UNIVERSITY OF THESSALY

Exploiting Reconfigurable Heterogenous Parallel Architectures in a Multitasking Context: A Systems Approach

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in
Computer Science
by
Dimitris Syrivelis
April 2009

Dissertation Committee:

Dr. Spyros Lalis, Chairperson
Dr. Georgios Stamoulis
Dr. Catherine Houstis

The Dissertation of Dimitris Syrivelis is approved by:

Committee Chairperson

University of Thessaly

ACKNOWLEDGMENTS

I would like to express my gratitude to all those who made this dissertation possible.

To all

ABSTRACT OF THE DISSERTATION

Exploiting Reconfigurable Heterogenous Parallel Architectures in a Multitasking Context:
A Systems Approach
by
Dimitris Syrivelis

Doctor of Philosophy, Graduate Program in Computer Engineering
University of Thessaly, April 2009
Dr. Spyros Lalis, Chairperson

In the recent years, the continuous performance increase of the sequential execution on single CPU systems is facing an upper bound because it primarily relied on the respective circuit operating frequency improvement, which has already reached its limits. This low-level performance bottleneck caused a chain reaction to the above abstraction layers and practically changed the way computing systems are being built. Nowadays, realizing parallelism at all the design levels of a computing platform is the main goal of the respective domain research efforts. Application developers need to improve their skills and take into account architecture-level platform details to decide the optimal application partitioning, while respective re-targetable toolchains have been developed to automate tasks and abstract complexity to the extent possible. On the other hand, traditional general-purpose operating system support and related concepts have not been particularly reconsidered in the emerging broader context of parallel applications, tools and architectures but primarily focus on the efficient task scheduling on shared memory homogeneous and symmetric multicore systems of limited scalability. In this dissertation, we introduce new concepts at the operating system-level to take advantage of the runtime reconfiguration of hardware to exploit its benefits under a general purpose context. Regarding application development, we demonstrate the effectiveness of two different application development framework approaches that are tightly integrated with novel operating system support for optimal execution on many-element arrays. We outline how these contributions allow the operating system to efficiently distribute any type of platform resources, deal with performance asymmetry and load balance, at runtime, parallel applications that execute concurrently on the emerging parallel reconfigurable platforms. We have implemented prototypes of every proposed concept and we report the results of real life experiments.

Contents

Chapter 1

Introduction

Reaching the silicon frequency barrier has resulted in a bloom of radically different parallel computing platform designs that are now becoming the mainstream approach to achieve better performance. Recently proposed parallel platform configurations are very diverse, and their only common characteristic is the synergistic use of many tightly-coupled processing elements like traditional instruction set processors, graphics processors, application specific circuits and reconfigurable hardware. While the previous platform designs allowed a rather clean separation of concerns and minimal interaction between hardware designers, operating system and application developers, nowadays, efficient execution and flexible resource utilization requires a good understanding of various cross-layer details.

Obviously, boosting application development and execution performance on parallel platforms is now the main concern of many research efforts. To that end, most recently proposed frameworks support the development of applications based on the assumption that *all* the required target resources will be *dedicated* to the developed program throughout its execution. This seems a reasonable choice to make mainly because the traditional time-sharing technique that was used to achieve multitasking on single- or limited multi-CPU systems (shared memory, up to 16 cores) cannot be used efficiently on the emerging massively parallel platforms. Typical examples are parallel processor arrays, reconfigurable hardware systems-on-chip and multi-general purpose GPU platforms, that have a few hundreds of processing elements (not necessarily ISA processors). Interconnections between the processing elements are dedicated direct links and each one has private local memory to achieve independent execution. Indeed, for the time being, the operating system support on such platforms usually runs on a master CPU and is used to deploy highly optimized applications which have been developed to occupy a *fixed* amount of resources. If another resource demanding application arrives at the system, and it cannot be given the required amount of resources, it will have to wait for the previous one to finish.

Emerging fine-grained reconfigurable hardware technology introduces additional challenges to the parallel platform development tools and runtime support. It is now possible for the same hardware platform to form different hardware resource configurations, even at runtime, that can acceptably perform the execution of radically different applications. Levels of reconfigurability may vary from fully reconfigurable "soft" FPGA-based architectures, that can be reconfigured to form different accelerator circuits, to Instruction-Set Processor (ISP) Processor Array designs that feature reconfigurable dedicated interconnections. As the technology and the respective tools matured, commercial systems appeared that employed hardware reconfigurability to primarily support the so-called "softcore" platform updates, exactly as it happens with software updates. Nowadays, reconfigurable platforms can be further customized independently before each application execution. This process takes place in a static manner, before application loading, and can be repeated for the next application after the current one finishes execution. Note that with the term "Reconfigurable Parallel" we are referring to all platforms that support a form of h/w supported parallelism and h/w reconfigurability at any level, and do not explicitly imply the Field Programmable Gate Array (FPGA) platforms. For example many core platforms with reconfigurable interconnections known as Massively Parallel Processor Arrays (MPPAs), are also considered as Reconfigurable Parallel Processor Array (RPPA) platforms.

Building general purpose multitasking systems that can take full advantage of parallel reconfigurable hybrid platform capabilities will allow the end-user to effectively run different and not necessarily a priori known tasks on the same device for concurrent execution. Execution performance will not be compromised if the hardware reconfigurability can be exploited at runtime and the platform can be optimally adapted to carry out a given workload. To that end, in case of resource shortage, one could consider the *dynamic distribution* of available resources between concurrently executing applications. In other words, if a new application arrives, currently deployed applications could be re-configured to free some resources that can be exploited by it. Similarly, when an application finishes, the released resources can be reassigned to speedup the rest of the workload.

We envision a more radical exploitation of hardware reconfigurability. Current approaches at all design levels of Reconfigurable Parallel (RP) systems assume that the target platform is meant for dedicated execution and will never be used in a multitasking context. Even in the cases of dedicated systems where two or more applications are to be executed concurrently, the resources are statically assigned to the participating applications during design time. Accordingly, in a respective usage scenario, if only one of the aforementioned predefined applications needs to be executed, it won't be able to use more resources than the statically assigned amount, despite the fact that the rest are not occupied. Ideally, on a system that features hardware support to exploit parallelism, the applications should be able to take advantage of the available resources to improve execution, whenever possible. With this feature, RP systems would be capable to efficiently accelerate typical general purpose workload.

In the context of this work, we attempted the integration of the runtime hardware reconfigurability of platforms that are currently considered for dedicated execution with the concept of dynamic resource distribution and the mainstream multitask computing. More specifically, we introduce appropriate software concepts to all abstraction levels of a general purpose multitasking system so that it can be realized on hybrid Reconfigurable Parallel platforms (RP) that are currently being used in a dedicated context. More specifically we propose: i) a device driver model and low-level interaction with hardware, ii) respective kernel-level mechanisms, iii) user-level reconfiguration control infrastructures, and iv) a programming framework for Reconfigurable Parallel Processor Arrays (RPPAs) that includes two different approaches and models to address the new challenges. We have also prototyped several multi-element hybrid softcore architectures using the Xilinx Platform Studio suite, which include custom hardware modifications that improve the Hw/Sw components synergy. Finally, we developed prototypes of every proposed concept and tools and we deployed them on two different FPGA-based platforms with 5 customized implementations of different well-known applications that were executed according to a variety of workload scenarios.

As it happens with any other computing platform, RP systems are not suitable for the acceleration of all types of applications. As a rule of thumb, an application can be accelerated via parallel execution only if the respective computation type inherently contains parallelism. In the sections that follow we present the application domain that is suitable for RP systems and we discuss the considerations that are introduced by multitasking at the programming framework level, the operating system services and kernel support and the respective hardware platform issues.

1.1 Application Domain

All the application computations that contain parallelism can take advantage of RP systems. More specifically, there are two basic categories of parallelism: spatial and temporal.

In the first case that is depicted in figure 1.1A belong all applications where the same computation or different computations can be applied on different blocks of data independently, this type is also known as data parallelism. Taking advantage of the physical presence of more than one instances of the same or different resource types, target platform can process larger chucks of data at the same time and the respective execution is accelerated compared to the single resource operation.

In the second case that is depicted in figure 1.1B, the application can be divided to stages of execution that are assigned to different resource instances. Each stage can perform processing independently as soon as data are received from the previous one. In these configurations, each stage delivers processed data only to the next one and therefore the computation forms a so-called pipeline. Since data are delivered to the next stage after the current stage finishes processing, acceleration is realized after initial network loading is completed and every stage has data to work on concurrently. This acceleration can be realized in streaming applications.

Finally there are application computations which can take advantage of a combination of the aforementioned parallelism categories. In this case depicted in figure 1.1C the interconnected stages of application execution form a graph that is similar to the Kahn process network [1]. The target platform dedicated interconnections between the available processing elements can be reconfigured to satisfy the computation parallelization needs.

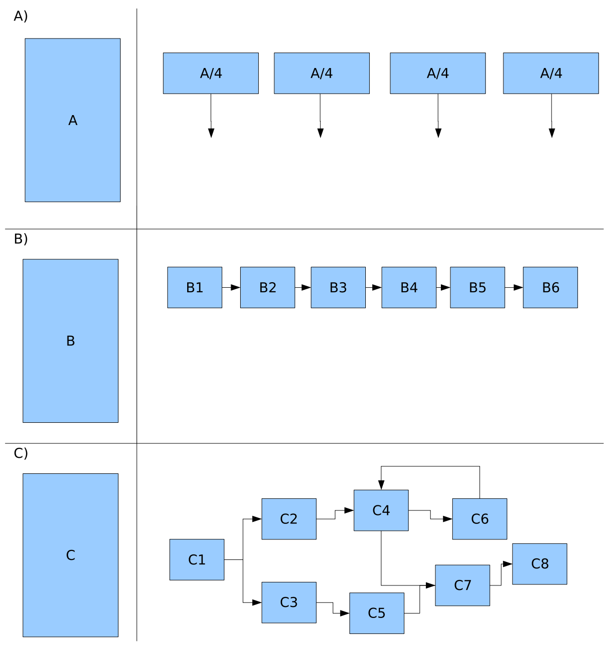


Figure 1.1: In case A) the application is divided in 4 tasks of the same type that perform the same computation to different but equal portions of data. In case B) the application is divided in stages of execution which are interconnected and form a pipeline. Case C) is a combination of A) and B) and forms a graph.

Even if spatial and temporal parallelism are not possible, RP platforms with fine granularity processing elements like FPGAs can be reconfigured to form application specific accelerators. In the case of applications that only use dedicated accelerators, parallelism can be realized only at the multitasking workload level were the available resources can be distributed among the running applications. Note, that fine-grained hardware processing elements like FPGA Lookup Tables (LUTs) cannot be dynamically shared between hardware accelerators and have to be statically assigned during design time. On the other hand, it is possible for these softcore accelerator cases to support the dynamic re-routing of these elements to accommodate more than one accelerators concurrently (in case they are sufficient LUT resources). For the time being, placement and routing on the FPGAs is static and arbitrary accelerators may not be concurrently accommodated unless they have been designed to coexist and the hardware has been appropriately configured to include them. Changes are not possible at runtime.

Since they are very versatile, RP platforms are suitable for a wide range of applications. Typically I/O bound applications may not be optimally served, but even in these cases a part of the computation may still be appropriate for acceleration. Therefore the best candidates are computations that perform CPU-intensive data transformations and operate on data block streams. Indicative examples are block cipher algorithms for encryption or authentication, data (de)compression algorithms that are widely used in data storage, and encoding algorithms for video or audio. Note that combinations of these application types comprise typical multitasking workload of an everyday general purpose system.

1.2 Programming Framework Considerations for Multitasking on RP Platforms

Achieving parallelism is more easily said than done. Several programming frameworks [2] have been proposed to abstract the design complexity of an application partitioning into independent entities that can be executed in parallel. Partitioning decisions are based on the computation dependencies and require extensive experience in order to be manually carried out by a programmer. Most frameworks observe the load distribution of a sequential execution of the computation and automatically decide partitioning and static load balancing.

There are two basic approaches to application partitioning: Coarse-grained and fine-grained. The former is usually applied to an algorithmic-level where a computation is divided in small tasks that can execute independently and is realized in most cases with code source-level restructuring. This approach is appropriate for PR platforms with powerful processing elements (CPUs or GPUs) also known as Massively Parallel Processor Arrays(MPPAs). Because of source level changes, the code can be further augmented during the restructuring process to cooperate with the platform runtime and enable execution coordination during multitasking. On the other hand, the fine-grained approach takes place after code analysis and during compilation, where small, usually loop fragments are scheduled to execute independently. In these cases the target platform features a large number of appropriately customized simple execution units, which are not typical CPU cores, that can be optimally exploited for fine-grained partitioning. An example is the Garp Array [3] and the respective toolchain [4]. Obviously the fine-grained approach significantly reduces possible dependencies between independent execution entities, simplifies the partitioning decisions and has the potential to impressively boost performance. A drawback is that the computation to communication ratio decreases with the processing load of each independent entity and, therefore, the partitioning granularity benefit is bounded. In some examples this limit can be very low, e.g, in decompression computations.

One of the major concerns that affects partitioning decisions is static load balancing. This is because parts with larger loads become the speedup bottleneck. In addition, uneven load distribution results in suboptimal resource utilization especially in cases of temporal parallelism where the next stage has to wait for the previous one to finish. Obviously coarse-grained partitioning performance is more vulnerable to radically diverse subcomputation loads which in the case of pipelines may result to extremely poor performance.

Automatic partitioning and static load balancing are the most important of the runtime performance challenges that the proposed frameworks for PR platforms deal with. The main focus is typically on the programming concepts and the abstractions that support rapid development, require less programming effort and enable source reuse. Other than that, the development of applications is based on the assumption that *all* the required target resources will be *dedicated* to the program throughout its execution. Moreover, for the time being, the operating system support on PR platforms usually runs on a master CPU and is used to deploy highly optimized applications which are expected to have been developed to occupy a *fixed* amount of resources. If another resource demanding application arrives at the system, and it cannot be given the required amount of resources, it will have to wait for the previous one to finish.

As a different approach, in case of resource shortage, one could consider the *dynamic distribution* of available resources between concurrently executing applications. In other words, if a new application arrives, currently deployed applications could be reconfigured to free some resources that can be exploited by it. Similarly, when an application finishes, the released resources can be reassigned to speedup the rest of the workload.

Since general purpose computing and radical multitasking environment may only be realized on instruction-set architecture (ISA) processors, dynamic resource redistribution cannot be realized on tiled architecture platforms [5] that primarily feature highly customized, application specific processing elements. The latter platforms may only support a primitive form of multitasking by allowing the immediate loading of applications that utilize resource combinations which can be hosted concurrently, regardless of their arrival at the system. This approach requires only operating system support and it will be discussed in the next section.

