System- and application-level support for runtime hardware reconfiguration on SoC platforms

Dimitris Syrivelis and Spyros Lalis
Department of Computer and Communications Engineering
University of Thessaly, Hellas

Abstract

This paper discusses the design and implementation of a system-level mechanism and corresponding application-level support that enables programs running on a reconfigurable SoC to modify the underlying FPGA at runtime. Applications may request the addition and/or removal of softcore devices at any point in time. Requests are handled in a coordinated way via a separate user-level process that fetches the configuration bistream from an exernal server. System reconfiguration is implemented via a fast suspend-resume mechanism with support for dynamic softcore device address management to achieve flexible device placement on the reconfigurable fabric. Even though our approach does not rely on advanced (and expensive) hardware that supports dynamic partial reconfiguration, the obtained functionality is sufficient for a wide range of application scenarios.

1 Introduction

The technology of field-programmable gate arrays (FPGAs) has the potential to change the way computing systems are being built. While FPGAs are not as fast or energy saving as corresponding ASICs [11] they have the considerable advantage of flexibility: it becomes possible to reconfigure a system not only in terms of software but also in terms of underlying hardware support. In order to exploit this potential one faces challenging issues, such as codesigning hardware and software components, and seamlessly deploying hardware logic on platforms.

In this context it is of particular importance to support a flexible yet robust runtime reconfiguration, allowing for the dynamic downloading and installation of new softcore components. This opens the way for a wide range of possible application scenarios regarding automated system upgrades and customized platform (re)configuration. For example, one may introduce several hardware/software codesigned components that employ customized hardware codecs and accelerators to offload the CPU, boost performance and lower power consumption. The system could also decide which modules fit concurrently on the reconfigurable fabric and select the most appropriate combination, based on the current state and explicitly provided specifications. Even more radical adaptation can be realized on systems with a softcore CPU, in which case it becomes possible to add mechanisms that track CPU usage and create application execution profiles. This information can in turn be exploited to fine-tune specific CPU components as well as to select the most beneficial combination of application-level hardware accelerators. Notably, the efficient online profiling for softcore CPU platforms investigated in [14] could provide the basis for such work.

Runtime reconfiguration in essence translates to *transparency*, i.e. the ability to maintain system and application state so that execution may proceed after (or even during) system reconfiguration without the need for a restart/recovery procedure. Compared to platforms where the FPGA is merely a peripheral of the CPU, this is harder to achieve in a system-on-chip (SoC) because the entire system and application runtime state resides within the reprogrammable fabric itself. Specifically, in order for the runtime state to be kept intact, the FPGA hardware must: (i) support partial reconfiguration; (ii) retain the main softcore logic active while it is being reconfigured; (iii) offer the means for self-controlling the reconfiguration process [22]. For the time being, FPGA vendors provide these features only in expensive product families, and even these devices have constraints in terms of the dynamic partial reconfiguration (DPR) that can be achieved in practice. For this reason approaches that rely on advanced FPGA hardware are not suitable for cheap commodity platforms, or systems with considerable reconfiguration requirements that cannot be implemented given the current limitations of DPR.

In this paper we present work on achieving runtime reconfiguration for SoC platforms featuring a softcore

CPU, without relying on advanced FPGA features. Our goal is to let applications add and remove softcore devices dynamically. The main contributions of this paper are: (1) the introduction of a system-level mechanism and application-level support for reconfiguring a SoC platform at runtime, (2) an implementation that runs on an off-the-shelf embedded device, and (3) a proof-of-concept demo system. We underline that our approach is entirely implemented in software, thus does not achieve the same functionality that is (theoretically) possible via DPR. It nevertheless provides considerable runtime flexibility that is sufficient for most conventional application systems.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 gives an overview of our approach. Section 4 describes a concrete system-level implementation based on the uClinux Xilinx Microblaze port. The corresponding application-level support is described in Section 5. Section 6 presents an extension of our scheme to achieve an integrated handling of softcore devices and physical off-chip peripherals. Section 7 discusses performance related issues. Section 8 presents a proof-of-concept demo setup. Finally, Section 9 concludes the paper.

2 Related Work

Significant efforts have been done on hardware-software codesign to exploit the potential of reconfigurable hardware. Researchers try to build a unified and transparent programming model as well as a standard interface for the integration of hardware accelerators independently of the underlying platform details. A metholology for codesigning applications along with corresponding development tool support is presented in [16]. It proposes a binary level hw/sw partitioner that takes as input a software binary, decompiles it to recover high-level information, determines the regions that should be implemented in hardware (using appropriate profiling information) and generates modified binaries that have the critical code fragments replaced by instructions that access the hardware versions. In [18] a high-level programming model is proposed based on a virtualization layer through which softcore devices can be accessed in a transparent way. Both approaches assume that the underlying platform provides appropriate dynamic reconfiguration support, allowing for arbitrary modules to be added at runtime. This is far from straightforward to achieve in reconfigurable SoC platforms where the CPU itself occupies an area of the reconfigurable fabric.

A lot of research has been conducted on FPGA architectures and development tools for dynamic reconfiguration support. The first step has been to enable partial reconfiguration through corresponding partial bitstream generation tool capabilities [5][9][13], then to change FPGA architecture design so that it can retain the rest of the logic active while it is being partially reconfigured [2][3][15][22]. It must be stressed that dynamic partial reconfiguration (DPR) is still an active field of research. For the time being there are several problems [22] which make it hard or even practically impossible to apply DPR, especially for large and complicated designs such as SoC platforms that feature a softcore CPU: the partially reconfigurable FPGA area placement and size; the external IOB routing constraints that enforce the whole FPGA board layout to be designed with DPR scenarios in mind; and –last but not least– the limited number of Tristate Buffers (TBUFs) that must be used to interconnect dynamically loaded modules with the rest of the logic [22].

Considerable work has been done to support the runtime reconfiguration on SoCs or platforms featuring a separate CPU. This typically concerns mission-specific platforms, or is integrated within a proper (embedded) operating system context. We briefly discuss indicative systems representing a variety of different approaches.

A typical framework for achieving dynamic reconfiguration of a dedicated SoC based on DPR is presented in [7]. Part of the FPGA is used for a fixed softcore control subsystem which communicates with a remote host. The rest of the FPGA is used to place custom logic. Reconfiguration can be triggered by the remote host, at any point in time, which sends the corresponding bistream to the control unit. The bitstream can also be encrypted for security purposes. This approach is suitable for single-application systems.

