Operating System Support for Difference-Based Partial Hardware Reconfiguration

Tiago de Albuquerque Reis and Antônio Augusto Fröhlich
Laboratory for Software/Hardware Integration
Federal University of Santa Catarina
88040-900 Florianópolis – SC – Brazil
{reis, guto}@lisha.ufsc.br

Abstract

Difference-based partial reconfiguration, although simpler to use by not needing previous floor-planning, has its utilization encouraged only for small changes due to its unpredictable nature. This paper proposes a mechanism to avoid this problem by saving the system global state. Therefore it doesn't matter how the difference-based partial bitstream will affect the hardware configuration. This way, partially reconfigurable system-on-chip development requires less design effort.

1. Introduction

Initially targeted at rapid prototyping, reconfigurable hardware technologies such as FPGAs have found a way into quickly changing markets during the last decade. For many electronic devices, in-the-field upgrading becomes a must in the face of evolving standards, particularly around multimedia and telecommunications. Meanwhile, the scientific community has taken advantage of these technologies to pursue the on-the-fly reconfiguration of those chips, giving birth to a new generation of chameleon devices that can dynamically modify their structure and behavior to match environmental changes. Among the list of devices currently deploying this technology, one can find multifunction personal gadgets, adaptive switches, cognitive radios and a large number of dedicated systems.

From the hardware point of view, modern FPGAs support dynamic partial reconfiguration, so that hardware modules can be kept in non-volatile memory until they are required. Sophisticated tools aid developers in tailoring designs so that logical components can be tracked down to FPGA slices [6]. From the software point of view, however, dynamic hardware reconfiguration is still mostly delegated to applications. Operating system support to dynamic hardware reconfiguration is quite limited and mostly focused on component replacement, without consistently handling side-effects of those changes.

We are still far from a dynamic reconfiguration support system able to identify "which", "when" and "how" hardware and software components which must be reconfigured in order to adapt a computing system to particular application demands. Indeed, considering the dynamically reconfigurable operating systems of the 80s and 90s, like APERTOS [13] and ETHOS [9], whose reconfiguration infrastructure showed very high overhead, one might conclude that the computational cost of such infrastructure is more likely to overwhelm the benefits associated with the technology. Notwithstanding, the current scenario for reconfigurable hardware brings about new opportunities that must be investigated from a more contemporary perspective.

In this paper, we target some major issues around *difference-based partial hardware reconfiguration*. Contrary to module-based reconfiguration, difference-based usually requires fewer resources —thus being more suitable for embedded systems— at the price of worsened predictability. Instead of relying on modules that have been designed, implemented and instantiated for reconfigurability, difference-based reconfiguration simply takes on the differences among bitstreams, thus requiring less memory, less time and, most important, less design effort to perform reconfigurations. Nonetheless, foreseeing impacts on the running software in this scenario becomes infeasible, so this kind of reconfiguration depends on the ability of the running software to adapt itself to unstructured hardware changes.

Within this context, we propose an operating system-level strategy to support difference-based partial hardware reconfiguration. This strategy is modeled around two main concepts: hardware objects that can be dynamically created and deleted; and a mechanism to consistently save and re-

This work was supported by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

store the system global state while manipulating hardware objects. This strategy was implemented and tested in the EPOS System [1] and will be detailed in the next sections.

2. Partial Hardware Reconfiguration

In this work, our vision of system-on-chip is a softcore processor on a FPGA that may control application-specific hardware coprocessors and an application that runs with an operating system support (Figure 1). As the application's needs may change during the execution, it's desirable that these coprocessors can be modified or replaced.

Concerning the hardware, it needs to be reconfigured to apply the desired modifications, which is done by a partial bitstream. Concerning the software, it needs to adapt to the new hardware configuration.

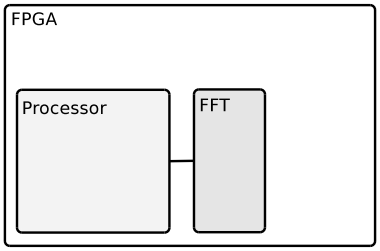

Figure 1. System-on-Chip example

2.1. Partial Bitstreams

There are two methodologies for generating partial bitstreams: module-based and difference-based partial reconfiguration. The first is the most used in research projects because of its modular structuring, allowing the utilization of conventional computer system solutions. This is achieved by defining reconfigurable regions on the FPGA area that can be easily structured in buses or interconnection networks. Based on this idea, some architectures proposed the module switching and reconfigurable space organization analogous to a PC motherboard [4].