Moreover, General Purpose GPU (GPGPU) platforms like Nvidia CUDA [6] organize the GPUs into groups that are connected to local group-level shared memory and may also access global memory if needed. Unfortunately, hardware-level support is used to control in a simple round-robin manner the concurrent execution of programmer defined lightweight threads that belong to the same computation. While we believe that next generations of multi-GPGPU platforms will be more versatile in this respect, we do not consider them in the implementation prototypes since the coordination of execution is not software controlled.

On the other hand, parallel ISA processor arrays that feature dedicated interconnections can be efficiently used for multitasking. Commercial examples are Ambric processor which features 360 32-bit processors [7], picoChip with 300 32-bit processors [8] and Intellasys SEAforth with 40 32-bit cores [9]. Academic examples are the 36-core Asap processor [10] which is very similar to Ambric and PARO [11] which is used to build processor arrays on reconfigurable hardware. All of these architectures come with respective programming tools.

1.2.1 Task Migration and Load Balancing Support

While we believe that each PR platform processing core should be used in dedicated mode, we also believe that it is a good idea for programming frameworks to produce application executable entities which are flexible enough to operate with a varying number of assigned cores that is determined at load time and can be changed during execution. Each application to be deployed on the PR should request resources from the underlying OS and execute on the provided number of cores. In order for this approach to be feasible, application executables should be produced in a way that allows more than one (predefined) tasks to execute on the same core, in a transparent fashion. Below, we discuss two different approaches that could be used to achieve this.

1.2.2 Distributed OS Client Approach

One solution is for each MPP core to feature a thin runtime layer which provides tasks with all the necessary abstractions that decouple their code from specific bus addresses and core ids, and to employ a (simple) mechanism that takes care of task migration and execution on a single core. The local runtime instances would be controlled by a full-fledged OS support that runs on a Master CPU. In this case, the programming framework would only have to produce application task executables that would be properly linked against this thin runtime layer. This approach is depicted in figure 1.2A.

Several performance issues have to be considered in this case. Firstly, task migration involves transfer of actual executable code and a coordinated suspend/resume scheme, which is not easy to achieve in case of runtime rearrangements and can cause serious performance degradation given that RP array core runtime instances must communicate with a central service on the Master CPU over a multi-hop interconnection infrastructure. Put in other words, each time an application adapts its execution, in practice, it needs to be reloaded. Moreover, to balance the application load between cores, each node should be capable of determining the load of each task running on it in order to report it to the central service, introducing additional overhead and complexity. Last but not least, code migration implies that all PR cores are of the same architecture. While current PR platforms usually feature the same CPU architectures, processor arrays on reconfigurable hardware could be used to employ different architectures, customized to certain types of computations. Since hardware reconfiguration has the potential to introduce heterogeneous PR platforms, it would be nice for the aforementioned support to work on heterogeneous PR systems as well. As a special case of heterogeneity, the Master CPU itself could be used to execute some of the application's tasks as well.

1.2.3 Integrated Task Execution Control

This approach requires each core to be fitted with a very small Basic Input Output System (BIOS) that mainly helps with initial application loading. Contrary to the other frameworks, code migration is *avoided* by building, for each core architecture and local runtime environment, a *single* image

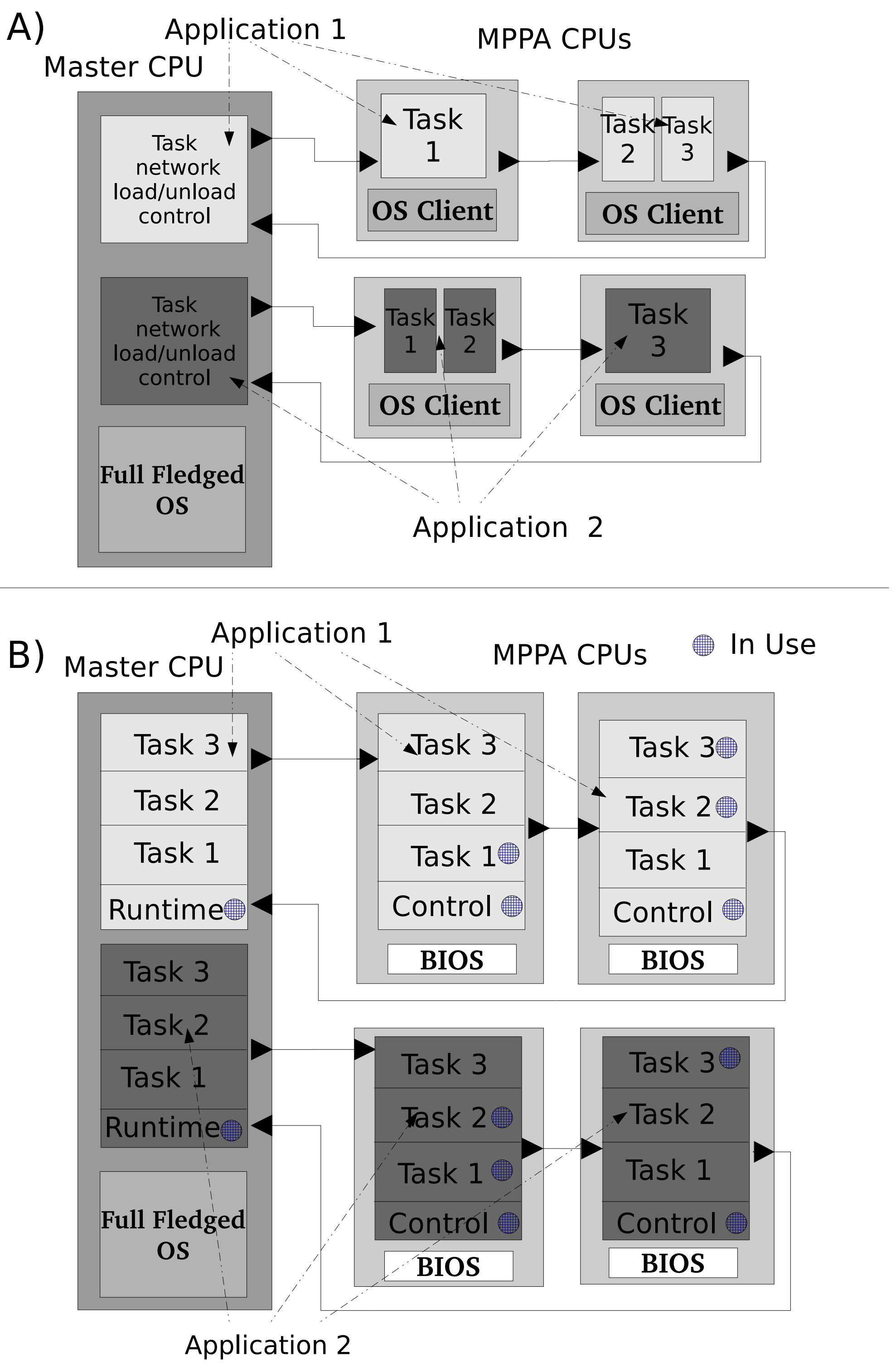


Figure 1.2: Approaches for the application deployment on RP CPU Arrays to support dynamic task reassignment.

that contains *all* the tasks of the application. Cores that are of the same architecture and execute the same runtime are loaded with a copy of the same executable. In this case, simple execution control is statically included in the application image to coordinate local task execution (the BIOS is ignorant of this functionality). Global coordination decisions are taken on the Master CPU, querying the operating system for resource availability. The Master CPU can also be exploited to perform computation iterations (sequentially) in an efficient profiling mode to gather data that can help to make better partitioning decisions. Note that the OS merely provides information about the available resources but partitioning decisions are taken by application-level logic (that is automatically generated by the toolchain). An indicative setup of this kind is depicted in Figure 1.2B

Application data flows between cores primarily transfer data blocks for processing, but they may also carry control commands downstream. Distinguishing between the two data types is supported in hardware by all the dedicated interconnection architectures. Control commands can be used to set appropriate control information indicating which tasks should be executed locally by each core. With this approach the execution flow can be *dynamically* redirected *inside* the PR CPU array *without* requiring the Master CPU to communicate with each core individually. Also, if new resources are to be occupied that do not feature the application executable, the closest processor with the same architecture and BIOS combination can provide the new member with the copy needed. However, while the previous approach enables a transparent dynamic task migration and load balancing without the involvement of the programming framework, in this case this functionality is achieved through a combined interaction between development and runtime environment.

It must be noted that in some cases the application image size may exceed the local RP CPU memory capacity. In this case the toolchain can build an appropriate number of different image types each one containing a partial number of the application tasks. This introduces a lower bound for the number of cores that will be needed for application execution, but this is a hardware constraint.

A good example of a programming framework that produces modularized flows of execution that are controlled by an integrated runtime environment is the *Click* modular router [12]. This software is used to create packet processor engine configurations that can be used to quickly implement novel OSI layer 3 protocols, which may also execute as a part of the linux kernel network stack in real life setups. The programmer has to develop C++ objects using the provided underlying runtime extensions as well as a high-level language for defining the interconnections between them to form processing configuration files. Taking advantage of this architecture, *Click* supports the so called hotconfig feature. The runtime instantiates as many modules (called elements) as the programmer defines at development time which are not necessarily used in all defined configurations. A new configuration file can be loaded at runtime and with minor overheads the *Click* running instance can switch between configurations which is very desirable feature for this class of network applications. While *Click* has been developed for a totally different purpose, we found many proposed concepts appropriate for a PR CPU array programming framework that can support task migration and dynamic load balancing.

1.3 Prototype Frameworks for PR CPU Array Platforms

These considerations inspired us to design and implement an PR platform development framework core with two different programming models. The first model is based on the OpenMP paradigm and introduces source annotations that can be regarded as OpenMP extensions for distributed memory PR CPU array targets. The basic idea behind this practical approach was the extraction of coarse-grained, mostly temporal, parallelism out of existing sequential applications. The obvious benefit is the extensive reuse of existing codebase, which is a major motivation for rapid system development. We were particularly interested to investigate, for PR CPU arrays, the potential of an incremental approach that still uses a form of the traditional sequential programming and acts as a first step of realizing pipelined parallel programming into the computing mainstream. This is the heart of the OpenMP concept as well. In Chapter 3 we present this approach and respective framework design to achieve multitasking by using the concepts that we discussed in 1.2.3.

Taking advantage of the experience of our first programming model and after revisiting initial concepts for multitasking throughout the first framework design, we developed *PipeIt*, a prototype programming framework with a different programming model that additionally supports task migration to achieve load balancing and enables multitasking on PR CPU platforms. Following the approach described in Section 1.2.3, *PipeIt* features a simple form of runtime execution control, integrated into the application executable, making it possible to seamlessly execute the programmers defined tasks (called components) on *any* number of cores. Notably, *PipeIt* explicitly focuses on supporting *pipelined* computations.

We have developed working prototype implementations of both programming models toolchains with respective runtime support for a custom PR CPU array platform of *Microblaze* soft processors deployed on a Xilinx FPGA target. The Master CPU is also a Microblaze that runs *uClinux* and is interfaced to all other platform peripherals like network and storage.

1.4 Runtime Hardware Reconfiguration

Reconfigurable hardware is no longer used only for prototyping purposes. End-user applications are nowadays deployed on reconfigurable platforms and take advantage of the reconfiguration ability. Since this type of applications require specific, per-application platform changes to execute, the basic perquisite for multitasking is the ability to optimally (re)distribute resources on demand and at runtime.

This could have been an easy task for the Operating System if the hardware runtime reconfiguration was entirely handled in hardware and therefore seamless to software. While there are significant research efforts to support this with the so-called Dynamic Partial hardware Reconfiguration (DPR) in FPGA's and other reconfigurable systems like the aforementioned parallel processor arrays with reconfigurable interconnections, the few runtime reconfiguration scenarios that are currently supported by this approach are not adequate for typical multitasking. More specifically, in DPR, the reconfigurable resources are statically grouped in hardwired areas that can be independently reconfigured and the development toolchain has to be aware of partial reconfiguration scenarios during design time in order to appropriately place logic into these predefined areas. From a multitasking perspective this is not optimal.

In our approach we abstract runtime hardware reconfiguration entirely in software and we only rely on a basic full scale reconfiguration mechanism. We achieve this transparency at the operating system level with proper kernel support and we distribute available RP resources with a corresponding system service that performs the respective housekeeping and controls the reconfiguration process. We have implemented a prototype that deals with the most radical reconfiguration scenario: Entirely soft System-on-Chip (SoC) platform with several soft-cpu(s) on FPGA chip, running a full-fledged Linux.

1.5 Outline of this Dissertation

The rest of this dissertation is organized as follows.

In Chapter 2 we present in detail the proposed system-level support design that enables runtime reconfiguration of the underlying Parallel Reconfigurable platform. A prototype uClinux implementation (with both kernel- and application-level support) that runs on an FPGA soft platform is also described along with experimental results for the case of a well known application. In addition, hardware extensions that improve Hw/Sw synergy are also proposed.

In Chapter 3 we present a programming model and a framework that enables the code source-level restructuring of existing C language codebase by extending the annotations concept of OpenMP [13]. With this approach, coarse-grained parallelism is extracted out of existing sequential applications, and along with a simple dynamic load balancing scheme, regular well known applications are accelerated on Parallel Reconfigurable Processor Array (PRPA) targets and in a multitasking context.

In Chapter 4 we present *PipeIt*, a programming model for PRPAs that is based on the evolving concept of wired components[14][12][2]. While *PipeIt* has a reasonably steeper learning curve than our first OpenMP-based approach, we consider it a cleaner design, that better supports dynamic load balancing and can optimally suit any PRPA resource organization. Moreover, *PipeIt* has been augmented to inherently support multitasking by building applications that are flexible with the required amount of resources which can be (re)configured at will and at runtime.

Chapter 2

Application and System-level Support for Runtime Hardware Reconfiguration

This chapter discusses the design and implementation of a system-level mechanism and corresponding application-level support that enables programs running on a PR SoC to modify the underlying platform at runtime. Applications may request the addition and/or removal of processing elements, that are referred to as "devices", or the modification of their interconnections at any point in time. In the presented prototype implementation we used an FPGA-based platform and requests are handled in a coordinated way via a separate user-level process that fetches the appropriate FPGA 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) FPGA hardware that supports dynamic partial reconfiguration, the obtained functionality is sufficient for a wide range of application scenarios.

2.1 Introduction

The technology of RP Systems has the potential to change the way computing systems are being built. While RPs, especially FPGA-based, are not as fast or energy saving as corresponding ASICs [15] 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, accelerators and customized CPUs to offload the main 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 main 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 [16] 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 [17]. 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 chapter 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 work 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.

2.2 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, corresponding partial bitstream generation tools support or any other hardware-level runtime reconfiguration technology. This way we can take advantage of simpler RP platforms that support only the basic full reconfiguration as well and end up with a very portable design which is not even specific to the FPGA technology. The next subsections give an overview of our approach, motivating the various decisions taken.