In [20] runtime DPR support is provided for a SoC featuring a softcore CPU and an embedded operating system. A dedicated kernel-level driver is introduced which provides raw access to FPGA configuration data, allowing it to be modified in an online fashion. This interface can be used by applications or shell scripts to change part of the FPGA at runtime. However, only simple reconfiguration scenarios can be implemented given the limitations of DPR.

A different approach is employed in [12] where the FPGA is pre-partitioned into a fixed number of custom softcore units, and an extra layer is used to provide the abstraction of unit allocation. The program loader distinguishes between software and softcore tasks and dynamically links the former with a free softcore unit. This approach has been implemented in a system with a separate (hardware) CPU. It can be used to eliminate some DPR constraints for a specific platform, but reduces flexibility. This is because it partitions the FPGA to an a priori defined number of nodes that communicate with each other via a fixed interconnection architecture. It is thus impossible to dynamically install arbitrary hardware components that are customized for different applications.

Task-based reconfiguration using a suspend-resume model for a multi-node architecture is presented in [8]. When a node needs to reconfigure, its tasks are suspended and restarted on another node. During this migration, hardware functions may be mapped to software versions thereof, depending on the resources available on the destination node. When reconfiguration completes the original node can be re-assigned its old tasks and proceed with their execution. This approach enables a full reconfiguration of a SoC node, but requires at least two nodes. It has also been implemented using customized hardware and a separate softcore CPU.

Our reconfiguration scheme is geared towards SoC platforms with a softcore CPU and an embedded operating system but does not rely on DPR (merely an off-chip reconfiguration circuit is required). It constitutes a practical option for achieving runtime reconfiguration on top of cheap FPGA systems, without DPR functionality nor any special support from the softcore development toolchain. We employ a suspend-resume model for the entire SoC, but a node can autonomously perform the entire reconfiguration without requiring a second node that must act as its slave. The proposed approach maintains application and operating system state during reconfiguration and lets drivers initialize or even re-establish the state of softcore devices after reconfiguration completes.

Given that we target primarily resource constrained platforms, the hardware configuration bistreams have to be retrieved from a remote server over the Internet. This is similar to the Xilinx Online (Internet Reconfigurable Logic) framework [21], which introduces a remote hardware-update capable facility on top of an operating system. The difference is that in our case it is the user applications that trigger a reconfiguration, not the remote server. It is in fact possible for any process to request a platform reconfiguration at any point in time yet in a controlled way that ensures graceful degradation in case of resource shortage. Moreover, our approach transparently maintains system and application state across reconfigurations.

3 Approach overview

The goal of our work is to support runtime reconfiguration for SoC platforms that feature a sofcore CPU. Specifically, we wish to let applications dynamically add and remove softcore devices that can be accessed via a fast bus or memory mapped I/O. For example, special hardware accelerators, bus drivers and controllers for external hardware, or extra CPU softcore units, could be installed on demand, according to the requirements of the applications running on the system. Again, we stress that this functionality is to be achieved without relying on DPR capable hardware and corresponding partial bitstream generation tools support. The next subsections give an overview of our approach, motivating the various decisions taken.

3.1 The Concept

Our approach is based on a suspend-resume technique, as follows. In a first step, before the actual reconfiguration process begins, the FPGA bitstream corresponding to the new hardware layout for the entire SoC is stored in external memory (we do not address the computation of the bitstream per se). Then, the system saves its current runtime state and initiates FPGA programming. When this completes, the system restarts and control goes to the first stage loader. This checks whether a reconfiguration took place, in which case it overrides the default boot sequence, restores the saved system state and adjusts basic system device information. Finally, prior to resuming normal execution, the device drivers are notified in order to handle the side-effects of FPGA reconfiguration; most notably to initialize / restore the state of the devices. A schematic illustration of this process is given in figure 1.

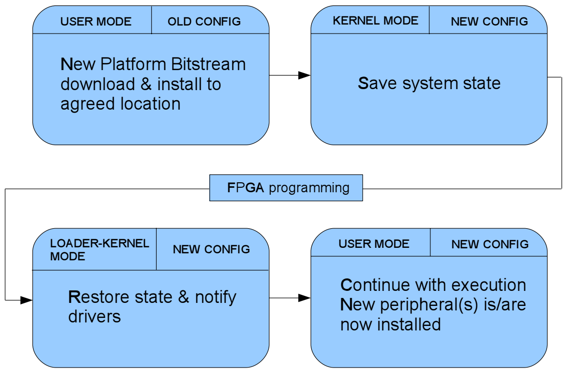
USER MODE
OLD CONFIG
New Platform Bitstream download & install to agreed location
KERNEL MODE
NEW CONFIG
Save system state
FPGA programming
LOADER-KERNEL MODE
NEW CONFIG
Restore state & notify drivers
USER MODE
NEW CONFIG
Continue with execution New peripheral(s) is/are now installed

Figure 1: The main phases of the reconfiguration scheme

Despite the fact that the entire FPGA is programmed from scratch in a conventional fashion, the reconfiguration flexibility provided to the application level is comparable to what would have been possible using techniques that rely on DPR. We note that, in principle, the same scheme could also be used to enable a radical modification of the softcore CPU itself (changing the softcore CPU characteristics according to application workload has been shown to boost performance [6]). However, our approach cannot be applied if part of the FPGA logic is required to remain active during reconfiguration, e.g. for hard real-time applications.

3.2 Device Address Assignment

Given that peripheral devices can be added and removed dynamically, the management of device addresses (more specifically, channel ids for devices that are accessed via a fast bus or specific addresses in the case of memory mapped I/O) becomes a central design issue.

The "obvious" approach of a priori assigning each softcore device a fixed address is not attractive. In the case of fast bus access this would considerably limit the number of devices that can be supported because only a few different channel ids are typically supported in such architectures. This holds to a far lesser extent for memory mapped I/O, but then again the corresponding address range (though large) is not infinite. Thus an artificial upper bound for the number of peripheral devices that can be considered is introduced in this case too. What's probably worse, to avoid conflicts, some central authority or service would be required to assign channel ids and address ranges to each softcore device being (ever) developed.

It is possible to eliminate these drawbacks by assigning addresses dynamically, when a device is first installed in the system. Still, in this case, each time the system reconfigures, the new platform memory layout would have to be computed based on the current configuration and so as to ensure that the addresses of all devices that continue to be a part of the new configuration remain valid. This implies that the new system image must be produced in an online fashion, taking such constraints as input.