Such systems have the advantage of controlling the modules' insertion, removal and substitution through software in a simple way, just by pointing to the partial bitstream's location, the mechanisms can find the appropriated module slot for reconfiguration [2, 10, 4]. But for this gain in the simplicity of utilization and understanding, these systems need a previous floor-planning step and the solutions tend to be focused on a FPGA model due to the reconfiguration region being physically specified on the FPGA. Besides, the bus or interconnection network share valuable design-space with the modules that actually do the work. It's also important to remember that these module slots have a pre-defined size, which may cause design-space fragmentation [8] or, even worse, a module may not fit.

The other way of obtaining partial bitstreams is through difference-based design, which has its use recommended for small changes like LUT equations, BlockRAM contents and I/O standards only [12]. Although it is not intended to reconfigure large amounts of logic, it can still be used for this purpose. This way, it's only necessary to modify the original design to get a partial bitstream without worrying about reconfigurable regions and sizes or previous floor-planning. The disadvantage here is the unpredictability of the reconfiguration place. Because of optimization algorithms executed during the hardware synthesis, a small alteration on the logic may propagate, causing a large difference between designs.

Module-based partial reconfiguration lets us know exactly where slots and modules are, and its modification or substitution may be controlled on-the-fly by the software running on the processor (Figure 2). With difference-based partial reconfiguration the unpredictable changes may happen inside the processor area, thus making it impossible to have any software running over it during the modification. An example would be Figure 1's configuration becoming Figure 3's after a reconfiguration.

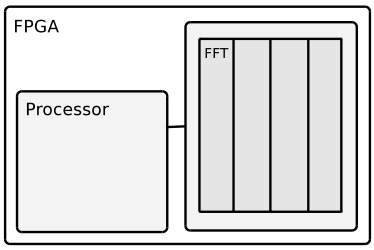

Figure 2. Module-based System-on-Chip

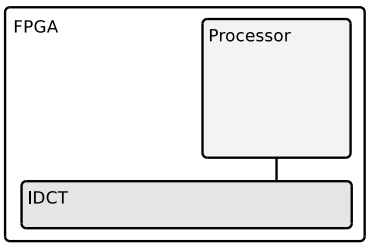

Figure 3. Difference-based System-on-Chip

With a mechanism that allows software to pause and continue its execution and to adapt itself to the hardware's new configuration, partial bitstream development can be simplified by using difference-based bitstream generation. This simplicity comes from the lack of necessity of floor-planning or module/CPU communication structuring. Also,

the design becomes FPGA-model independent and design space is saved. This may result in bigger bitstreams and consequently slower reconfiguration time because it's impossible to predict partial bitstreams sizes.

2.2. Software Impact

A hardware reconfiguration may affect the software when a coprocessor insertion or removal happens. If it's just a modification to a coprocessor that receives data and returns results through a standardized interface, the software probably won't have to change. But this is not always the case.

As the system-on-chip's most important part is the application, it's considered that the application is responsible for informing hardware changes to the operating system, which is done through a software reconfiguration interface.

When a coprocessor insertion happens, the operating system needs to include a driver to communicate with it. So, using the interface, the application informs what coprocessor must now be supported and the operating system initializes the driver for it. The operating system must already have support for the coprocessor.

On the removal, excluded coprocessor drivers need to be freed from memory to avoid unwanted access to it. This is done by informing the operating system that the coprocessor was removed and references to it must be erased.

It is intended that the application may have a way to interfere with drivers, after all these coprocessors are there to be used by the application. This support is on operating system-level to avoid removal of drivers being used elsewhere in the system.

2.3. Related Work

There are several projects aiming at enabling partial reconfiguration through frameworks or complex architectures, most of them using module-based partial reconfiguration. A simple implementation, proposed by Gonzalez [2], defines a reconfigurable area, physically placed on a Xilinx Spartan-3 FPGA, that is connected to a MicroBlaze softcore processor. In this implementation, the reconfigurable area can be loaded with one coprocessor module, which resides on a configuration Flash memory. The execution is controlled by the application running on MicroBlaze.