2.2.1 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

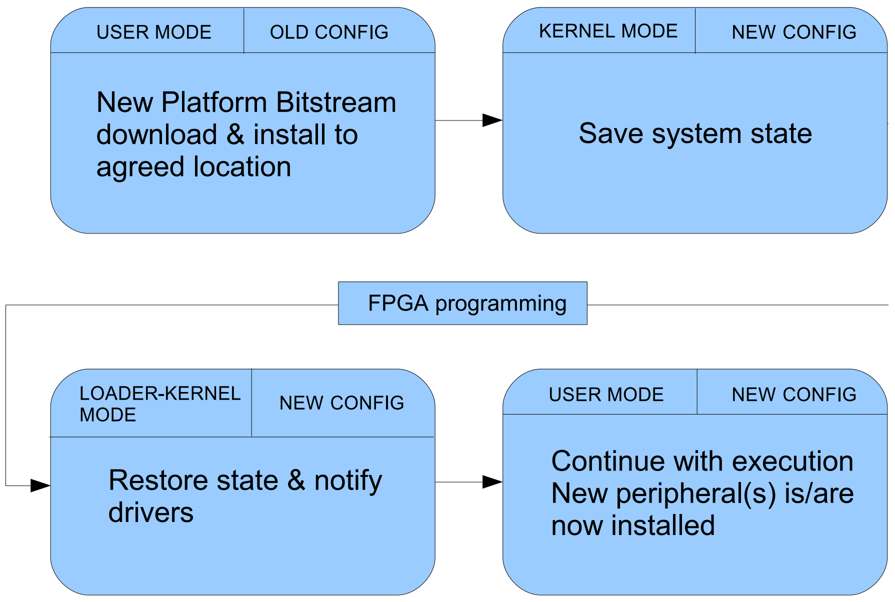

USER MODE
OLD CONFIG
New Platform Bitstream download & install to agreed location
KERNEL MODE
NEW CONFIG
Save system state
FPGA programming
LOADER-KERNEL MODE
NEW CONFIG
Restore state & notify drivers
USER MODE
NEW CONFIG
Continue with execution New peripheral(s) is/are now installed

Figure 2.1: The main phases of the reconfiguration scheme

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 2.1.

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 [18]). 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.

2.2.2 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.

2.2.3 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.

2.3 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.

2.3.1 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 [20] (Figure 2.2) 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

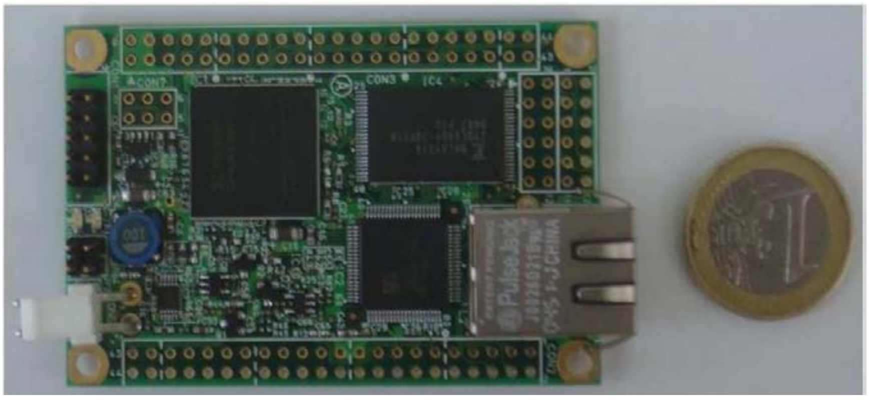


Figure 2.2: Atmark Techno Suzaku

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.

2.3.2 The Peripheral Device Location Table

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

2.3.3 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 [21] 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.

2.3.4 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.

2.3.5 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.

2.4 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 currently 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.

2.4.1 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.

2.4.2 Example

This functionality is illustrated in Figure 2.3 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

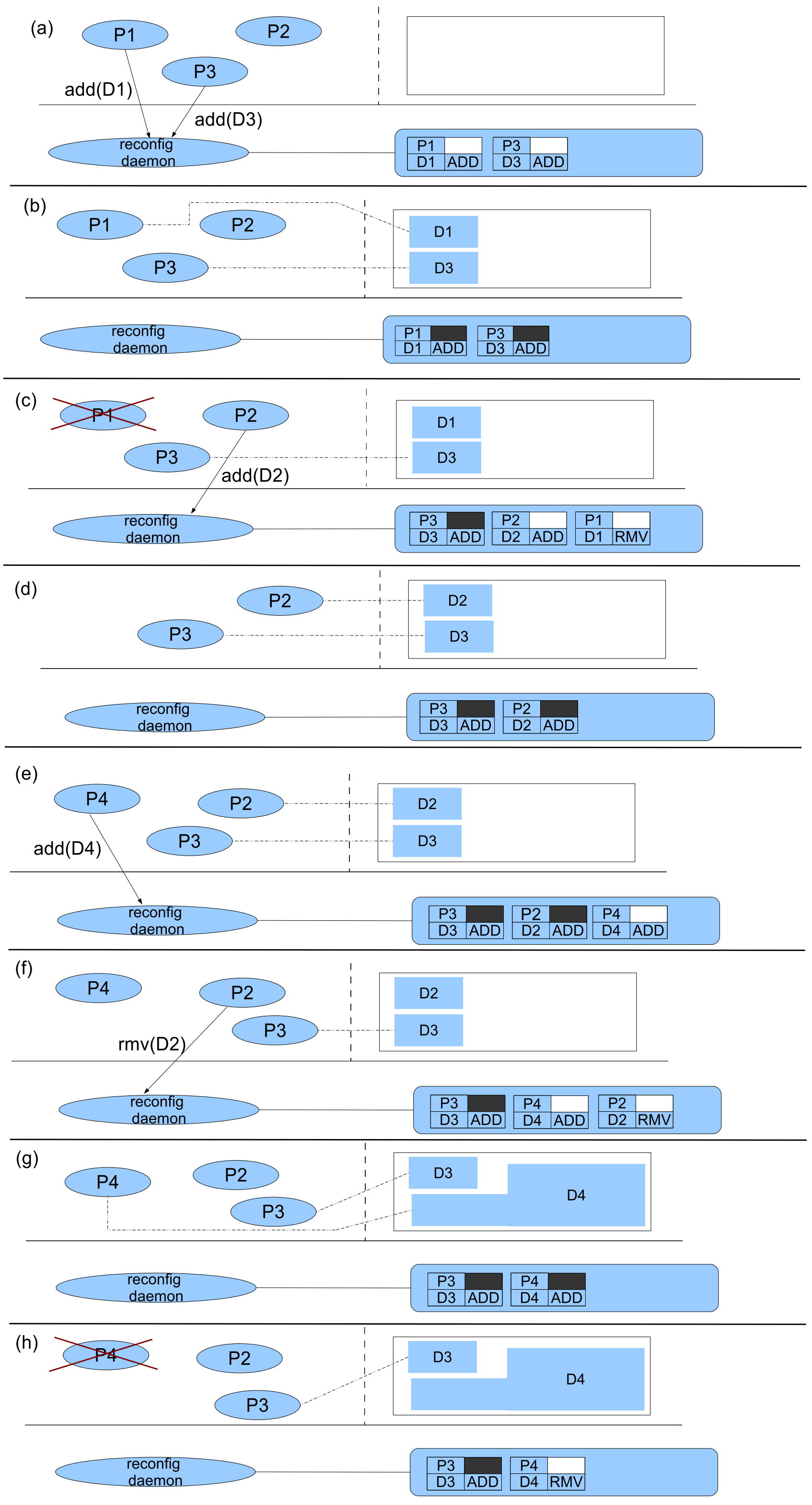


Figure 2.3: A reconfiguration scenario

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

2.4.3 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.

2.4.4 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 [22].

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.

2.5 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.

2.6 Evaluation of the Reconfiguration Coordination

The presented reconfiguration framework allows applications to add and exploit custom H/w accelerators at runtime, without caring about various coordination and optimization issues. The ideal case would be for the system to have abundant hardware resources so that all requested accelerators could be promptly installed. However, most real systems are likely to accomodate only a few accelerator devices, making it impossible to satisfy all device addition requests if several application tasks execute concurrently. In this section we discuss indicative experiments that have been conducted to investigate the performance of the system under load stress.

2.6.1 Experimental Setup

For experimenting with our prototype system, we developed a codesigned application that uses DES-56 cipher to repeatedly encode and decode a 64-bit block of local memory data with a key. The application uses a software version for DES computation or a special H/w accelerator, if available. When the applications starts, it issues an addition request for the accelerator and then begins the computation in software mode. It switches to hardware mode as soon as the accelerator is installed. When the computation finishes, the application issues a removal request for the accelerator and terminates.

The DES accelerator is integrated as a memory mapped peripheral of the Microblaze platform and can be accessed via the On-chip Peripheral Bus (OPB). It has two 32-bit registers for input data, two for the key, two for the output data, and one register which acts as a control register. Once the input is loaded, the results are produced after eight bus cycles. This is fast enough to avoid the use of interrupt logic for result delivery.

Performing the computation in software, requires 75 seconds at 1.080 watts. Only 2 seconds at 1.082 watts are needed when the H/w accelerator is used. This corresponds to a 37.5 speedup. In our prototype, each reconfiguration requires about 22 seconds at an average of 1.091 watts (11 times more than doing the computation in H/w). Most of the time is spent to download the bitstream over the network (3 seconds) and to write it to the external flash in polling read mode (18 seconds). The actual FPGA reprogramming and the entire suspend-resume cycle needs less than 1 second to complete. Application execution is suspended during reconfiguration.

In our experiments, the system workload is formed out of tasks that are identical instances of the above application. To reproduce the effects due to contention for hardware resources, we artificially differentiate between different tasks classes, each requiring a different accelerator; in reality, the application code and accelerators are identical, but this information is not made available to the reconfiguration daemon. A simple workload generator program is used to submit new tasks to the system. The generator stops after having created a certain number of tasks.

2.6.2 System Simulator

Although our prototype system can be used to test different application workloads, the experiments that can be performed using it are limited. This is mainly due to the small size of the reconfigurable hardware, the time it takes for applications to complete their execution, and the considerable overhead of reconfiguration.

In order to investigate different scenarios and system parameters in a straightforward way, we have also built a discrete event simulator of the reconfiguration framework. The simulator can be parameterized in terms of the size of the reconfigurable hardware (in abstract H/w units) and the reconfiguration policy used by the daemon. The system workload is specified as a mix of tasks belonging to application classes with different S/w and H/w execution times (in abstract time units) and accelerator sizes (in abstract H/w units). To keep it simple, the simulator does not account for the delay due to the CPU time spent by the reconfiguration daemon, which is not negligible on our platform. As a consequence, the simulated results are expected to be better than the results obtained via the system prototype.

2.6.3 Runtime reconfiguration vs static configuration

In a first set of experiments we investigate the importance of the ability to dynamically install hardware accelerators, according to the application workload. As a reference we use a static system configuration where accelerators are pre-installed and cannot be changed at runtime.

We employ 4 different application classes with the performance characteristics (software and hardware execution time, and accelerator size) of the DES application discussed above. The workload comprises a total of 12 application tasks, 3 for each application class. All tasks are submitted to the system at once. The reconfiguration daemon satisfies the corresponding addition requests as soon as possible. It also satisfies several requests via a single reconfiguration, provided that there are sufficient H/w resources to accomodate the corresponding accelerators.

Four different cases are considered in terms of H/w capacity: (i) no hardware accelerators are supported, thus all tasks execute entirely in software; (ii) the hardware can accomodate one accelerator; (iii) the hardware can accomodate two accelerators; (iv) the hardware can accomodate three accelerators.

Figure 2.4 shows the average application task completion times on our prototype and simulator, for a reconfigurable system versus a static system that comes with pre-installed accelerators. As it can be seen, the benefit of runtime reconfiguration increases as H/w resources decrease. Especially

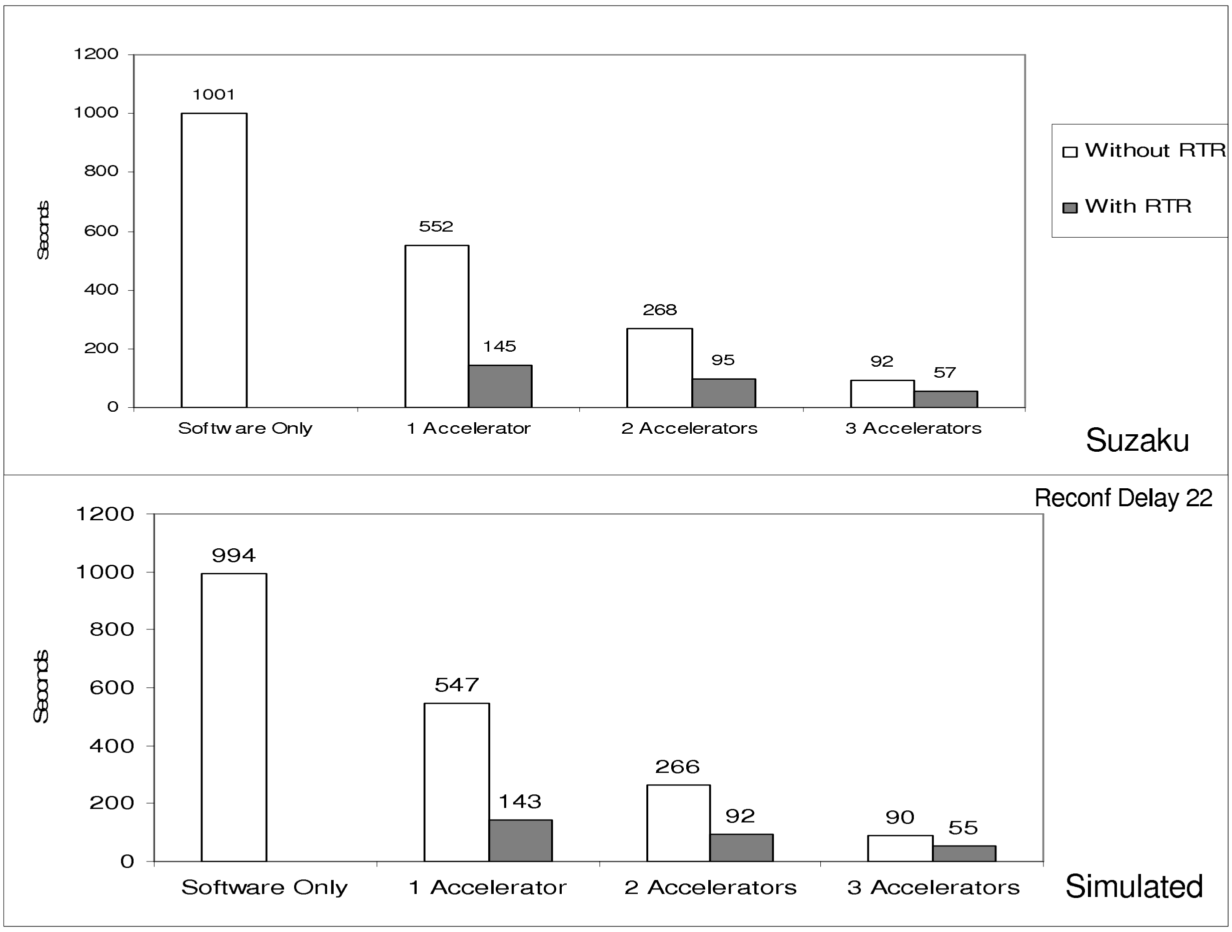


Figure 2.4: Real and simulated average task completion time

noteworthy is the difference for the case where the hardware can accomodate only 1 accelerator. The comparably moderate performance for the case where 3 accelerators fit on the hardware is due to the fact that 9/12 tasks will get a chance of executing in hardware even in the static configuration. Nevertheless, the reconfigurable system performs significantly better in absolute terms (almost twice as fast) due to its ability to provide H/w resources to the rest of the 3 tasks and the fact that the reconfiguration overhead (though very large compared to the H/w execution time) is small relatively to the time for doing the computation in software.