To maximize flexibility, we do not require device ids and addresses to remain fixed across system reconfiguration(s). This decouples the process of computing the new system image from any dynamic constraints, other than the type and number of softcore devices that need to be placed on the FPGA. Furthermore, rather than having to compute bitstreams on demand, it becomes possible to exploit databases of pre-fabricated perhaps even highly optimized hardware layouts that could be maintained by device manufacturers.

3.3 Device Access Transparency

Since device addresses are not a priori known and may change in the midst of program execution, additional support is required so that applications are able to access softcore devices in a reliable fashion.

The desired access transparency and safety at the application level could be achieved via a device address translation and checking mechanism, in the spirit of a virtual memory management unit. This would have required a non-trivial modification of the softcore architecture, which is beyond the scope of our current work.

For this reason we adopt a more restricted but yet comparably functional solution, by requiring applications to access peripheral devices via corresponding drivers. Device drivers are a natural way for introducing new hardware functionality in a structured fashion. This also guarantees that applications access softcore devices in an explicit fashion and under system control. Last but not least, device access transparency can still be achieved provided that drivers offer suitable *reconfiguration-transparent* primitives to the application.

4 Implementation of system-level support

This section presents the implementation details of our reconfiguration scheme, for the case of a concrete embedded device, softcore architecture and runtime system. We also discuss issues concerning the development of device drivers in order to deal with the dynamics of reconfiguration.

4.1 Platform

System-level support for our reconfiguration scheme has been integrated into the Microblaze-uClinux kernel port [19] that runs on top of the corresponding MMU-less softcore architecture. Microblaze utilizes Harvard-style separate instruction and data busses which conform to IBM's CoreConnect On-Chip Peripheral Bus standard. Bus arbiters can be automatically instantiated, permitting the instruction and data busses to be tied together in order to create conventional von Neumman-style system architectures.

The host Platform is an Atmark Techno Suzaku [1] (Figure 2) featuring a Xilinx Spartan-3 (XS3C1000) FPGA along with off-chip 16MB SDRAM, 8MB flash, a MAC/PHY core and a configuration controller. The main on-chip softcore modules are the Microblaze core with local memory, Onboard Peripheral Bus, Local Memory Bus, Fast Simplex Links Bus, system timer, interrupt controller, SDRAM controller and an external memory controller.

FPGA configuration is initiated and controlled via Select Map by the external controller and the bitstream is stored in an external flash memory. The reconfiguration procedure can be initiated both by hardware (during power up) and software (write 0x1 to a special register) means. Notably, the power supply is not cut-off during reconfiguration and that the SDRAM data are *not* corrupted because the chip supports self-refresh.

4.2 The Peripheral Device Location Table

As discussed in the previous section, devices may change their addresses after each reconfiguration (with the ex-

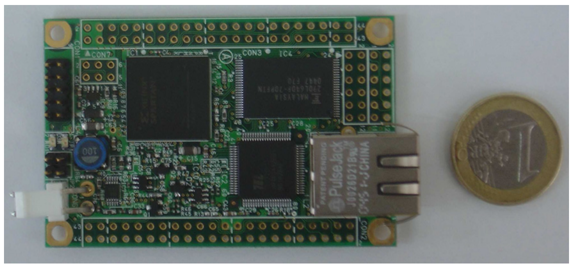

Figure 2: Atmark Techno Suzaku

ception of the execution and data memory controller which are mapped at a specific location because code and data are statically linked to fixed addresses). This means that drivers must be given a mechanism for retrieving the device addresses that are valid at any point in time.

For this purpose the kernel is augmented with the so-called Peripheral Device Location Table (PDLT), an array that contains the addresses of the devices that are available in the current configuration. Each device is assigned a globally agreed offset in the PDLT that is known to program developers. For convenience, we define these offsets based on the well-known major and minor numbers combination of device drivers in the Linux kernel. A PDLT of a few Kilobytes is sufficient to accommodate a large number (thousands) of different devices; of course, the number of devices that can actually co-exist in a system configuration (FPGA image) is limited.

Drivers must be programmed to retrieve the current device base addresses from the corresponding PDLT locations. A zero value implies that the device is not available in the current configuration. The PDLT contents are also exported in user space through the */proc* filesystem so that applications can check the current platform configuration for a particular device.

When a reconfiguration takes place, the contents of the PDLT are updated by the first stage loader during system resume. The loader is programmed into softcore processor local memory and is stored in the configuration bitstream, together with the PDLT entries of the current system layout. Updating the PDLT in kernel space therefore requires a simple copy operation. Since the location of the PDLT depends on the kernel configuration, its start address is stored into a well defined non-volatile memory location so the bootloader can access it.

The bootloader is build using the Board Support Package tool of the Xilinx EDK (Ver 6.3) environment, which generates C #define preprocessor directives with BaseAddresses, thereby making the process of generating the PDLT and storing its contents in the configuration bitstream quite simple. It would also be possible to enrich the Xilinx development environment with scripts that automate this task; though we have not done this.

4.3 Triggering Reconfiguration

Reconfiguration is triggered via a special system call that executes as follows. First, interrupts are masked and the old interrupt mask is stored in a local variable. Then all pending interrupt bottom halves are executed by waking up the linux kernel *ksoftirqd* daemon. The timer bottom half is excluded from this process because it may result in a context switch. Susequently the Interrupt Vector Table and relevant machine registers are stored in a designated area in the kernel image in DRAM (instead of saving the current value of the Instruction Pointer, the address of the resume function is stored). At this point the external controller is triggered to initiate FPGA programming. When this completes, control goes to the bootloader which retrieves the state saved via the system call, calculates the PDLT address in kernel memory, copies its contents from the image, and restores the Interrupt Vector Table and registers. Finally, control returns to the system call context, and the device drivers are notified (see next section) before restoring the interrupt mask and proceeding with the execution of the process that invoked the call. The entire procedure takes less than a second to complete on our platform.

A drawback of letting the system state reside on DRAM is that after a power reset the system reverts to its “original” state and configuration. For this reason, we have implemented a so-called hibernation option. In this case, the system image is copied from DRAM to non-volatile storage (flash) before initiating FPGA programming. The reconfiguration mode (normal vs hibernation) is specified as a parameter of the system call and is stored along with the rest of the runtime state. This information is retrieved upon restart by the bootloader, and if reconfiguration was performed in hibernation mode, the system image is restored into DRAM prior to continuing with the default resume action sequence.

