6, and at this stage of the design the implementation is a purely software implementation, using standard Linux programming paradigms.

Secondly, the application is moved to the target rSoC Linux system, as the same set of communicating processes using the same set of IPC mechanisms. Most application code can be ported to an embedded Linux platform from a PC with little, if any, modifications except recompilation. In general, the code will run at least an order of magnitude slower on a software embedded Linux platform.
Thirdly, the embedded application can be profiled to find those modules which would benefit most, in terms of speed or power, from migration to hardware. In some cases, profiling might reveal that just a small section of one process requires hardware assistance, in which case a new hardware-software partitioning can be undertaken, and the profiling repeated.

Finally, using profiling results, individual tasks can be moved from software to hardware, preserving the same communications mechanisms. Ideally, software tasks will be unaware of whether they are communicating with a software or hardware task.

6 Linux IPC Architecture

6.1 Inter Process Communications (IPC)

We are interested in a number of different IPC mechanisms used in conventional Linux programs [SF03], to communicate between processes. The mechanisms of most current interest to us include:

- Pipes, which provide a FIFO buffer for communicating between processes.
- Shared memory, which allows a section of memory to be simultaneously accessed by different process. Often, one process writes data to a shared memory, which is then read by another.
- Semaphores, which synchronise access to a shared resource (such as a shared memory).
- Signals, which also allow different processes to synchronise their execution.

A key aim of our current research is to find ways to implement these same IPC mechanisms for hardware-software and hardware-hardware communications. In some cases, a hardware process requires the ghost process executing on the CPU to act as its proxy for certain IPC operations. Our current ideas on hardware communications, which are still under development, are described in the following sections.

6.2 Pipes

The Xilinx Microblaze processor provides special low-overhead FIFO structures called FSL (Fast Simplex Links), and these can provide pipe-like communications to hardware. Reading and writing to the FSLs is done with special instructions which are equivalent to register read and write instructions.
Within the kernel, the FSLs can be presented to the system via device drivers, and can be accessed with normal file read and write instructions. The overheads associated with using these pipes are just the same as the overheads in using software pipes between processes. The FSL interface hardware and device drivers provide appropriate mechanisms to allow blocking or non-blocking reads and writes, so that software processes awaiting input can be scheduled to run once new data is received via the FSL. In other words, the FSL links provide data communications and synchronization.

### 6.3 Shared Memory

Xilinx FPGAs’s provide dual-port memory structures called BRAMs, which are suitable for access by the processor on one port, and a hardware process on the other port. To the processor, this memory simply appears as a normal shared memory structure, and the only overheads are the normal overheads associated with shared memory blocks, which need to be accessed via pointers. As such, shared memory can provide very fast interprocess communications, since access to on-chip BRAMs is typically faster than access to off-chip main memory.

A major difficulty with use of shared memory is the need for the Linux kernel to be informed that it should use this special BRAM memory when a shared memory call is made. In our first set of experiments, this has been done by replacing the calls to the Linux shared memory allocation kernel functions, by custom code which returns direct pointers to the correct memory blocks. Because the Microblaze does not have a memory management unit, this is quite simple because physical memory addresses can be used directly for the specific shared memory blocks. Another problem with shared memory is synchronization – by themselves shared memories don’t provide communicating processes with any signaling that new data is available, unless processes busy-wait on memory locations, which is not an acceptable solution in multiprocessing software systems. Other mechanisms, such as semaphores or signals are needed.

### 6.4 Semaphores

Semaphores are standard data objects in concurrent programming systems which allow synchronization between processes, and mutual exclusion of shared resources. Semaphores are provided as a standard part of the Linux OS. In our current applications, semaphores are primarily used to manage read-write access to a shared memory by consumer-producer processes. Typically, two semaphores are used by each shared memory. One is used by the producer to signal that data is available for processing, and the other is used by the consumer to signal that processing is complete and the data can be overwritten. For such semaphores, there is only one process which signals the semaphore, and one which waits on it, so there is not a problem with attempts to simultaneously update the same semaphore.
For such restricted semaphores, one possibility is to busy wait on a shared memory location. This is suitable for a hardware process, since it cannot do any work until the semaphore is signalled, and our first version of a software-to-hardware semaphore uses this technique. However, the reverse is unacceptable. A software process should not busy wait on a shared memory location. Instead, the hardware process sends an interrupt to signal a semaphore, and the interrupt service routine responds to this interrupt by signalling the appropriate semaphore to the operating system. This automatically makes the waiting software process ready to run. This is used for hardware-to-software semaphore signalling.

We are now working on a second generation implementation of hardware-software semaphores which provides a more consistent implementation. Software-to-hardware semaphores will send a hardware signal (similar to an interrupt) directly to the hardware module, so that all software-hardware signalling uses interrupt-like signals. Initial investigations with semaphores shows that signalling or waiting on a semaphore is an expensive process. An initial experiment with signalling individual words written to a shared memory demonstrated that semaphore signalling costs approximately 100 times the computational cycles of a shared memory read or write. Therefore, such signalling is only appropriate with large blocks of data. When used with 1000 word blocks, the semaphore time overhead becomes a more acceptable 10%.

6.5 Signals

Because we do not use the full power of semaphores (they are not used to provide mutual exclusion in any of our applications), we may be able to use a simpler mechanism of Linux signals for interprocess coordination. Signals consist of a one-way message, similar to an interrupt, and are commonly used for Linux operations such as error conditions. Signals can be useful for operations such as process synchronization.