These figures are expected to get even better as the cost of reconfiguration drops. This is confirmed by additional simulated experiments for a system with 1/2 and 1/4 the reconfiguration overhead of our prototype. The results are shown in Figure 2.5. Even though, as noted, the simulator results give an optimistic lower bound, they are a solid indication of the performance to be expected in a real implementation. This can be confirmed by comparing relative difference between the real

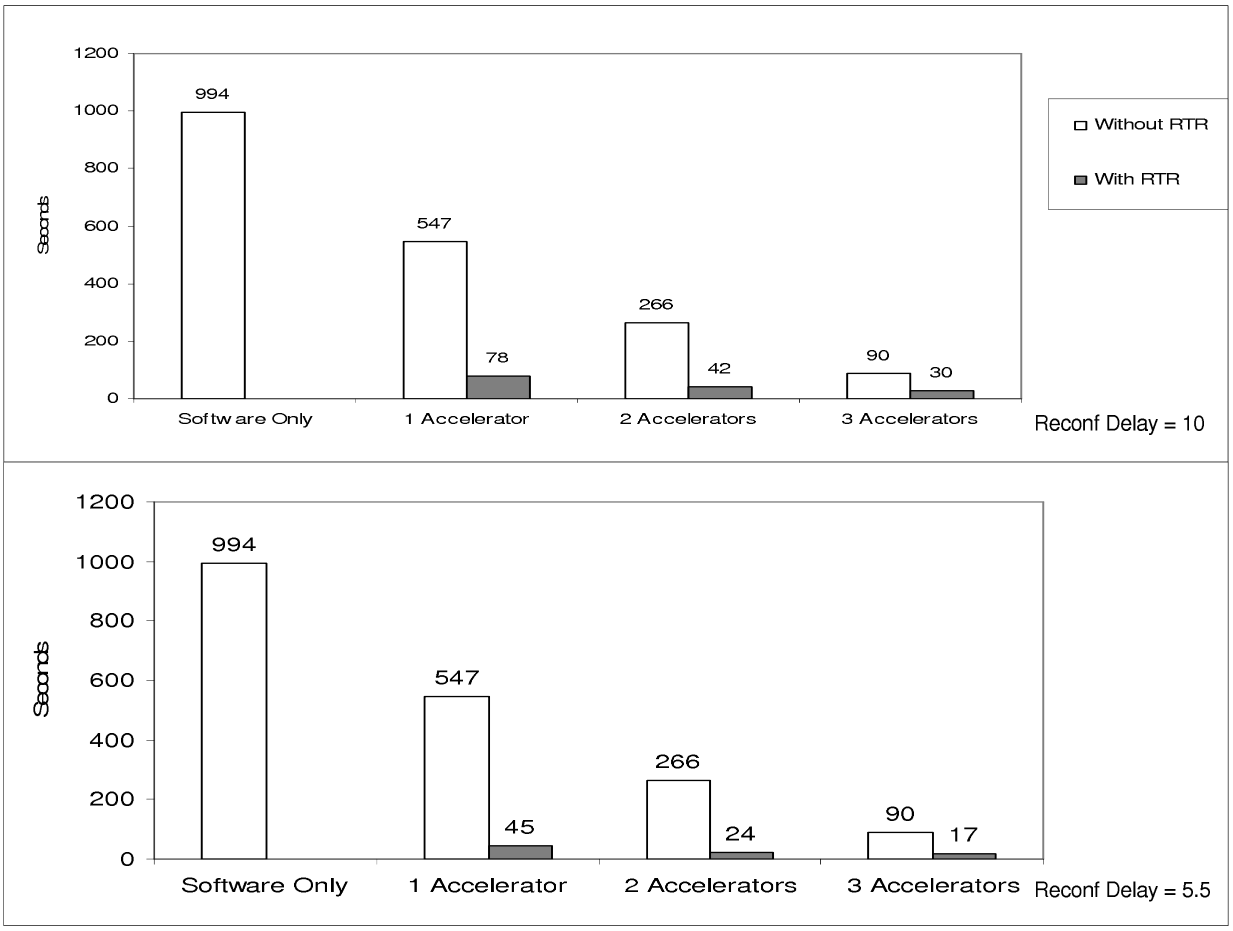


Figure 2.5: Simulated average task completion time

vs simulated results in Figure 2.4.

2.6.4 Impact of the reconfiguration policy

In a second set of experiments, we investigate the impact of the policy adopted by the daemon to defer reconfiguration. This is done in the hope that new addition requests will arrive which can be satisfied, together with the ones that are already pending, via a single reconfiguration.

We employ 5 different application classes with different performance characteristics, listed in table 2.1. The system workload comprises a total of 1000 application tasks, 250 for each application class. Tasks are submitted to the system following a Poisson distribution. The available hardware resources are 110 units, making it impossible to host the accelerators of all applications classes at the same time. Note that the average software execution time of application tasks is 1500 time

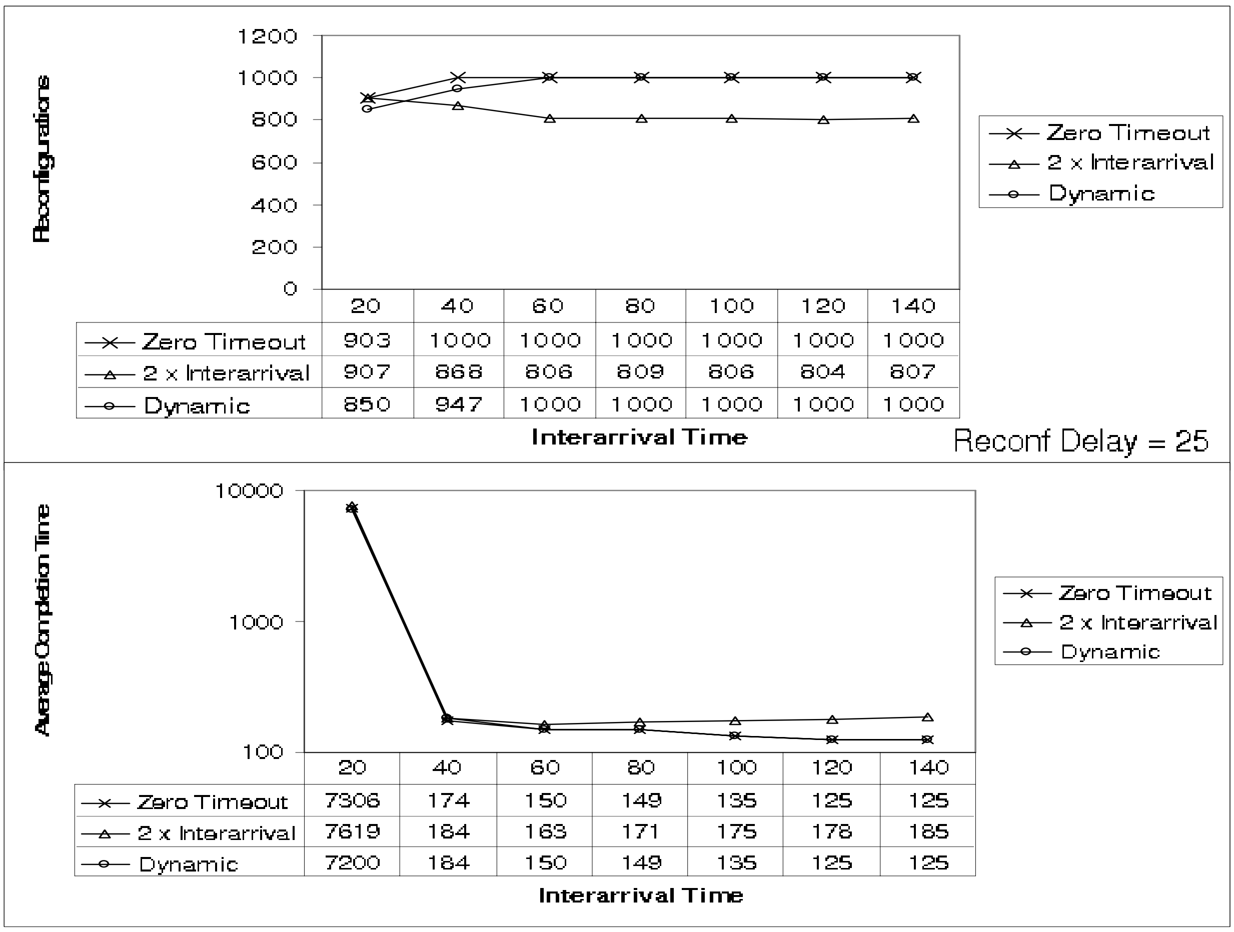
App Class	1	2	3	4	5
S/W Time	500	1000	1500	2000	2500
H/W Time	100	100	100	100	100
Speedup	5	10	15	20	25
H/W Size	25	35	40	40	25

Table 2.1: Characteristics of application classes

units, whereas the hardware execution time is 100 time units for all tasks.

Three policies are considered. The first policy is to reconfigure as soon as possible. This corresponds to the lack of any policy and is used as a baseline. The second policy is to defer reconfiguration for a constant amount of time that is twice the mean task interarrival time. This corresponds to our initial implementation, where the idea is to give a fair chance for new addition requests to arrive before performing a reconfiguration. The third policy is as follows. If the remaining hardware resources are not enough to accomodate an accelerator of average size (based on the size of accelerators that have been installed so far), then reconfigure (taking into account all pending requests). Else, if the reconfiguration time is *relatively small* compared to the average interarrival time, then reconfigure. Else, if the time during which a request has remained pending is *relatively large* compared to the hardware execution time of an application task yet at the same time *relatively small* compared to its software execution time divided by the number of concurrently executing tasks, then reconfigure. Else, wait some more time hoping that a new addition request will arrive which can be satisfied together with all pending requests, as long as the total waiting time is *relatively small* compared to the interrarival time. The factors used to compare times have been chosen based on numerous experiments which are not shown here for brevity.

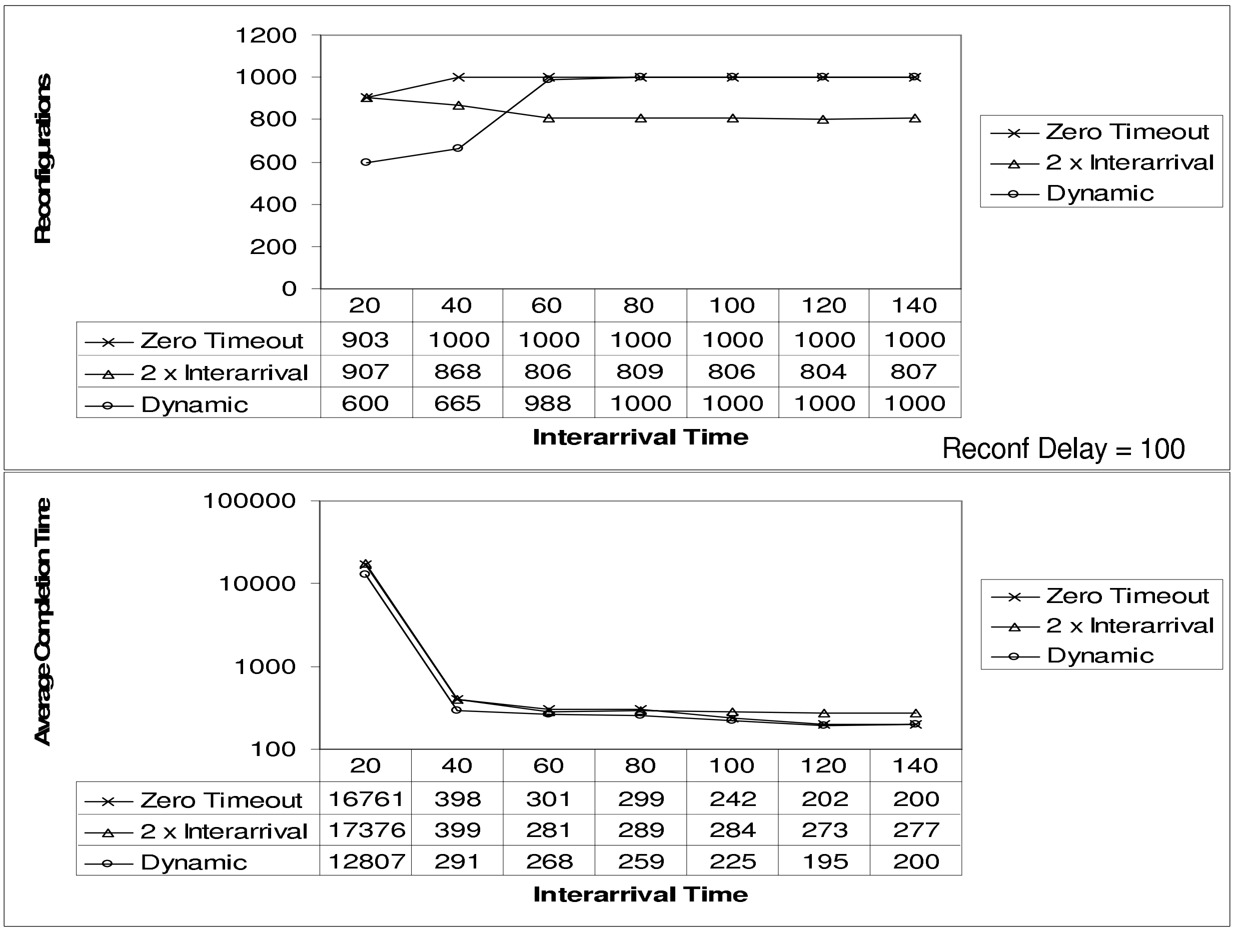
Figures 2.6, 2.7, 2.8 and 2.9 show the number of reconfigurations performed and the average task completion time (the latter in logarithmic scale) as a function of the interarrival time, for a system reconfiguration cost of 25, 100, 200 and 400 time units, respectively. Based on these results, it is quite obvious that the reconfiguration policy used becomes relevant only in the cases where the reconfiguration time is (a) significantly larger compared to the H/w task execution time, but also (b) larger than half of the average task interarrival time. Else, both the fixed time and dynamic