While the hibernation option enhances robustness, it also introduces a significant delay. The total time needed to dump the DRAM image on flash is well above 30 seconds for our platform. Our backup scheme is simple (e.g. the flash is written in polling read mode since all interrupts are disabled) and lacks advanced features, such as checkpointing. Faster non-volatile media and more elaborate I/O operations could reduce this delay, but this could also lead to inconsistencies with respect to the state being saved, in which case more sophisticated hibernation mechanisms [4] may have to be employed.

Notably, our approach is not directly applicable on systems that are interfaced to complex hardware. In this case, a system-wide quiescing of user space processes and kernel thread activity would be required, both prior and after the system suspend sequence so that the state of peripherals can be properly saved and restored, respectively.

4.4 Device Driver Notification

When the system reconfigures, all devices are destroyed, and then re-installed, possibly in a different area within the FPGA fabric; and in a state that most likely requires further initialization before the device becomes operational. As a consequence, even though the device addresses are properly stored in the PDLT, additional device driver specific repair actions may be needed in order to preserve the continuity of device usage at the application level.

For this purpose device drivers may register a so-called *reconfiguration handler*, which is invoked by the kernel after reconfiguration, before returning control to the application. This rountine can be used to perform various housekeeping tasks, such as to initialize the device to a default operating mode, perhaps even restoring it to a previous state, and abort pending operations whose execution may have been compromised due to the FPGA reconfiguration. Device drivers that do not require any initialization/restoration actions need not provide a handler. A simple priority scheme is used to enable the execution of handlers in a certain order.

In our current system prototype we have successfully implemented reconfiguration handlers for the UART, Ethernet, flash, GPIO, interrupt controller and system timer drivers. Since our platform has a softcore timer, each reconfiguration introduces a real-time clock lag (noticeable from an external observer). This error could be corrected by measuring the (fixed) amount of time required for the system to reconfigure, and letting the timer driver increment the system time by this value after each reconfiguration.

4.5 Reconfiguration-Transparent Drivers

Application programs should access devices without caring about reconfigurations that may take place during their execution. Put in other words, device drivers should offer *reconfiguration-transparent* operations. Although the specifics of how to achieve satisfactory functionality are highly device-dependent, we have found the following guidelines to be of use for most cases.

Upon startup the device driver initializes its internal state as well as the device, as usual. When the reconfiguration handler is called, the device is initialized so that it can be properly accessed via the driver operations. Moreover, all processes that have been suspended inside a driver operation are resumed. This implies that the reconfiguration handler must be able to access all driver specific wait queues used by blocking operations, which can be typically achieved via a global wait queue list.

Each driver operation retrieves the device address from the PDLT and uses it to access the device. Notably, the PDLT entry may contain a zero value, indicating that the device is not installed in the current configuration, in which case the driver operation returns an error (e.g. ENODEV). Moreover, configuration parameters and/or additional internal state are recorded so that the device can be properly initialized via the reconfiguration handler. Blocking operations also maintain global state that is used to determine, when they eventually resume, whether a reconfiguration took place in the meantime. If this is not the case, the operation proceeds as usual. Else, the device address is retrieved from the PDLT and the operation can be re-tried.

For the sake of completeness we note that some devices may be *asynchronous*, in which case the effects of driver operations do not necessarily take place within the context of the respective invocations. Moreover, it may be impractical or even impossible for the driver to maintain the device's internal state so that it can be restored. In this case, reconfiguration could lead to state loss, violating transparency. This could be avoided by introducing a locking scheme that allows a driver to block (a requested) reconfiguration until all such operations are acknowledged by the softcore device. We plan to address this issue in a future version of our implementation.

5 Application-level support

The described system-level support enables applications to trigger reconfiguration at any point in time according to their needs. However, it is undesirable to give applications such direct control over the system's resources. One problem is that some applications use devices merely as a performance enhancement option, whereas others may be unable to operate without the requested devices being available. If there are not enough hardware resources to accommodate all devices, precedence should be given to the ones that are vital to application execution. This also implies the removal of devices that are currenty installed but not of vital importance to the applications using them. Another issue is that concurrently running applications will trigger multiple consecutive reconfigurations, even though in some cases the same result could be achieved more efficiently, via a single reconfiguration.

This functionality cannot be achieved if each application is allowed to reconfigure the system while being oblivious to the needs of others. With this motivation we do not allow the reconfiguration call to be invoked from within user processes, and instead introduce a separate mechanism through which reconfiguration is triggered in a coordinated way that ensures maximum overall performance.

5.1 API and background processing

To control reconfiguration according to system-wide policies, applications send device addition and removal requests to a system process with root privileges, called the reconfiguration daemon. The corresponding API (shown below) is implemented as a shared library and communicates with the daemon via unix domain sockets.

```
#define DEV_RMV 0
#define DEV_ADD 1
#define DEV_MND 2

struct dev_req {
   char dev_name[64];
   int actionflags;
};

int device_request(struct *dev_req, \
                            int nofreqs);
```

Applications may use the *device_request* call to issue one or more device addition and/or removal requests. Each request contains the device name (the file name of the corresponding kernel driver) and the action to be performed (the DEV_ADD and DEV_RMV flag is used to specify device addition and removal, respectively). An addition can be specified as mandatory (via the DEV_MND flag) indicating that the device is needed for the application to perform properly.

Requests are processed asynchronously and notification occurs via a SIGRECONF signal. This signal is sent to processes that issued an optional addition request which was satisfied. It is also sent after *each* reconfiguration to processes that requested a mandatory addition, even if this was not satisfied; allowing them to take corrective action or abort. Applications may catch the SIGRECONF signal in a conventional manner, by registering a handler which can determine the presence of the requested device via the */proc* file system.

The reconfiguration daemon maintains a list of requests issued by applications, each carrying the id of the sender process and status information (pending or applied). When a new request arrives, the daemon inserts it in the list and waits for more requests to arrive. If no new request arrives within a given time threshold, the list contents are combined to produce the new platform configuration. In case it is not possible (due to resource constraints) to satisfy all addition requests, these are considered in a first-come-first-serve order, and priority is given to the mandatory requests. When a feasible configuration has been determined, the daemon writes the corresponding FPGA bitstream to the designated memory area and triggers reconfiguration via the system call. It then updates the status of the requests to reflect the current configuration and notifies application processes via a SIGRECONF signal. As a trivial optimization, removal requests do not lead to a reconfiguration unless the list contains at least one pending (and feasible) device addition request.