Ullmann [10] proposed a more sophisticated solution for Virtex II FPGA, four reconfigurable slots are connected to each other and to the outside by a bus, which is controlled by a bus arbiter. A run-time system, running on MicroBlaze, controls the reconfiguration and is responsible for handling the incoming and outgoing messages.

Even more complex, the Erlangen Slot Machine [4] is a complete architecture for reconfigurable computing targeting Virtex II FPGA. It's also slot-based and is concerned with very specific problems like I/O-pin dilemma, intermodule communication and local memory dilemma. It is a multi-FPGA solution, besides the main FPGA, the reconfiguration unit and a crossbar are also FPGA-based.

Such implementations are suitable for large projects with several computation units, solving problems regarding placement, communication with the rest of the system and memory access. But with little or no concern with the software part of the system. More focused on the software, Steiger et al [8] proposes an operating system for reconfigurable embedded systems.

This operating system is partially on software running on a general purpose CPU and partially on hardware inside a reconfigurable area. It is basically composed of a Scheduler and a Placer to manage the hardware tasks. With this functionality, a task can be placed anywhere, not limited by slots, reducing the reconfigurable area fragmentation.

Those implementations are either too simple or complex for a project that is not going to use several hardware coprocessors. Probably, a system with 2 or 3 functional units won't need a bus to communicate with the main processor, just a mapped I/O that almost doesn't impact the logic usage may be enough. Also, as the number of FPGA models increase on the market, porting the system to different models shouldn't be a complex operation. Since synthesizing tools solve all these issues, it is a clever option to let them do it.

3. Operating System Support to Hardware Reconfiguration

To support a hardware reconfiguration without restarting the application, this paper proposes a state saving mechanism. It is responsible for saving relevant information to the software execution for restoring afterwards, like an hibernation process. To avoid problems caused by hardware changes to the software, a software reconfiguration interface was included on the operating system, so the application can inform which drivers should be inserted to and removed from the system. The idea was implemented on the EPOS operating system to the MIPS architecture using the Plasma softcore processor.

3.1. Background

EPOS is a framework for building architecture independent dedicated application support systems [11]. The generated system is composed only by the application and the necessary software and hardware support; this is achieved by a previous domain engineering step. The support comes from the embedded operating systems domain: scheduling politics, synchronization, timing, memory management, interrupt handling and I/O support [5].

Plasma is a 32-bit RISC softcore processor that supports all MIPS I instructions, except the patented unaligned load and store opcodes [7]. It's currently on the third version and is considered stable. The VHDL code implements either a two or three-stage pipeline, interrupt controller, hardware multiplier, timer and UART, SRAM, DDR SDRAM, Ethernet and Flash controllers.

3.2. System Modeling

Our goal is a system, like shown in Figure 4. The hardware is an FPGA with a synthesized softcore processor and application-specific coprocessors, which can be currently on the system or stored elsewhere. At any point the application may request a coprocessor, resulting in a hardware reconfiguration.

The software is an operating system with an application that can reconfigure the software through an operating system interface. This reconfiguration is done by inserting or removing coprocessor drivers, C++ classes that define how the software will communicate with the coprocessor.

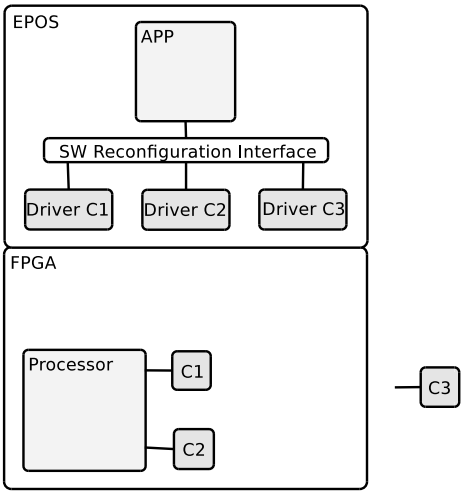

Figure 4. The system's model

3.3. Implementation

Initially, the EPOS memory map was extended to accommodate 35 CPU registers and a 32-bit word for communication between executions, as seen on Figure 5. Based on MIPS' context save assembly routine, a method was created to save the software state. Besides this assembly, modified to save the registers to a specific memory region and signalize that a context save happened, there is also a C++ routine to save the whole memory to non-volatile memory. This is executed only in cases where a system shutdown is required, since a shutdown is not needed for reconfiguration, the RAM contents are not lost.