	20	40	60	80	100	120	140
Zero Timeout	903	1000	1000	1000	1000	1000	1000
2 x Interarrival	907	868	806	809	806	804	807
Dynamic	850	947	1000	1000	1000	1000	1000

	20	40	60	80	100	120	140
Zero Timeout	7306	174	150	149	135	125	125
2 x Interarrival	7619	184	163	171	175	178	185
Dynamic	7200	184	150	149	135	125	125

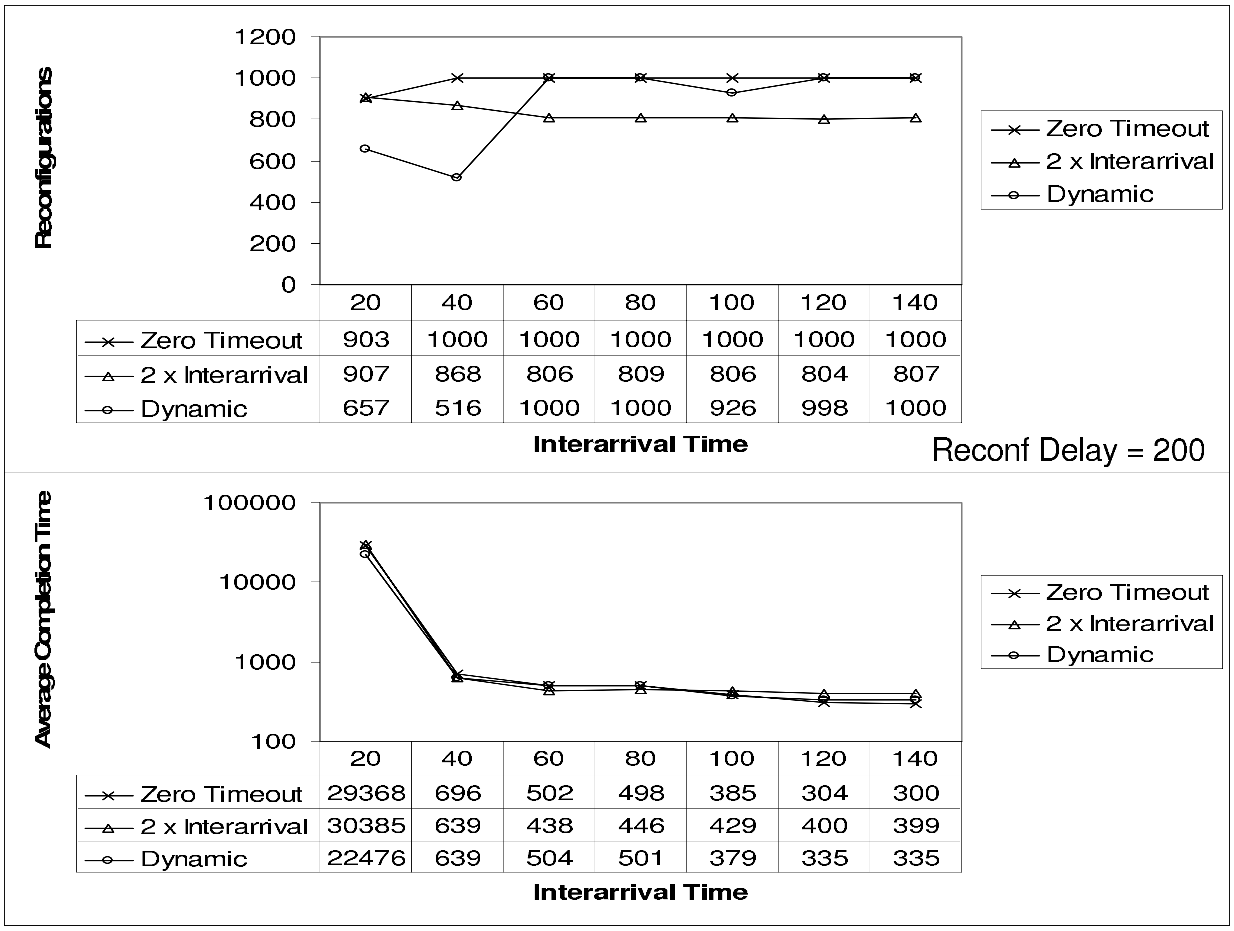
Figure 2.6: Timeout policies with reconfiguration cost 25



Interarrival Time	20	40	60	80	100	120	140
—×— Zero Timeout	903	1000	1000	1000	1000	1000	1000
—△— 2 x Interarrival	907	868	806	809	806	804	807
—○— Dynamic	600	665	988	1000	1000	1000	1000

Interarrival Time	20	40	60	80	100	120	140
—×— Zero Timeout	16761	398	301	299	242	202	200
—△— 2 x Interarrival	17376	399	281	289	284	273	277
—○— Dynamic	12807	291	268	259	225	195	200

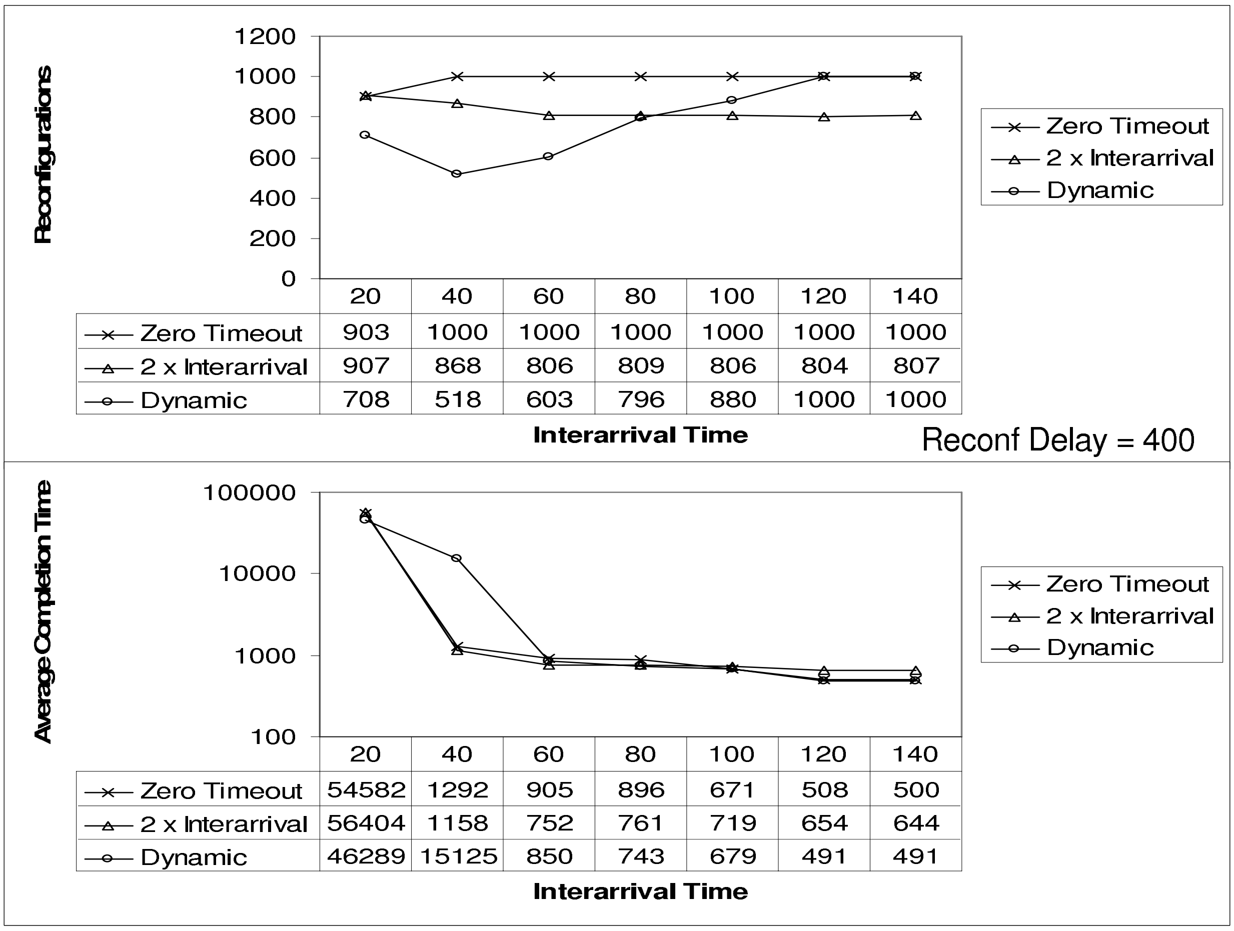
Figure 2.7: Timeout policies with reconfiguration cost 100



	20	40	60	80	100	120	140
Zero Timeout	903	1000	1000	1000	1000	1000	1000
2 x Interarrival	907	868	806	809	806	804	807
Dynamic	657	516	1000	1000	926	998	1000

	20	40	60	80	100	120	140
Zero Timeout	29368	696	502	498	385	304	300
2 x Interarrival	30385	639	438	446	429	400	399
Dynamic	22476	639	504	501	379	335	335

Figure 2.8: Timeout policies with reconfiguration cost 200



Interarrival Time	20	40	60	80	100	120	140
Zero Timeout	903	1000	1000	1000	1000	1000	1000
2 x Interarrival	907	868	806	809	806	804	807
Dynamic	708	518	603	796	880	1000	1000

Interarrival Time	20	40	60	80	100	120	140
Zero Timeout	54582	1292	905	896	671	508	500
2 x Interarrival	56404	1158	752	761	719	654	644
Dynamic	46289	15125	850	743	679	491	491

Figure 2.9: Timeout policies with reconfiguration cost 400

policy result in a performance that is close, sometimes marginally worse, than the one obtained when reconfiguring as soon as possible.

We note that saving as many reconfigurations as possible does not per se improve completion time. This is because some tasks are excessively delayed, waiting for a reconfiguration to occur, and spend most of their time to perform their computation in S/w (which is on average significantly slower than H/w). It is by putting the cost of reconfiguration in perspective of both the task interarrival time and hardware vs software execution time that achieves the best results. Indeed, the dynamic policy, designed according to this rationale, achieves the best overall results for a very wide range of task interarrival and reconfiguration time values; even though it does not always result in the fewer number of reconfigurations.

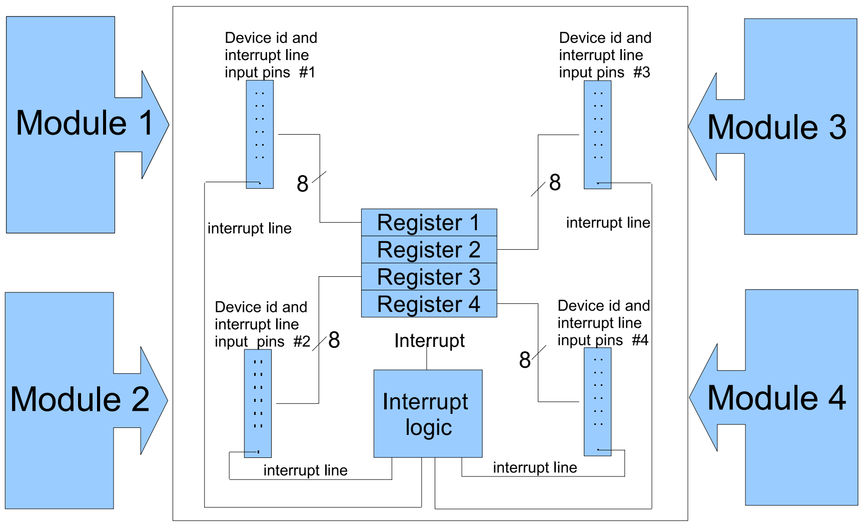
2.7 Support for off-chip peripherals

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.

2.7.1 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



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 2.10: The hotplug detector high-level schematic

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 2.10.

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[23] for a similar DPR-based implementation.

2.7.2 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.

2.7.3 Example

Figure 2.11 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

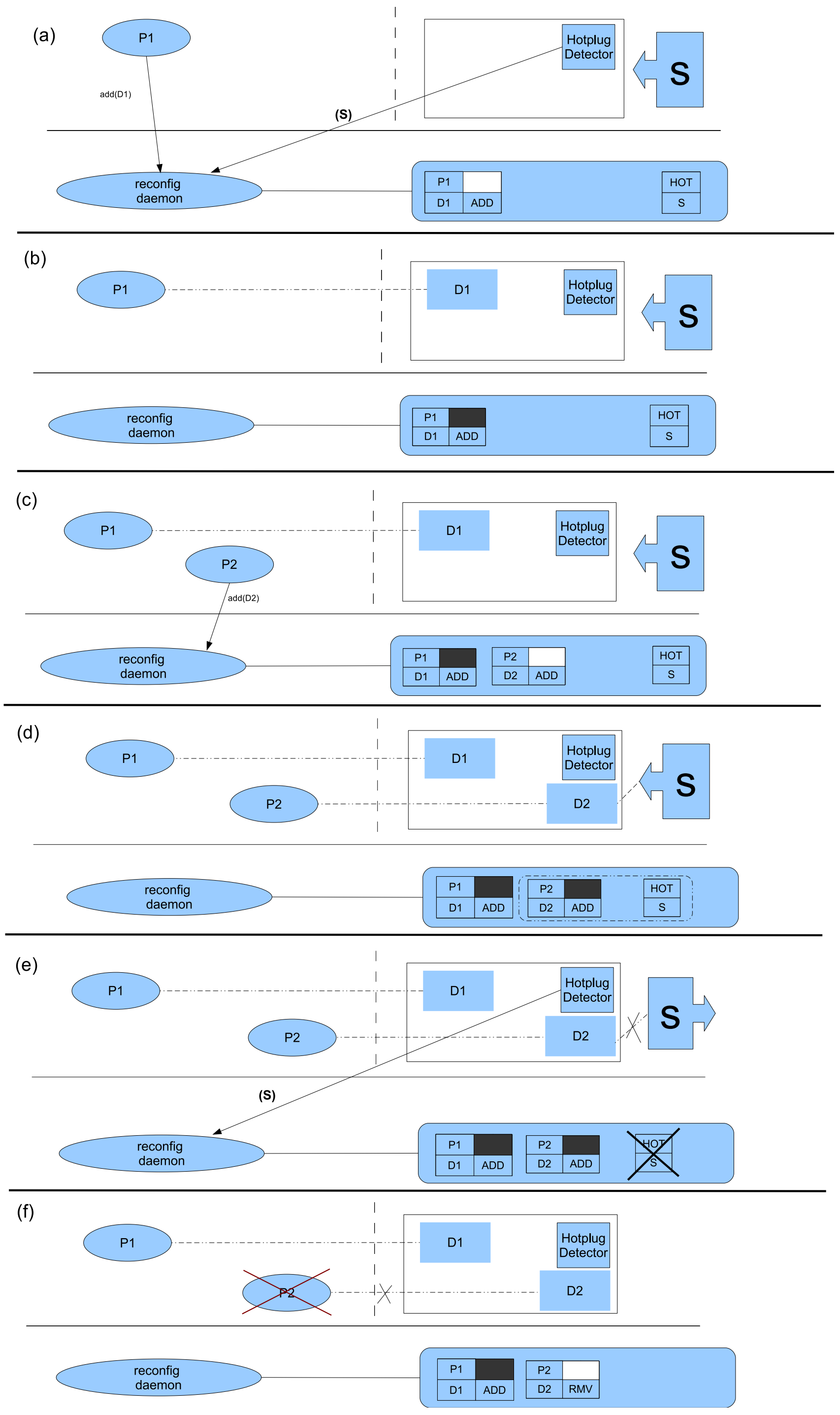


Figure 2.11: A reconfiguration scenario with hotplug event

already installed as a mandatory device.