A process that has issued an addition request may terminate without issuing a corresponding removal request. For this reason the daemon periodically checks (through the */proc* filesystem) the liveness of all processes that issued addition requests. If a process terminates and its addition request has not yet been applied, it is removed, else a corresponding removal request is added to the list to ensure that the device that has been added by this process will be removed. Moreover, at this point it is also convenient for the daemon to remove the kernel drivers, that were used by processes that are no longer alive, since they will no longer be useful in the new configuration.

5.2 Example

This functionality is illustrated in Figure 3 for an indicative scenario. The application processes, softcore devices and request list maintained by the reconfiguration daemon are shown for each step. The eclipses on the top left area denote running processes that issue reconfiguration requests. The installed softcore devices are represented by rectangles in the top right area. The request list is depicted in the bottom part, showing for each entry its process id (upper left), device name (lower left), action type (lower right) and status (upper right, where white and black color stands for pending and applied, respectively). For simplicity, we assume that all addition requests are flagged optional.

We briefly discuss each illustrated step in the following. Initially (a) there are three processes, P1, P2 and P3. At some point in time P1 issues a request to the reconfiguration daemon for the addition of device D1, and P3 issues a request for device D3. Assuming that both devices can be accomodated using the available hardware resources, these are installed via a single reconfiguration (b). Later on (c) P1 terminates (a remove request for D1 is added on its behalf) and P2 issues a request for device D2, leading to a new configuration where D2 is added and D1 is removed (d). Then P4 requests the addition of device D4 (e), but assuming there are not enough hardware resources no reconfiguration takes place. Eventually (f) P2 requests the removal of D2, making it possible to install a new configuration with D4 (g). Finally (h) P4 terminates and, as a result, a remove request is added on

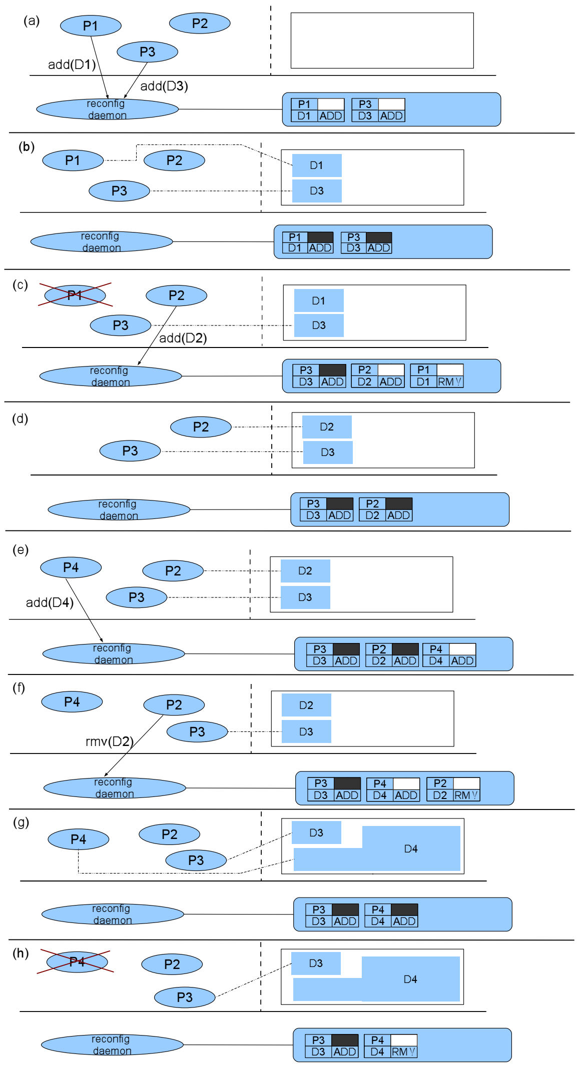
Figure 3: A reconfiguration scenario

its behalf, but no reconfiguration occurs (yet) since there are no pending addition requests to be satisfied.

5.3 Application-level transparency

Applications that rely on basic platform features (e.g CPU, RAM, Ethernet) run safely on our system. They can be executed without any modification, and remain unaffected despite the (repeated) system reconfigurations that may take place at runtime.

If however a program wishes to use a custom softcore device, it must be implemented accordingly. To begin with, it must explicitly issue a device addition request and register a reconfiguration handler that will result in the desired adaptive behavior, e.g. start exploiting the device as soon as it becomes available. Once a device is added to the current configuration, transparency is achieved if (a) the device is mandatory and (b) it is accessed via a reconfiguration-transparent device driver. Else, a program may fail to access the device due to its relocation or removal from the FPGA; and should be prepared to deal with this case in a robust way.

We note that the addition of a device must be explicitly requested, even if it already exists in the current platform configuration. This is to let the reconfiguration daemon keep a correct reference count for each device. Also, given that device references are kept in user space, these are not inherited from parent to child processes and kernel-level threads. Thus separate device addition requests must be issued on behalf of each execution context, independently of whether this has already been done by the parent.

5.4 Remote bitstream fetch

To achieve a clean separation of concerns the FPGA bistream of the desired platform configuration is provided by a separate process, called the bistream server. The communication between the reconfiguration daemon and the bistream server is over TCP/IP, hence the server can be conveniently placed on a remote host; which is particularly useful in the case of resource constrained platforms.

When the daemon wishes to reconfigure the system, it sends to the bitstream server the list of optional and mandatory devices that may need to remain or become available, respectively. Based on this input, the server replies with the configuration that can be implemented given the amount of hardware resources available (perhaps depending on other limitations as well) and a corresponding bitstream url. The daemon then downloads the bitstream from the server using the netflash utility [17].

The underlying working assumption is that the bitstream server knows the host platform details and has access to a database of pre-fabricated configuration bitstreams. For example, it could be a platform vendor service responsible for providing fully tested and highly optimized configurations. In principle, it would also be possible to integrate the bitstream server functionality with the hardware development toolchain so as to be able to synthesize new platform configurations on demand. Given that this task is quite time consuming (10 minutes approx. on a PC), this is not a very attractive solution for the time being.

6 Support for off-chip peripherals

Functional units requested by applications may require not only a softcore module but also additional off-chip peripherals, e.g. a sensor. In this particular case it makes no sense to add the sofcore module unless the peripheral is also physically connected to the system. Wishing to unify the softcore and physical aspect of peripherals, we extended the reconfiguration mechanism to handle application requests and the asynchronous event of a peripheral plug-in in an integrated fashion.