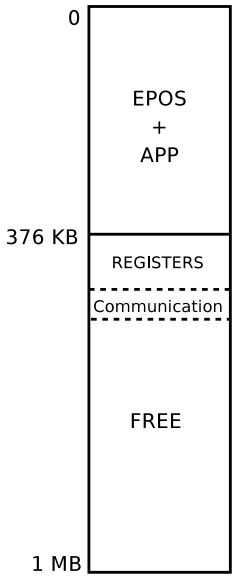

Figure 5. Memory Map

A routine to save the memory contents to an unused part of the RAM was also included, allowing application to perform operations before the reconfiguration that will not appear in the future execution. The routine was included after the context saving. This data must be copied back before the context restore routine.

Immediately after the assembly routine is the returning point to the execution. At this point it should be known if a save or restore just happened. In the second case, the execution continues as if it had never stopped. In the first case it's necessary to copy the RAM contents to the non-volatile memory or the free space on RAM, if desired, and halt the CPU. This context saving method was included on EPOS' MIPS processor mediator as *inline*, to avoid interfering on the execution stack.

After the reconfiguration, Plasma's boot loader is automatically executed, so it has the responsibility of deciding whether the system is being turned on for the first time or a reconfiguration happened and it should restore the software (Figure 6). To achieve that, code was added to make this decision based on the content of the communication word on RAM. If a restore must happen, the boot loader executes a C++ routine to copy the non-volatile memory or RAM data to EPOS' memory space and then an assembly routine to restore the CPU's context.

When the execution flow comes back to the application, it must inform the operating system what changes are expected. A software reconfiguration interface support was created with two methods, insert and remove. The insertion method receives the driver name and instantiates the driver object. This method returns a pointer to this object.

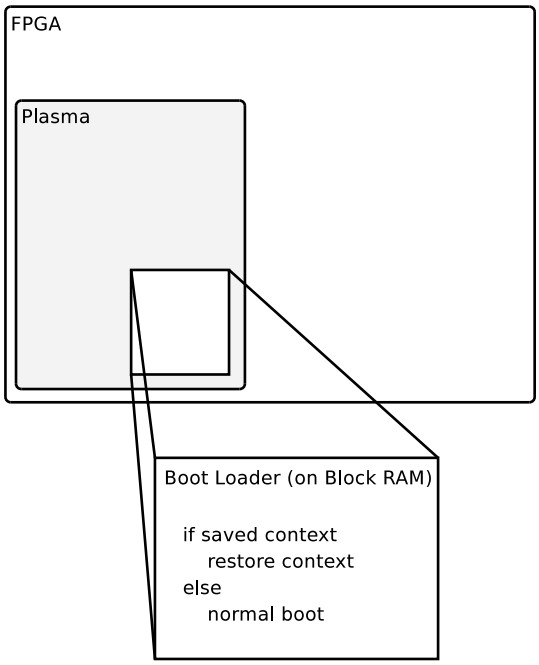

Figure 6. Boot loader

On the removal method, the method receives a pointer to the driver's object and deletes it, returning a status code to indicate if the delete was successful and, if not, why.

3.4. Results and Evaluation

To test this new system functionality a *light-keyboard* synthetic application that, just after starting the execution, invokes the save method and stops to be reconfigured. This reconfiguration doesn't change anything on the design, just forces Plasma to reset and execute the boot loader. After the reconfiguration, immediately the LEDs are turned on when buttons are pressed. This very simple application was used to get a better idea of the support's impact on the EPOS binary size.

This size overhead was measured by compiling EPOS with and without support with the same options, the same was done to the boot loader. The results are shown on Table 1. The EPOS binary increased 280 bytes or 1.49% while the boot loader increased 340 bytes or 7.6%, but still smaller than the Plasma's 8 Kilobytes internal limit.