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

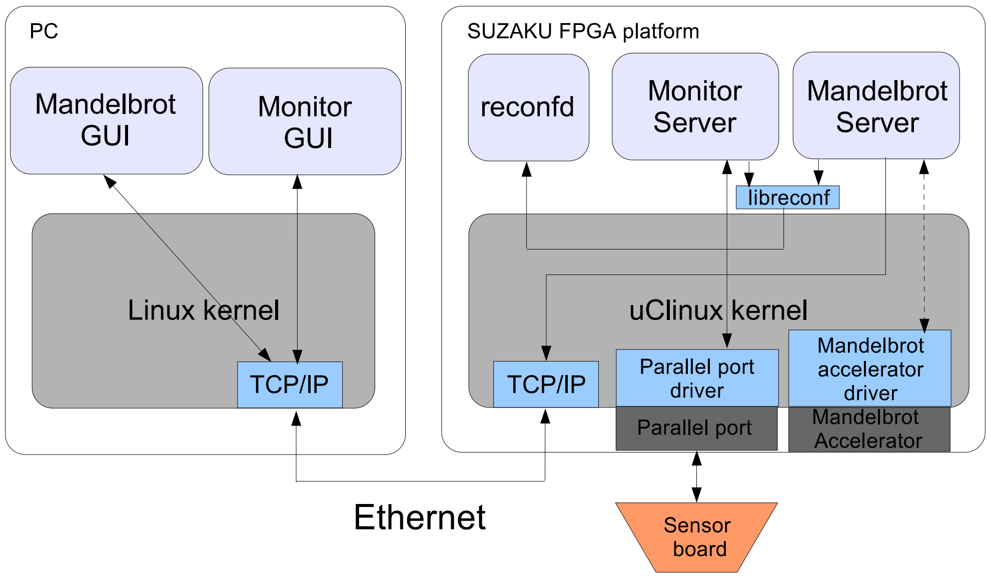


Figure 2.12: Demo Setup

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).

2.8 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 2.12. A picture of the Suzaku board setup is shown in Figure 2.13.

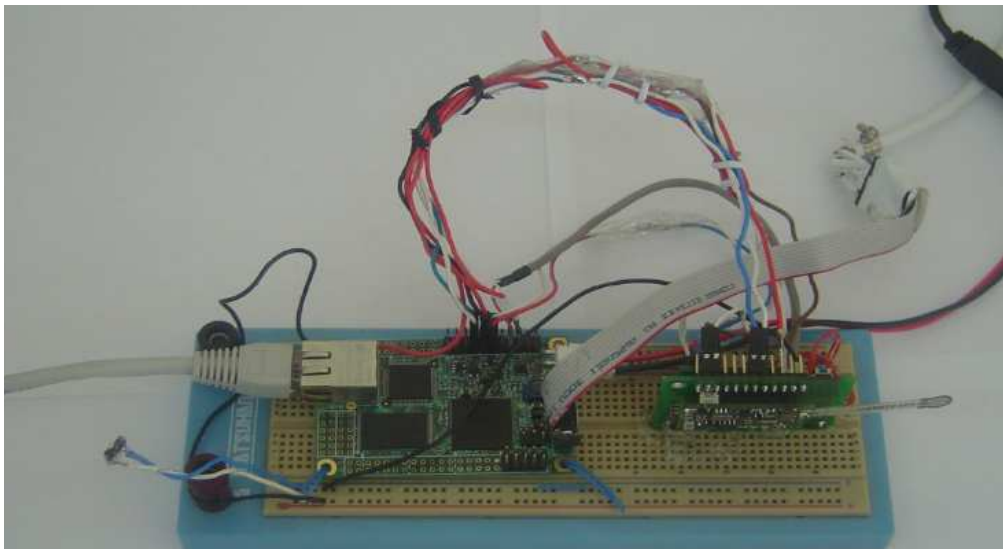


Figure 2.13: Suzaku board setup

2.8.1 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 2.14 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 soft-

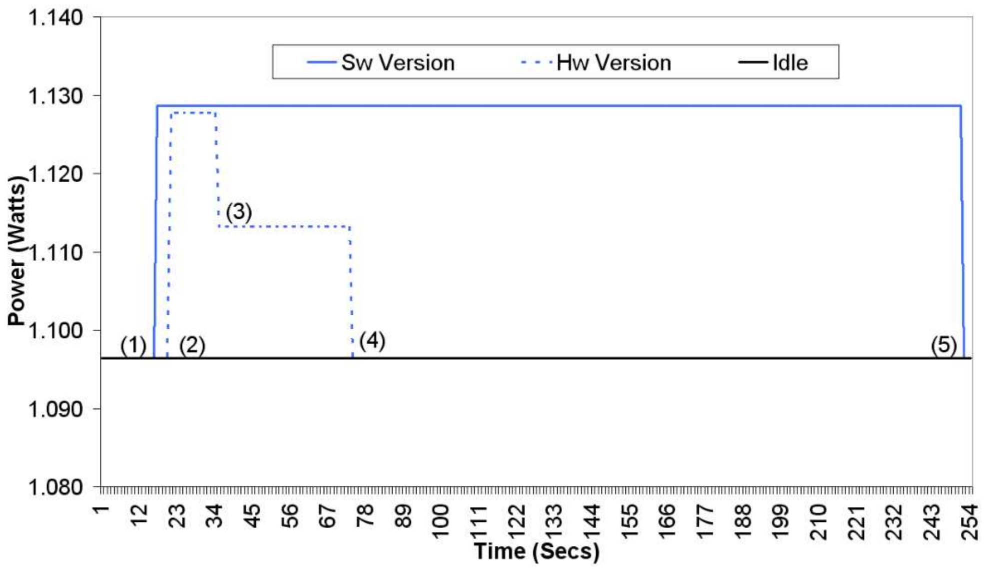


Figure 2.14: Suzaku power consumption diagram

ware 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).

2.8.2 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.

2.8.3 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.

2.9 Related Work

Reconfiguring the hardware at runtime is a very important attribute for hardware-software codesigned applications. Significant codesign efforts have already been conducted on exploiting the potential of reconfigurable hardware platforms. 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 methodology for codesigning applications along with corresponding development tool support is presented in [24]. 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 [25] 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.

Moreover a lot of research has been conducted on FPGA-based architectures and development tools for dynamic reconfiguration support. The first step has been to enable partial reconfiguration through corresponding partial bitstream generation tool capabilities [26][27][28], then to change FPGA architecture design so that it can retain the rest of the logic active while it is being partially reconfigured[29][30][31][17]. It must be stressed that dynamic partial reconfiguration (DPR) is still an active field of research. For the time being there are several problems [17] 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 [17].

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 [32]. 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 [33] 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 [34] 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, at the application-level, for a multi-node architecture is presented in [35]. 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 proposed 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). As we have demonstrated, 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. Our suspend-resume model is kernel-level and is applied on the entire SoC and the 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 targeted 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 [36], 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.

2.10 Summary

In this chapter 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.

2.11 Availability

More information about this work and uClinux patches are all available at *http://www.syrivelis.net/vss*

Chapter 3

An OpenMP-based Programming Framework for PR Processor Arrays

We present development and runtime support for building application specific data processing pipelines out of sequential code, and for executing them on a general purpose platform that features a Reconfigurable Parallel Processor Array (RPPA). Our approach is to let the programmer annotate the source of the application to indicate the desired pipeline stages and associated data flow, with little code restructuring. A pre-processor is then used to transform the annotated program into different code segments according to the indicated pipeline structure, generate the corresponding executable code, and produce a bundled application package containing all executables and deployment information for the target platform. There are special mechanisms for setting up the application-specific pipeline structure on the RPPA and achieving integrated execution in the context of a general-purpose operating system, enabling the pipelined application to access the usual system peripherals and run concurrently with other conventional programs. Moreover, we have extended the framework support to enable seamless task restructuring and load balancing of the produced pipelined application at runtime, making it possible to dynamically pick the stages that will be executed as separate tasks on distinct RPPA CPUs depending on the currently available resources and the execution context. This support has been implemented on top of two prototype soft RPPA FPGA-based platforms, and has been integrated into a Linux OS environment. Two proof-of-concept applications and indicative performance measurements for a variety of use scenarios on the prototype platforms are also discussed.

3.1 Introduction

The advent of embedded distributed memory RPPA platform solutions like Ambric [7] introduces new workload acceleration possibilities. Application logic can be now implemented as a program that executes on a CPU-based subsystem, which in turn plays the role of an application-specific co-processor for the platform main CPU. However, developing applications that can exploit the potential of such a system is far from trivial. One encounters most of the challenges faced when trying to write parallel programs for conventional multi-processor systems; above all, to structure the code in a way that enables its efficient parallel execution on top of the underlying hardware. Moreover, the RPPA CPUs may not have enough resources to run a complex runtime with support for multi-threaded execution, thread placement and thread migration. It is also important to reuse the existing (sequential) codebase of applications instead of developing them from scratch for this particular type of system.

As far as multitasking is concerned, traditional operating systems for shared memory multiprocessors achieve parallel execution by distributing tasks among processors using information such as processor idleness, task priority and the remaining quanta of tasks. At the level of the application, several models together with corresponding toolchains and/or libraries have been introduced to let the programmer define the multi-tasking structure most appropriate for the computation to be performed. Currently available support typically covers three basic types of parallel computations: (i)independent execution; (ii) data parallelism; and (iii) pipelined execution.

In the general case, task/data partitioning is non-trivial as it depends on the nature of the computation, and usually requires considerable programming experience in order to be effective. As a rule of thumb, a fine-grained task partitioning provides more potential for parallelism. Still, having *too* many tasks may degrade performance because of the increased communication and coordination overhead. When the number of CPUs is less than the number of tasks, more overhead is introduced due to context switching. An aggressively multi-tasked application may also get more time quanta than applications that employ a smaller number of tasks, having a negative impact on their individual response times. Ideally, besides the load balancing provided by the operating system at the task level, it would be desirable to adjust the task structure at the application level, at runtime, as a function of the current system state, i.e., task workload and available processing and communication resources. This is more or less straightforward to do for data parallel computations, where it suffices to adjust the number of (identical) tasks that process different chunks of data. On the contrary, it is quite hard to achieve in the case of pipelined execution.

Supporting parallel computing becomes even more challenging in the context of RPPA systems. Contrary to fixed multi-core hardware, such platforms have the advantage of flexible customization, namely each processor core can be fine-tuned for a certain application task, and the links between the cores can be arranged to suit the application-specific communication pattern between tasks; the latter is particularly relevant for computations with a priori known communication structures, such as custom pipelines. Moreover, reconfigurable hardware makes it possible to change the softcore architecture and interconnect dynamically, according to the current application workload.

In this chapter, we present work on the dynamic re-structuring and balancing of pipelined computations on reconfigurable RPPA systems, using a combination of application-level and system-level support that integrates transparently with the existing general-purpose operating system mechanisms. Our approach is to let the programmer annotate the source code of the application to indicate the desirable full-fledged pipelined execution structure, and then employ special tools to transform the source code into different segments that are compiled for execution on the target platform. Once the application is deployed, the specified pipeline stages are dynamically separated or merged, to execute on a different or the same task/CPU, respectively, as a function of their relative processing weight and overall workload. The current implementation is geared towards special pipeline-oriented softcore CPU networks and runs on a prototype system implemented on Parallel Reconfigurable soft-processor arrays that have been deployed on a Suzaku Xilinx Spartan-3 and a Memec Xilinx Virtex-II Pro FPGA boards. However, our approach is quite generic and the respective tools largely platform-independent so that they could be used (with the proper extensions) to support the dynamic balancing of computations with a pipeline structure on other multi-core platforms, including conventional SMPs.

The rest of the chapter is structured as follows. We start by giving an overview of our approach and the corresponding application development cycle. Next, we present key elements of our programming framework and system-level support designed to achieve the desired dynamic task restructuring and load balancing. We also discuss the parallelization and performance of a proof-of-concept application. Finally, we compare with related work.

3.2 Overview

Our high-level objective is to support the pipelining of applications on top of reconfigurable hardware in a straightforward and efficient way, exploiting the ability to setup customized softcore CPUs and interconnection networks that best fit the characteristics of the computation. In addition, we wish to achieve this in the context of a conventional multitasking operating system that is used to run other (conventional, non-pipelined) applications as well, concurrently to the pipelined applications.

Towards this end, we adopt a heterogeneous architecture that comprises a *main CPU* which is used to run a conventional operating system, and a network of dedicated *coCPUs* for running the stages of the application pipeline. The CPUs have strictly private memories which can be accessed efficiently without any contention. This architecture is implemented on a homogeneous processor architecture on the Suzaku Xilinx Spartan FPGA board (Xilinx Microblaze Architecture) and on a heterogeneous processor architecture on a Xilinx Virtex-II Pro FPGA board. In the latter case, a hardware PowerPC plays the role of the main CPU. The execution of the application-specific pipeline is done in both cases via a set of Xilinx Microblaze softcores which are appropriately linked to each other and the main CPU. The network of softcores can be installed on the FPGA at runtime using the support we described in chapter 3.

In terms of programming methodology, we introduce annotations for instrumenting the sequential source code to specify the desirable pipelined execution structure and data flow between the various stages. These primitives are introduced as extensions of OpenMP [13]. Even though OpenMP is targeted at shared memory architectures, we believe it is quite appropriate for our purpose as well. Most notably, it allows one to reuse existing (sequential) code with moderate code re-structuring.

This is quite important because the partitioning of an application may have to be adjusted several times during the development process.

A preprocessor is employed to transform the annotated code and partition it into separate compilation units, called segments, one for each pipeline stage. Subsequently, the preprocessor generates the executable code of each segment and produces an application package for the target system. This package includes a transformed sequential version of the original program that invokes our system-level support and contains code for dynamically changing the application-level task structure. We currently support two (radically) different platform types: (i) a process-based execution environment for a conventional Linux system, which is merely used as an emulator to guide/refine the partitioning of the computation; (ii) our FPGA-based prototypes, which are used as the "ultimate" target platforms where the application shall run. The preprocessor can be extended to support more target platforms, e.g., in principle it is possible to add "backends" for different multi-core systems (hard or soft, shared or distributed memory).

The typical development cycle is as follows: (1) the source code of the application is annotated to indicate the desired pipeline structure; (2) the preprocessor is used to generate an executable for the emulation platform; (3) the application is run on the emulation platform and profiling information is generated; (4) based on these results, one may wish to reconsider the partitioning of the application, going back to the first step; (5) the preprocessor is used to generate the final deployment package for the ultimate target platform; (6) the application is deployed and executed on the ultimate target platform.