6.1 The hotplug detector

To accomplish this we introduce a special softcore device, the so-called hotplug detector. Its role is to capture the fact that external hardware has been connected to or disconnected from the system, respectively. In our prototype we allow up to 4 peripherals to be simultaneously plugged on the Suzaku board.

The corresponding module amounts to 1% of our FPGA resources. It is hooked on the Microblaze On-Chip Peripheral bus and is accessed through 4 memory mapped registers. Eight of the least significant bits of each register are connected to external I/O FPGA pins while the rest are grounded. We assume that an off-chip peripheral will be attached to the pins of a register, and will redirect Vcc and Gnd to form a code that uniquely identifies it (in our implementation we require this to be the major number of the corresponding kernel driver). We also expect Vcc to be redirected to the peripheral interrupt line which is connected to an external I/O pin as well. An illustrative schematic is shown in Figure 4.

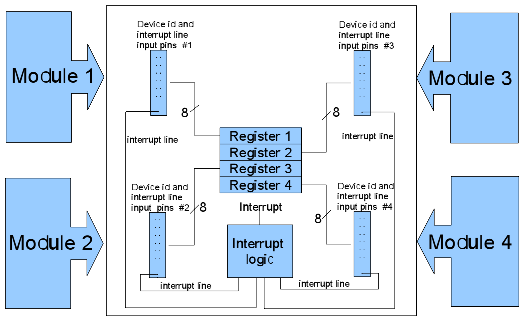
Module 1
Module 2
Module 3
Module 4
Device id and interrupt line input pins #1
Device id and interrupt line input pins #2
Device id and interrupt line input pins #3
Device id and interrupt line input pins #4
8
8
8
8
interrupt line
interrupt line
interrupt line
interrupt line
Register 1
Register 2
Register 3
Register 4
Interrupt
Interrupt logic

Figure 4: The hotplug detector high-level schematic

Access to the hotplug detector is provided via a character device driver, which supports the standard file operations interface as well as the *select* and *poll* system calls. The driver also registers an interrupt handler that is invoked when an off-chip peripheral is connected to and disconnected from the interface pins. The read operation is blocking and waits for an interrupt to occur.

The memory mapped registers have the value of zero when no peripheral is hooked and the driver remembers previous register states so it can determine whether a peripheral is connected or disconnected. When an interrupt occurs the Interrupt Service Routine scans all registers to determine which one has changed value, reads its contents and unblocks any waiting processes. Subsequent read operations then return the device id, the register number of the pin region, and a value (zero or one) indicating whether the device is connected to or disconnected from the system. To discover peripherals that have been hooked on the platform before starting the reconfiguration daemon (or powering up the system), the hotplug registers are examined via the ioctl system call when the daemon starts.

In our implementation we tried to avoid a complex hardware design that consumes a significant amount of resources. This is because we want to keep the hotplug detector as a basic platform feature that will be included in every configuration. By keeping state information in the device driver, rather than the hardware logic, we are also able to achieve reconfiguration transparency for the hotplug driver. Admittedly, using a 8 pin interface for device identification is a waste of external I/O resources. In principle one may use just 1 pin but this requires a more sophisticated communication protocol; see[10] for a similar DPR-based implementation.

6.2 Unified reconfiguration handling

The hotplug detector is accessed by the reconfiguration daemon to receive information about the addition and respectively removal of an off-chip peripheral. This information is then sent to the bitstream server, along with the contents of the request list.

It is the responsibility of the bitstream server to determine the possible layouts that may be installed on the FPGA, also taking into account the dependencies between softcore devices and off-chip peripherals. More specifically, a pending addition request for a softcore device that requires an off-chip peripheral is considered only if the peripheral is connected to the system. This decision naturally belongs to the bitstream server (rather than the reconfiguration daemon) since in our model it is the former that has access to platform-specific implementation data.

Once a mandatory softcore device that relies on a peripheral is installed, it is not automatically removed even if the required peripheral is disconnected from the system. In this case, the application will simply receive an error from the corresponding device driver. It may then explicitly request the removal of the device or terminate. On the other hand, the application may wish to keep the softcore device installed, expecting the peripheral to be re-connected to the system; this may involve out-of-the-loop information (e.g. the user's intention) which is not available to the low-level system mechanisms such as the reconfiguration handler.

6.3 Example

Figure 5 gives a scenario illustrating this additional functionality (employing the same visual metaphors as in the previous example). Note that the hotplug detector module is considered to be already installed as a mandatory device.

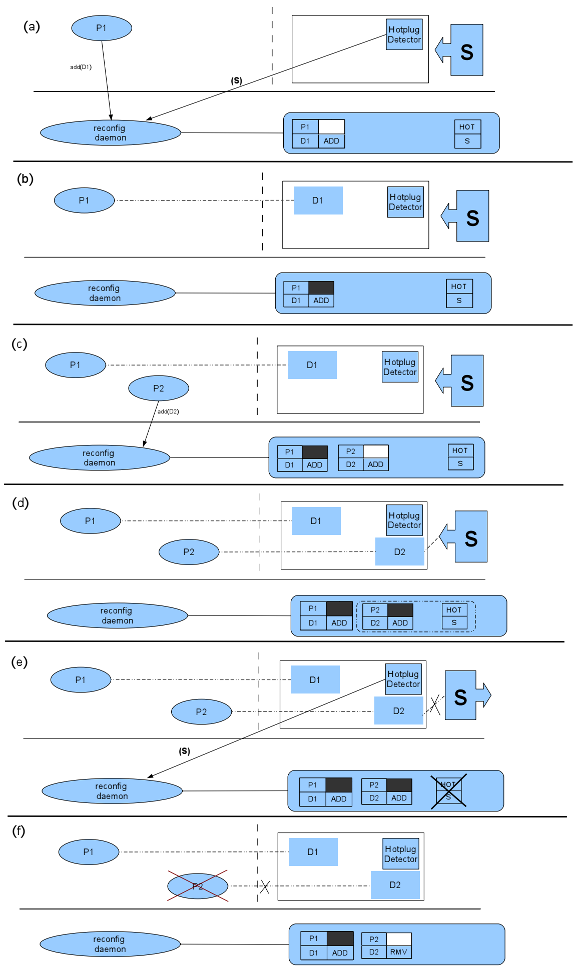

Figure 5: A reconfiguration scenario with hotplug event

Initially (a) process P1 requests the addition of device D1. At the same time, the user plugs in sensor S (that can be used only via device D2). The hotplug detector informs the reconfiguration daemon, which in turn adds a corresponding presence entry. The system then reconfigures and D1 is installed (b). After a while (c) a new process P2 starts, and requests the addition of device D2 (which requires the S). Given that S is already connected to the system, the system will reconfigure and D2 will be installed (d). Later on (e) S is disconnected from the system, leading to a corresponding update of the reconfiguration daemon, but D2 remains installed. Finally (f) P2 receives an error from D2 (which tries to access S without success) and terminates (a removal request for D2 is added on its behalf).