Table 1 - EPOS and boot loader sizes (in bytes)

	without support	with support
EPOS	18768	19048
Boot loader	4468	4808

The time spent by the save and restore routines on Plasma with 25 MHz clock was also measured. The results are shown on Table 2. To measure the restore time, the second timestamp was read as the first operation after the execution flow returned to the application. The copy of RAM contents increased the operation time 30.3 milliseconds on average. The non-volatile memory copy time wasn't measured because there's no such memory on the used device.

Table 2 - Save and restore execution time (in μs)

	CPU registers only	RAM copying
Save	3.36	32203.4
Restore	26955	55304
Total	26958.36	87507.4

To test the hardware reconfiguration, a FFT coprocessor was added to Plasma. This coprocessor executes a FFT algorithm over data already on memory in a given address, controlled by the application.

After the application tells EPOS that a FFT coprocessor driver should be inserted, it's able control it using high level calls like fft.start() or fft.getresults(). Information like the memory address with data and signaling are masked by the driver.

The size of the difference-based partial bitstream generated was 123 bytes. Its reconfiguration time couldn't be measured because it's done directly between the FPGA and a special configuration Flash memory. This configuration happens when the system is turned on or by pressing a button on the board. This time may be estimated based on the communication speed between the two devices: 33 Mbits/s.

Currently, a video encoder proposed by Husemann [3] is being integrated to Plasma to perform H.264 CIF resolution encoding at 30 frames per second. In his work, Husemann presented an approach to increase the performance of Motion Estimation (ME) algorithm by using two complementary techniques, 4:1 sub-sampling and truncation of two least significant bits of each sample. This gain of performance results in a small quality loss (lower than 0.25 dB). Although small, this noise can be a problem if the original image sequence has low temporal redudancy. So this ME algorithm will only be used to encode sequences that produce acceptabe noise, being reconfigured to a standard algorithm if a low temporal redundancy image sequence is detected.

4. Conclusion

This paper proposed that difference-based partial reconfiguration is viable for system-on-chip development if its bitstream generation unpredictability can be avoided. This is done by a software state save and restore mechanism.

As hardware changes impacts the software running over it, this work also proposes a software reconfiguration in-

terface to allow the application to inform the operating system about the coprocessors entering and leaving the system. This is considered an application job because it is the application which requests coprocessor changes, besides using them directly.

This way, a partially reconfigurable system-on-chip can be designed without intermodule communication structuring such as buses or interconnection networks on the FPGA, simplifying the design, saving design space and making it model-independent.

References

[1] A. A. Fröhlich. *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.

[2] I. Gonzalez, E. Aguayo, and S. Lopez-Buedo. Self-Reconfigurable Embedded Systems on Low-Cost FPGAs. *IEEE Micro*, pages 49–57, 2007.

[3] R. Husemann and V. Roesler. A new approach for high performance motion estimation algorithm implemented in hardware. *Symposium on Integrated Circuits and System Design*, 2007.

[4] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. The Erlangen Slot Machine: A dynamically reconfigurable FPGA-based computer. *The Journal of VLSI Signal Processing*, 47(1):15–31, 2007.

[5] H. Marcondes, A. Junior, L. Wanner, R. Cancian, D. Santos, and A. Fröhlich. EPOS: Um Sistema Operacional Portável para Sistemas Profundamente Embarcados. *Workshop de Sistemas Operacionais*, 2006.

[6] A. Raghavan and P. Sutton. JPG-a partial bitstream generation tool to support partial reconfiguration in virtex FPGAs. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 155–160, 2002.

[7] S. Rhoads. Plasma - Most MIPS I Opcodes. http://www.opencores.org/projects.cgi/web/mips, 2008.

[8] C. Steiger, H. Walder, and M. Platzner. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. *IEEE TRANSACTIONS ON COMPUTERS*, pages 1393–1407, 2004.

[9] C. A. Szyperski. *Insight Ethos: On Object Orientation in Operating Systems*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1992.

[10] M. Ullmann, M. Huebner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[11] L. Wanner and A. Fröhlich. Operating System Support for Wireless Sensor Networks. *Journal of Computer Science*, 2008.

[12] Xilinx. Xilinx Development System Reference Guide, 2006.

[13] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages and Applications*, pages 414–434, Vancouver, Canada, Oct. 1992.