Note that application partitioning in terms of identifying the separate pipeline stages is *static* and is decided by the programmer, e.g., based on the performance results on top of the emulation platform. It is typically designed assuming an *unloaded* system with enough *idle* CPUs to run the full-fledged pipeline. Task restructuring in terms of the pipeline stages that will actually execute as separate tasks, and ideally on different CPUs, is *dynamic*, based on the current system load and the speedup expected to be achieved vs the current task structure.

3.3 Annotations and code transformation

To harness the potential of setting up a customized data processing pipeline, the programmer explicitly indicates the desired pipeline structure and data flow, by annotating the source code of the application. The annotated code is then transformed into different segments that can be separately compiled for execution on different CPUs, each possibly having a different architecture or configuration that is most suitable for that code segment. The most essential aspects are discussed in the following.

3.3.1 Annotation primitives

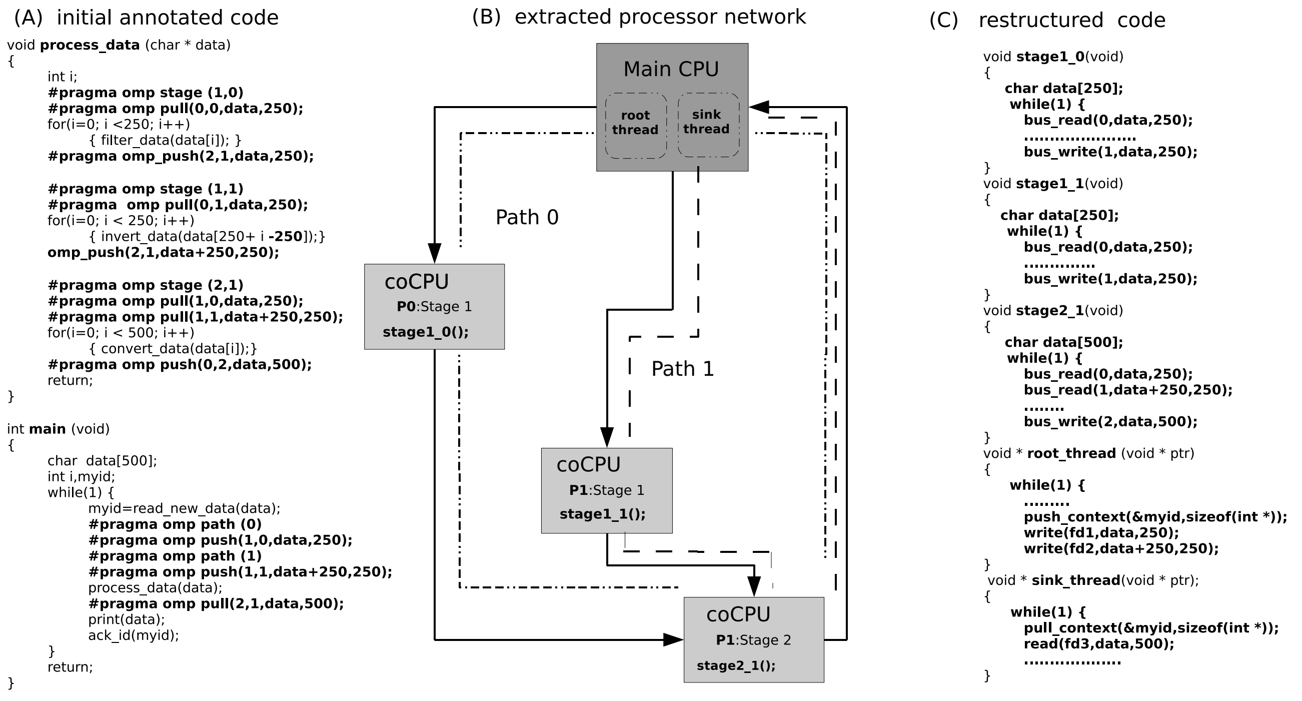
The annotations for marking the stages of the application pipeline and the data exchange between them are introduced as extensions of *OpenMP* [13]. They comprise two region definition directives, one declarative directive and two library calls, as follows.

The region directive *#pragma omp stage (<stage no>, <path no>, <function name>)* defines a pipeline stage. It takes as arguments the stage number (0 for the root and $N + 1$ for the sink of the pipeline, where N is the total number of the pipeline stages), the path number and the entry function name. The function name can be left blank if the directive is used inside a function body, in which case its location indicates the boundary between two stages.

The region directive *#pragma omp path (<path no>)*, taking as an optional argument a path number, is used to define a data flow path.

The declarative directive *#pragma omp threadprivate (<variable name>)* already exists in the *OpenMP* specification. In our case, it is used to identify the variables shared among the root and sink segments, for which different "copies" need to be managed properly (as will be discussed in the next subsection).

Finally, the library templates *#pragma omp push(<stage>, <path no>, <data pointer>, <size>)* and its counterpart *#pragma omp pull* are used to transfer data between two pipeline stages. The location of these annotations in the source code is of key importance, because they implicitly define a boundary between two or more stages (also the forks and joins of paths in the pipeline). Also, when a *pull* is used in isolation, without an accompanying *stage* or *path* directive, it indicates the end of the pipeline, i.e., the boundary with the sink segment.



```
void process_data (char * data)
{
        int i;
        #pragma omp stage (1,0)
        #pragma omp pull(0,0,data,250);
        for(i=0; i <250; i++)
                { filter_data(data[i]); }
        #pragma omp_push(2,1,data,250);

        #pragma omp stage (1,1)
        #pragma  omp pull(0,1,data,250);
        for(i=0; i < 250; i++)
                { invert_data(data[250+ i -250]);}
        omp_push(2,1,data+250,250);

        #pragma omp stage (2,1)
        #pragma omp pull(1,0,data,250);
        #pragma omp pull(1,1,data+250,250);
        for(i=0; i < 500; i++)
                { convert_data(data[i]);}
        #pragma omp push(0,2,data,500);
        return;
}

int main (void)
{
        char  data[500];
        int i,myid;
        while(1) {
                myid=read_new_data(data);
                #pragma omp path (0)
                #pragma omp push(1,0,data,250);
                #pragma omp path (1)
                #pragma omp push(1,1,data+250,250);
                process_data(data);
                #pragma omp pull(2,1,data,500);
                print(data);
                ack_id(myid);
        }
        return;
}
```

```
void stage1_0(void)
{
    char data[250];
     while(1) {
        bus_read(0,data,250);
        ......................
        bus_write(1,data,250);
}
void stage1_1(void)
{
   char data[250];
    while(1) {
        bus_read(0,data,250);
        ..............
        bus_write(1,data,250);
}
void stage2_1(void)
{
    char data[500];
    while(1) {
        bus_read(0,data,250);
        bus_read(1,data+250,250);
        ........
        bus_write(2,data,500);
}
void * root_thread (void * ptr)
{
     while(1) {
        .........
        push_context(&myid,sizeof(int *));
        write(fd1,data,250);
        write(fd2,data+250,250);
}
 void * sink_thread(void * ptr);
{
     while(1) {
        pull_context(&myid,sizeof(int *));
        read(fd3,data,500);
        ...................
}
```

Figure 3.1: Code annotation example

As a simple example, part (A) of Figure 3.1 shows the annotated code of a sample program, defining a pipeline with three stages (there are no branches, hence only one path is declared). Ideally, in a full-fledged configuration, the root, the sink and each stage of the pipeline execute as a separate program/task on a different CPU. For reasons discussed in Section 3.5, in our current FPGA-based platform, the root and sink execute on the main CPU, as depicted in part (B). Part (C) shows the most relevant parts of the code generated for each segment, after going through the transformation explained in the sequel.

3.3.2 Code transformation

Based on the location of the annotations in the source code, the program text is split into different parts, one for each pipeline stage defined. Notably, it is possible to split a function of the original program into many parts as well as to have a single part that includes the code of several different functions. The code and local variables of each part is packaged as an autonomous unit, called segment, that can be compiled separately; also for a different processor (softcore) architecture, if desired.

The integration of the pipeline with the rest of the application program is done based on the “entry” and ”exit” point(s) of the pipeline. Each of the functions called until the beginning of the first stage is reached, is split in two parts: the *pre-subroutine* and *post-subroutine* which contains code to be executed before pushing data into the pipeline and after pulling data out of the pipeline, respectively. Special root and sink code parts are then produced by grouping together the pre-processing and respectively post-processing subroutines in a suitably arranged call tree. The sink may include additional functions down the call chain, if the programmer decides to end the last pipeline stage inside a function body. Note that the root and sink segments are artifacts of the code transformation, rather than (real) pipeline stages. They are responsible for performing the necessary pre- and post-processing and feeding data into and retrieving data out of the pipeline, respectively. In our implementation, the preprocessor arranges for this code to be executed via two different threads that share memory (see respective body functions in part (C) of Figure 3.1).

Figure 3.2 illustrates the concept of function splitting. In this case, a call chain that comprises of seven nested function calls is annotated as to let the code of f4 and f5 be executed using four pipeline stages (without branches). Function f5 calls f6 but the pipeline end is defined before that call. As a result of this annotation, six different code segments are produced: the root segment containing the pre-subroutines of f1, f2 and f3; a segment for each indicated pipeline stage; and the sink segment containing f6 and f7 as well as the post-subroutines of f5, f3, f2 and f1.

3.3.3 Coding restrictions

Besides using annotations in the expected way, the application code must conform to a few additional restrictions. Firstly, function invocation should be *explicit* so that the preprocessor can properly resolve the respective call tree at source level. Any function pointers in the code must be replaced by hand. Secondly, system calls may be issued *only* from code that will execute on the main CPU, i.e., the root and sink segments. This is because, according to our architecture model, coCPUs are not connected to system peripherals and do not feature a full-fledged runtime environment.

One of the consequences of the latter restriction is that a pipeline segment is not allowed to invoke the standard dynamic memory allocation primitives. This is not as crucial as it seems though, for applications that are typically attractive to pipelining. In most cases, such programs simply allocate a fixed amount of memory in the beginning of the computation, hence the maximum required memory can be figured out by reading the source code. Else, a memory usage trace tool like *valgrind* [37] can be used to determine the memory requirements for typical inputs; but of course this does not guarantee error-free operation for all inputs. To make execution robust against memory overflows, one could introduce a dynamic memory management library especially crafted for the coCPU runtime, and let the preprocessor substitute the original invocations with calls to this API. So far we have managed to do without placing such support on the coCPUs, but this could be easily done, if needed.

3.3.4 Data passing and synchronization

To enable a pipelined execution of the application code, each local and global variable referenced in a function of the original program under the regime of sequential execution must be duplicated and updated in all segments, in a properly staged fashion. The programmer must explicitly forward the respective values downstream, via the *push* and *pull* primitives. These are replaced by the preprocessor with invocations to the actual data passing routines for the target platform (note in Figure 3.1 how the *push* and *pull* primitives in part (A) are replaced by platform-specific I/O calls in part (C), also depending on whether a segment will execute the mainCPU or a coCPU).

In platforms such as ours where CPUs do not share memory, it is obviously meaningless to transfer "raw" pointer values between two segments, instead the respective data objects must be transferred. To handle this problem, the preprocessor groups all pointers together with the rest of the variables to be transferred in a struct (pointers are placed at the beginning of the struct). This struct is then passed to the data transfer primitives, along with a special *pointer transfer structure* which contains the number of pointers to be found in the struct and an array with the size of the respective data objects, making it possible for the data transfer code to read/write data in an automated way (in essence, performing a scatter-gather I/O operation). If the preprocessor is unable to determine

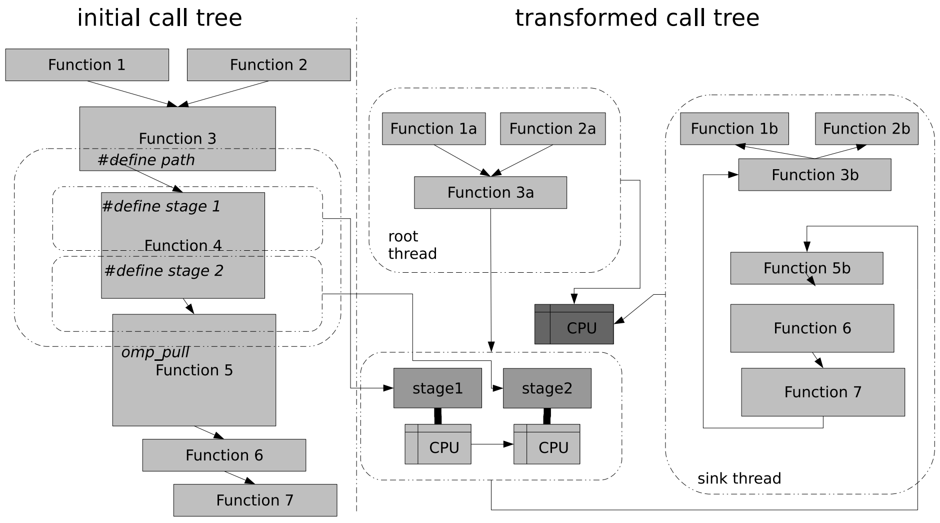


Figure 3.2: Function splitting

the size of pointed data at the source-level, the programmer is notified and he/she must add simple functionality that creates a proper *pointer transfer structure* instance.

For variables accessed *only* between the root and the sink segments, a separate context management mechanism is employed for keeping track of the values that correspond to different pipelined instances of the computation. When starting a new instance, before pushing data into the pipeline, the root adds a context entry with the proper values, and, conversely, the sink removes the next context entry before attempting to retrieve (and process) data from the pipeline (see respective body functions in part (C) of Figure 3.1).

The context management mechanism is accessed as an abstract data type via a library which hides its internals from the preprocessor. In our current implementation, where the root and sink execute on the same CPU via two different threads, we use a properly synchronized FIFO queue. This is similar in concept with the so-called versioned memory introduced for shared memory multiprocessors [38]. Obviously, a different implementation would be needed if the root and sink were to execute on different CPUs with separate memories.

3.4 Task restructuring

The developer specifies the pipeline based on the assumption that there are enough CPUs to host all the stages of the pipeline. Also, the partitioning is chosen as to evenly distribute the processing load among all stages, since it is the slowest stage that will ultimately set the throughput of the entire pipeline. However, the situation encountered at runtime may differ, i.e., there may not be as many CPUs available or some CPUs may be more loaded than others. As a consequence, keeping the full-fledged pipeline, as this was defined by the programmer, may not be efficient despite the load balancing performed by the operating system. This problem becomes even more relevant when it is not possible to rely on multitasking and task migration across all CPUs, as this is the case in our prototype platform (see Section 3.5).

As one possible solution to this problem, we decided to support dynamic task restructuring, at the application level, assisted via a separate service designed for this purpose.