7 Performance considerations

Since applications exploit softcore devices via kernel device drivers, each access operation comes at the cost of a system call. Each driver operation must also retrieve the base address of the device via the PDLT. This amounts to one extra instruction compared to the code that would have been generated using a fixed address scheme. A second instruction is needed for each different base-relative address used within a driver operation. We believe that this overhead is reasonable given that our approach is implemented in software, without requiring a modification of the softcore CPU architecture.

In our current implementation platform and setup, switching to a new configuration takes about 12 seconds to complete from the moment a process issues a device addition request (assuming the reconfiguration daemon does not wait for other requests to arrive). This delay is decomposed as follows. The communication with the bistream server including the download of the FPGA bitstream takes about 1.5 seconds (over a 10 Mbit Ethernet). Writing the bitstream on flash takes about 9 seconds. Finally, performing the reconfiguration system call (saving state, programming the FPGA, restarting and notifying drivers) takes less than 1 second. It is important to note that application processes continue their (concurrent) execution during this amount of time, except for the last step, i.e. the execution of the reconfiguration system call.

These figures are obviously specific to our implementation platform. The FPGA programming delay, for example, could grow for larger platforms; though these also tend to support higher programming speeds. What is more important, if it were possible to program the FPGA directly from DRAM (rather than requiring the bistream to be copied on flash), the total reconfiguration delay (including the bitstream download from the network) could shrink to about 2-3 seconds.

8 Demonstration

To demonstrate our implementation we have developed a simple environment that comprises two different applications: a mandelbrot calculation and an audio signal monitor. Both applications are structured in the form of a client-server pair. The servers run on the Suzaku board as convetional application processes. The clients run on a PC providing a graphical user interface for controlling the servers. Client-server communication is over TCP/IP and a LAN to which the Suzaku is connected via its Ethernet adapter. A schematic of the various components is given in Figure 6. A picture of the Suzaku board setup is shown in Figure 7.

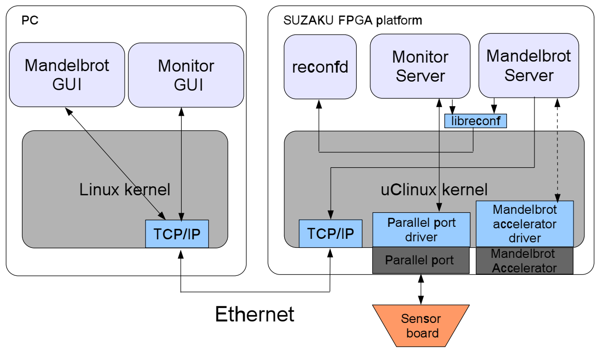

Figure 6: Demo Setup

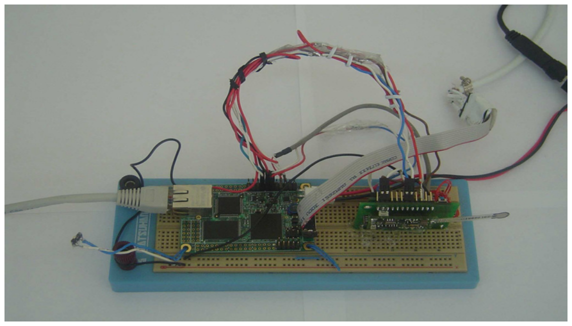

Figure 7: Suzaku board setup

8.1 The mandelbrot application

The mandelbrot client is used to send the computation parameters to the server and display the results produced. Moreover, it can be used to request the addition or removal of an accelerator module that is exploited by the server to perform the computation faster.

The server waits for incoming requests, performs the computation and sends the results back to the client. It is initially in a default state, performing the computation without relying on the accelerator module. When it receives a client command to add the accelerator, it issues a corresponding request, and enters an optimized mode of computation as soon as the softcore module is added to the system. Similarly, the server issues a removal request when it receives a corresponding client command. It continues however to opportunistically exploit the accelerator until the driver returns an error; indicating that the module has been actually removed.

As expected, hardware acceleration boosts performance both in terms of time and power consumption. Notably, our softcore CPU does not have a floating-point unit, hence the software version of mandelbrot uses the integer-based floating point operations of the gcc library. Figure 8 depicts the average energy needed to perform a certain computation for the software-only versus the hardware accelerated version. The average power consumption of the system in idle mode is used as a reference. Specifically, the hardware-based version requires about 7,6% of the energy the software version needs and is 7,33 times faster (labels 2 and 4 vs 1 and 5). The former version includes the extra delay and power consumption for downloading the bistream and reconfiguring the system *before* initiating the computation (labels 2 and 3). The hardware-based version requires about 5,6% of the power that the software version consumed and is 10 times faster in case the accelerator is already installed (labels 3 and 4).

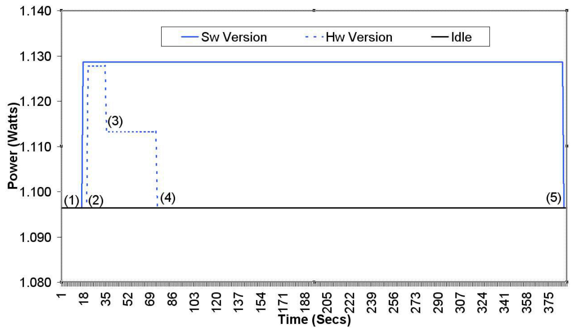

Figure 8: Suzaku power consumption diagram

8.2 The sensor monitor application

The monitor client is used to start / stop the sensing activity of the server and to display the values received. The server starts in an idle state. When it receives a start command, it launches a child process that requests the addition of a sensor specific softcore module. If the request is not satisfied the process terminates and the server sends back a failure message. Else, the child process starts reading sensor values and forwards them to the client. The child process can be terminated at any point in time via a corresponding client command.