6.6 Other hardware-software IPC

Whilst our initial investigations are looking at the above IPC mechanisms, there are also other IPC possibilities that we are interested to investigate, such as Linux sockets, RPC (Remote Procedure Call) and MPI (Message Passing Interface). In conventional Linux systems, these mechanisms are typically used to communicate between processes running on different processors, and so are built on top of the computer networking infrastructure of the kernel.

The OCF (Open Crypto Framework) in the OpenBSD operating system, which has also been ported to Linux [Mc06], is based on the notion of a number of operation servers. These could be either hardware or software, with the OCF interface choosing the best currently available server for processing requests.

For hardware modules that communicate with a single software process, then a simple procedure call through a library-based API could be appropriate.
7. Example Application: Video Processing

An example face recognition application has been developed [Ha06] for our architecture. This application uses four steps in an image processing pipeline, which starts with a colour image as a file, and produces either a best match face from a database, or else returns a “no-match”. The image processing pipeline as shown in Figure 1 is used. Shared memory, protected by semaphores, is used between each processing step.

![Figure 1: Processing Pipeline](image)

The application first converts RGB to a greyscale image and applies a median filter to remove noise from the image. Next a template matching algorithm is used to identify a face from the image using a cross-correlation operation. Then a histogram equalization step is used to normalize image intensity. An Eigenfaces algorithm [TP91] (based on Principal Component Analysis) is used to find the best match within the database. The face detection and face recognition algorithms are standard algorithms – the aim of this work was not to develop new image processing techniques but rather to investigate the process-based design methodology.

The system was implemented and results were obtained for several different configurations. The algorithm was first implemented on a desktop PC (CPU- Pentium 4, 2.8GHz, 512MB DDR, Hard Disk-40G, Operating system- Linux Fedora Core 4, Language: C, Compiler: GCC). Next, the same algorithm was moved to the FPGA-based Microblaze processor. The only changes required were some minor adjustments to the shared memory implementation, to use specific dual-ported on-chip shared memory blocks. Finally, various components were moved from software to hardware. Profiling from the PC-based and software Microblaze implementation in Table 1 showed that the template-matching and Eigenfaces modules used the majority of computing time, so hardware-process versions of these two modules were built, and versions of the system with one or both of these modules in hardware were tested.

<table>
<thead>
<tr>
<th>Video processing speeds in frames per second</th>
<th>ALGORITHM MODULES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Module 1</td>
</tr>
<tr>
<td>PC - software</td>
<td>1934</td>
</tr>
<tr>
<td>Microblaze –software</td>
<td>37</td>
</tr>
<tr>
<td>FPGA - hardware</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 1: Performance in equivalent frames per second for different modules and platforms
Performance can be described in equivalent “frames-per-second” for the complete application. Table 2 shows the results for the PC and for various HW/SW configurations on the rSoC.

<table>
<thead>
<tr>
<th>Platform</th>
<th>System performance Frames per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC – all modules in SW</td>
<td>5.5</td>
</tr>
<tr>
<td>Microblaze SW-SW-SW-SW</td>
<td>0.1</td>
</tr>
<tr>
<td>Microblaze SW-HW-SW-SW</td>
<td>0.37</td>
</tr>
<tr>
<td>Microblaze SW-SW-SW-HW</td>
<td>0.12</td>
</tr>
<tr>
<td>Microblaze SW-HW-SW-HW</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 2: Performance of each software module in equivalent frames per second for different platforms

There is clearly a substantial improvement by moving the template matching and Eigenfaces modules both to hardware, since if just one is moved the other remains a software bottleneck. The results show that the performance of a Linux software implementation on a PC can be matched by the performance of a Linux-based implementation on a small embedded processor whose raw software performance is only 1/50 of that of the PC.

8. Discussion

Our premise in this research is to make hardware look as much like software as possible. Modern software systems can be composed from many interacting software modules, without a need to know the internal operations of those modules. Indeed, software re-use encourages modules to be used simply through defined interfaces, such as object-based methods.

In this case, we are investigating the idea that a process, with defined IPC interfaces, is the appropriate computational encapsulation which can be used for mixed hardware-software systems, where individual processes can communicate without knowing whether other cooperating processes are hardware or software. This approach has substantial advantages, in that reusable hardware components can be readily accessed by programmers, without specialist hardware knowledge. Processes can be easily swapped between hardware and software. The video-processing application demonstrates that once an application has been partitioned into separate processes with appropriate inter-process communications mechanisms, then it is relatively easy to perform software-hardware tradeoffs.

The largest potential disadvantage of the process-based approach is performance. The most common reason for moving computations from software to hardware is performance, either speed performance or power performance [SG02].
The software IPC mechanisms tend to incur significant performance penalties by their reliance on calls to the operating system kernel, and this performance penalty may be relatively worse with higher speed hardware.

Our next major research goals are firstly to quantify the magnitude of these penalties, and so identify the characteristics of systems where hardware speedups are not lost in communications overheads; and secondly to optimize the implementation of these “standard” IPC mechanisms between hardware and software so as to reduce the magnitude of the performance penalties and so broaden the range of applications which are suitable for this approach.

9. Acknowledgement

This project is proudly supported by the International Science Linkages programme established under the Australian Government’s innovation statement, Backing Australia’s Ability, and we also acknowledge the financial support of the Australian Research Council.

References

[Al06] Altera, the Leader in Programmable Logic, www.altera.com
[Ha06], Jie Han, “Video Processing with Reconfigurable System-on-Chip”, M.Phil Thesis, University of Queensland (in preparation, to be published mid 2006)