The softcore device requested by the child process cannot function properly without a corresponding sensor being attached to the Suzaku board. For this reason the addition request is not considered unless the appropriate sensor is connected (by hand) to the board. When the sensor is disconnected from the board, the child process receives a driver error and terminates, making it possible for the corresponding softcore module to be removed.

8.3 Configuration scenarios

We have configured the bitstream server to deliver four bitstream files that have been pre-built for this particular setup: (1) the base system configuration, (2) the base configuration plus the mandelbrot accelerator module, (3) the base configuration plus the sensor access module, and (4) the base configuration plus the mandelbrot accelerator and the sensor access modules.

The system is started with the first configuration. From that point onwards, any other configuration can be dynamically installed, depending on the sequence of requests issued by the mandelbrot and monitor applications (via their respective clients) as well as the physical presence of the sensor. Given that reconfiguration does not take place solely for the purpose of device removal, the system will stop reconfiguring once configuration (4) has been installed, because in this case all (future) requests issued by these applications are trivially satisfied.

9 Conclusion

In this paper we have presented the design and implementation of system-level mechanisms and application-level support for the dynamic addition and removal of softcore devices on a reconfigurable SoC featuring a softcore CPU and embedded operating system. This functionality is achieved without relying on DPR. Although the entire FPGA is re-programmed from scratch when a reconfiguration takes place, system, application and relevant device state can be maintained to a large degree, thereby achieving satisfactory transparency.

Application programming support comes in the form of a library for issuing device addition/removal requests that are asynhcronously acknowledged via signals. Reconfiguration is triggered via a user-level process that collects and handles application requests in a bundled fashion. The configuration bistream is downloaded from a remote server over the network, making it possible to support resource constrained systems with communication capability. In case of resource scarcity, priority is given to critical devices. Once the bitsream is saved in the designated memory area for programming the FPGA, reconfiguration (during which application processes remain frozen) takes less than a second to complete in our current platform.

Finally, we have considered softcore devices that rely on off-chip peripherals, and have extended our implementation to take such device addition requests into account only if the required peripheral is physically connected to the system.

10 Acknowledgments

This work was supported by the Greek General Secretariat for Research and Technology through the 3rd Community Support Programme; Measure 8.3: Research and Technology Human Resources; Action 8.3.1: "Reinforcement Programme of Human Research Manpower - PENED2003".

11 Availability

More information about this work and uClinux patches are all available at *http://www.inf.uth.gr/vss*

References

[1] ATMARK TECHNO INC. *Suzaku Series.* http://www.atmark-techno.com/en/products/suzaku.

[2] ATMEL, I. *Field Programmable System Level Integrated Circuits (FPSLIC)(2002).* http://www.atmel.com/products/FPSLIC/.

[3] BLODGET, B., JAMES-ROXBY, P., KELLER, E., MCMILLAN, S., AND SUNDARARAJAN, P. A Self-Reconfiguring Platform. In *Proceedings of Field Programmable Logic and Applications* (2003), pp. 565–574.

[4] CUNNINGHAM, N. *Suspend2 project.* http://www.suspend2.net/.

[5] DYER, M., PLESSL, C., AND PLATZNER, M. Partially reconfigurable cores for xilinx virtex. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications* (London, UK, 2002), Springer-Verlag, pp. 292–301.

[6] FLETCHER, B. FPGA Embedded Processors: Revealing True System Performance. In *Embedded Systems Conference San Fransisco* (2005), no. ETP-367.

[7] FONG, R. J., HARPER, S. J., AND ATHANAS, P. M. A versatile framework for fpga field updates: An application of partial self-reconfiguation. In *IEEE International Workshop on Rapid System Prototyping* (2003), pp. 117–123.

[8] HAUBELT, C., KOCH, D., AND TEICH, J. Basic OS Support for Distributed Reconfigurable Hardware. In *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation* (Samos, Greece, July 2003), pp. 18–22.

[9] HORTA, E. L., AND LOCKWOOD, J. W. Parbit: A tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (fpgas). Tech. Rep. WUCS-01-13, Washington University Department of Computer Science, 2001.

[10] LU Y, BERGMANN N, W. J. Dynamic loading of peripherals on reconfigurable system-on-chip. In *SPIE Microelectronics: Design, Technology, and Packaging II* (2005), vol. 6035.

[11] MATSUMOTO, Y., AND MASAKI, A. Speed improvement of FPGA by mixing multiple-gate-width routing switches. In *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* (2005), pp. 14–22.

[12] NOLLET, V., MIGNOLET, J.-Y., BARTIC, A., VERKEST, D., VERNALDE, S., AND LAUWEREINS, R. Hierarchical run-time reconfiguration managed by an operating system for reconfigurable systems. In *Engineering of Reconfigurable Systems and Algorithms* (2003), pp. 81–87.

[13] RAGHAVAN, A. K., AND SUTTON, P. Jpg - a partial bitstream generation tool to support partial reconfiguration in virtex fpgas. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2002), IEEE Computer Society, p. 192.

[14] SHANNON, L., AND CHOW, P. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays* (2004), pp. 190–199.

[15] SIDHU, R. P. S., AND PRASANNA, V. K. Efficient metacomputation using self-reconfiguration. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications* (London, UK, 2002), Springer-Verlag, pp. 698–709.

[16] STITT, G., VAHID, F., MCGREGOR, G., AND EINLOTH, B. Hardware/software partitioning of software binaries: a case study of h.264 decode. In *CODES+ISSS* (2005), pp. 285–290.

[17] UNGERER, G. netflash utility. http://docs.linux.com, Oct. 2002.

[18] VULETIC, M., POZZI, L., AND IENNE, P. Programming transparency and portable hardware interfacing: Towards general-purpose reconfigurable computing. In *ASAP* (2004), pp. 339–351.

[19] WILLIAMS, J. *The Microblaze-uClinux kernel port Project.* http://www.itee.uq.edu.au/ jwilliams/mblaze-uclinux/.

[20] WILLIAMS, J., AND BERGMANN, N. Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms* (2004), pp. 163–169.

[21] XILINX INC. *Architecting Systems for Upgradability with IRL*, 2001. Aplication Note XAPP412.

[22] XILINX INC. *Two Flows for Partial Reconfiguration: Module Based or Difference Based.*, 2003. Aplication Note XAPP290.