3.4.1 Configuration options

To keep our implementation simple and easily applicable in our prototype FPGA-based platform, we restrict ourselves to the case where only consecutive stages execute as separate tasks (on different CPUs). More formally, assuming a pipeline structure with n stages $root- > S1- > S2- > ...- > Sn- > sink$, we support task structures with the following property: if the leftmost and rightmost stage that is executed as a separate task is Sk and $Sk + x$, respectively, then every stage $Sk + i$, $0 < i < x$, also executes as a separate task. In this case, the root task executes $root- > ...- > Sk - 1$ and the sink task executes $Sk + x + 1- > ...- > sink$.

This specification includes the strictly sequential execution where the entire computation is performed by a single task (the root and sink tasks coincide), as well as the full-fledged pipelined execution where all stages execute as separate tasks (the root and sink tasks execute only the root and sink segments, respectively). As an example, Figure 3.3 shows all such task structures for a 3-stage pipeline.

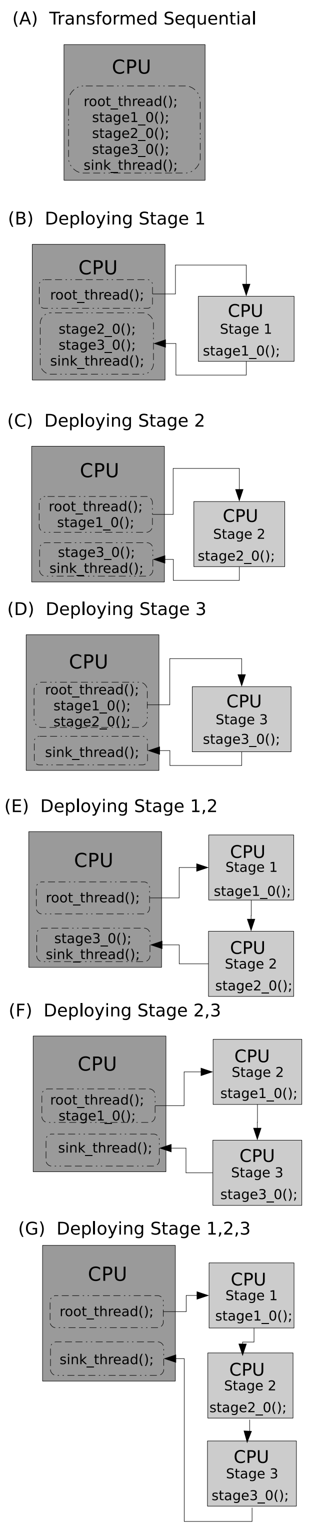


Figure 3.3: Possible execution configurations

3.4.2 Preprocessor support

The transformation of the annotated source code and the generation of executable code for each application segment is performed using a 2-pass preprocessor. In the first pass, the preprocessor builds the function call tree, determines the number of pipeline paths and stages, and marks the functions associated with the corresponding entry/exit points. It also performs basic sanity checks to verify that the annotations are placed in a logically correct order, contain proper arguments and that the data exchanges between the stages are consistent. If errors are found, a report is generated and the tool exits. In the second pass, the preprocessor makes the changes to the source code, i.e., splits functions as needed, adds calls to properly manage the context information between the root and sink segment, and arranges for their execution via two separate *POSIX* threads. Also, the abstract data transfer directives found in the various code segments are substituted with the appropriate code (library calls) for performing the necessary communication between the CPUs on the target platform.

A separate source file that can be compiled independently is generated for each segment. This makes it possible to employ a customized CPU architecture and different runtime environment for each pipeline stage, subject to the capabilities and flexibility of the target platform. In any case, the preprocessor produces the executable for each stage by compiling and linking against the corresponding runtime. Finally, all executables along with corresponding platform configuration information are bundled in a single deployment unit.

The processing of annotations and most of the code transformation is orthogonal to the particular characteristics of the target platform. The main platform-specific part is the code that needs to be injected for sending and receiving data between segments, and the compiler that needs to be invoked for generating the executable for each segment. This allows the preprocessor to be implemented in a structured way so that it can be extended to support more targets, e.g., by adding “backends” for different multi-core systems (hard or soft, shared or distributed memory). In fact, support for the emulation and prototype FPGA-based platforms, discussed in

Another challenge for the preprocessor is to enable the transparent switching between different task structures at runtime. To achieve this, it generates a special sequential version of the application code, where each stage is self-contained in a separate function. These functions are arranged in a proper call chain, so that the function for stage Sn invokes the function for stage $Sn + 1$, and is itself invoked from within the function for stage $Sn - 1$. In addition, the code of each stage includes indirect calls to *push* and *pull* function templates, via pointers stored in an array-based structure. This makes it possible to dynamically switch the data transfer functionality of a stage just by changing the value of the corresponding index variable. There are three I/O options: (i) send/receive data to/from a different task/CPU by invoking the corresponding platform-specific primitives; (ii) perform a dummy call when the previous/next segment executes within the same task context; (iii) invoke the profiling service (for *push* only). The data transfer mappings of each stage need to be properly set each time the task structure changes. The code for this is generated by the preprocessor. As a convention, *push* calls must return a code indicating whether the calling stage should proceed with the default sequential execution flow, i.e., invoke the function of the next stage, or return immediately. The latter is necessary if the next stage executes in a separate task/CPU, and the respective code is returned only when using the platform-specific primitives for inter-task/CPU communication.

The code for each stage under sequential execution is produced essentially by reusing the code produced anyway for autonomous execution via separate tasks/CPUs. However, to make the communication between stages that execute within the same task more efficient, their code is changed to operate on the same (shared) memory locations (rather than sending/receiving data via buffers via appropriate versions of the *push* and *pull* primitives). This is achieved by placing shared data in a globally allocated structure and letting the individual members thereof be accessed via a pointer. Two different versions of this structure, i.e., two separate copies of the data, are used for the stages that execute in the context of the root and sink task, respectively. These are arranged in a 2-slot array, and each stage accesses the right element via an index variable, which is properly set in case of restructuring to point to the first or second element of that structure, as needed. If the application executes in strictly sequential mode (the root and sink tasks coincide), the first element is used by all stages.

3.4.3 Monitoring and notification service

Task restructing is only meaningful when it is expected to boost performance. The application program must be informed about this in order to adjust the data transfer and memory mappings as well as to adjust the dynamic behaviour of each stage accordingly.

For this purpose, we introduce a service that keeps track of the current task structure, computes the time spent at each pipeline stage during sequential execution, determines whether a different task structure is expected to be more efficient than the one currently employed, and informs the application program about this new task structure. This is abstracted using an API with two primitives: an implementation for the *push* function template, and an *ioctl* function. Their role is described in more detail in the following.

The *push* function is invoked (via the transfer mapping mechanism) under sequential execution, to perform profiling. Its role is to record the time when the call is made. Given that the *push* calls are done at stage boundaries, this enables the service to profile the application pipeline at runtime to determine the processing load for each segment. The root and sink segments are indirectly profiled together by measuring the application usage time that elapses between the very first profiling *push* and the *ioctl* of the next iteration and subtracting the sum of all the pipeline stages recored usage time. This way the driver can figure out the maximum application workload percentage that can be offloaded to the pipeline. In turn, all the above information can be used to estimate the performance of the application under different task structures, and to decide whether a new structure should be employed. The profiling version is used only when the entire pipeline runs in sequential mode.

The *ioctl* function is invoked from within the root code, before calling the *push* function towards the first pipeline stage. It checks the current system load, decides whether a new task structure could be more beneficial and returns a corresponding code and mappings. If the return code does not suggest a restructuring, the root continues with the default flow of execution. Else, it switches to the new structure as follows. First, the current task structure is checked against the newly proposed structure to determine whether the stages executed within the sink task remain the same. If this is not the case, the sink task is notified via a flag (we remind that in our implementation the root and sink tasks execute as threads on the same CPU), and the root waits for the sink to exit. The sink

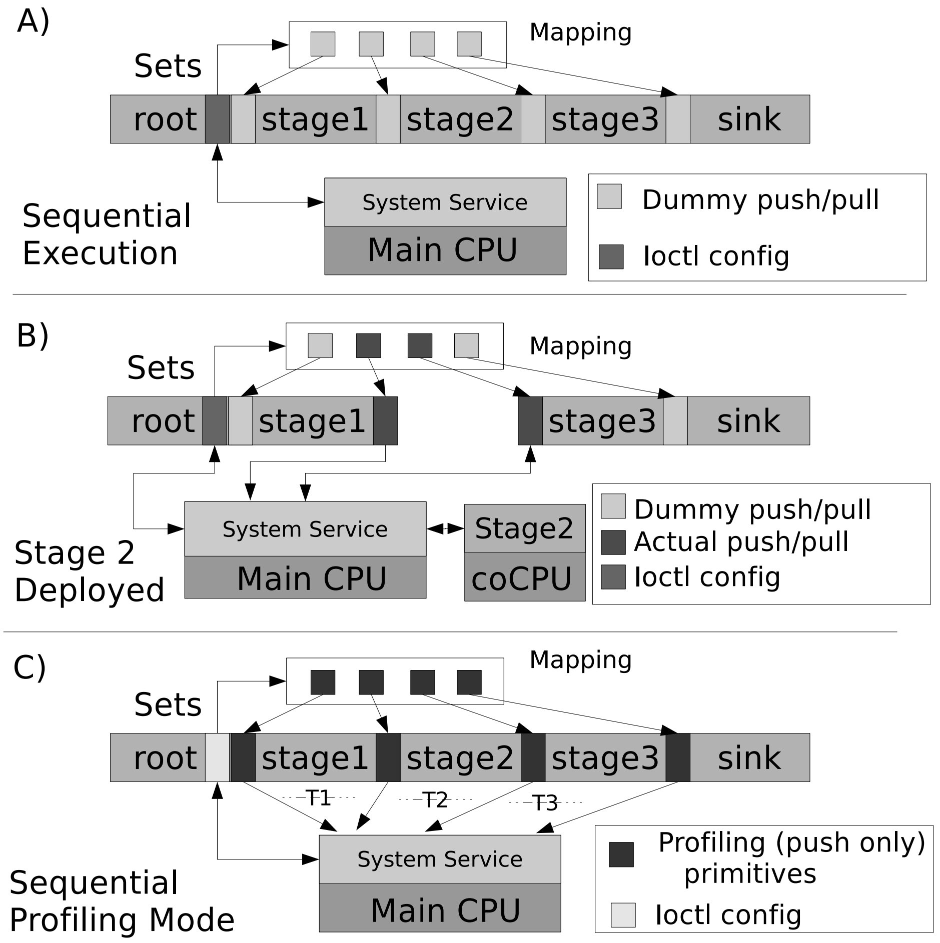


Figure 3.4: An illustration of task restructuring support for different modes of execution

task checks this flag each time the context queue (discussed in Section 3.3.4) is found empty, before blocking on the queue. If set, the sink exits; this is safe, since the entire pipeline is guaranteed to be idle at this point. When the root resumes, based on the information returned by the *ioctl* call, it adjusts the data transfer and memory mappings of the stages which shall execute in the new sink, and launches a new task that invokes the function of the leftmost stage that shall execute in the sink. After dealing with the (new) sink, the root adjusts the data transfer and memory mappings of the stages that shall execute in its own context, invokes (once again) the *ioctl* function to confirm the transition to the new task structure, and proceeds with the default flow of the execution.

Figure 3.4 shows three indicative execution configurations for a 3-stage pipeline application, together with the corresponding data transfer mappings. In first case (A) all stages execute sequentially within the root task on the main CPU, with the default (dummy) data transfer behaviour. In the second case (B) the second stage is set for execution (as a separate task) on a different coCPU, and the output mapping of the first stage as well as the input mapping of the third stage are set to invoked the appropriate communication primitives to send/receive data between the main CPU and the coCPU). (C) is identical to (A), but in this case the data transfer mappings are set to perform application profiling, i.e., record the time spent in each stage.

3.5 FPGA-based Prototype Platforms

The presented design is quite generic so that it can, in principle, be supported on top of different multi-core platforms. This requires introducing appropriate “backend” extensions to the preprocessor as well as providing a suitable implementation of the system-level service functionality. We have developed support for 2 radically different platform types and 2 variants of the second type: a process-based execution environment for a conventional Linux system which is mainly used as an emulation and debugging tool, a prototype entirely FPGA-based SoC and a prototype FPGA-based platform that also has a typical high performance hardware CPU which are both used as the ultimate target system. In the sequel, we discuss the most important aspects of the FPGA platform imlementations.

3.5.1 Basic setup on the homogenous platform

The first prototype platform is entirely softcore, prototyped on an FPGA and is used to install multiple identical softcore CPUs as well as arrange the interconnections between them. This platform is the Atmark Techno Suzaku [20], featuring a Xilinx Spartan 3 FPGA along with off-chip 16MB RAM, 8MB flash, a MAC core and a configuration controller. For the softcore main CPU and coCPUs we choose the Xilinx Microblaze, operated at $100Mhz$.

The main CPU is interfaced to all platform peripherals and runs a customized version of uClinux [39]. The coCPUs have access only to their local memory and are connected to each other and the main CPU using FSL links (which will be described in the sequel), reflecting the topology that is specified by the application developer. The runtime for coCPUs is a small basic input/output system we have developed that must be statically linked to the application code to be loaded on the coCPU (the coCPUs do not feature a proper runtime nor any kind of support for multi-programming, multi-threaded execution, thread migration, etc). The BIOS is about 512 bytes long, keeping the memory requirements of coCPUs low; of course, the amount of memory each coCPU will ultimately be equipped with depends on the complexity of the pipeline stage that will run on it.

As already mentioned, the main CPU serves both as the root and sink of all pipelines. It must also run all other (conventional) applications. Given this bottleneck situation, it is crucial to avoid any busywaiting on the main CPU while waiting for the pipeline to get ready to accept data and/or deliver data. For this reason, we developed an FSL *Slave Interrupt Generator* (SIG) peripheral that is interfaced to the *first* and *last* coCPU of each pipeline and the Interrupt Controller of the main CPU. The interrupts are generated by the coCPUs using standard FSL instructions, when they are ready to accept data from and respectively deliver data to the main CPU. On the uClinux side, we developed a *FlowControl* driver that handles the respective IRQs. Applications register with this driver in order to get notified via a SIGUSR1 signal as soon as it becomes possible to push/pull data into/from their pipeline.

3.5.2 Basic setup on the heterogenous platform

Our second prototype hardware platform is implemented on a Memec Design 2VP7 Evaluation Board with a Xilinx Virtex-II Pro FPGA and off-chip 32MB RAM, 16MB flash, a configuration controller and several other peripherals. The FPGA includes a hardware PowerPC 32-bit big-endian core that operates at $300Mhz$, which we use as the main CPU of the system. For the sofcore coCPUs we choose the aforementioned Xilinx Microblaze architecture.

The main CPU runs the full-fledged Linux kernel ver 2.4.30. The userland includes custom support for performing a full runtime reconfiguration of the FPGA in order to setup the coCPUs and pipeline interconnect for a given application. The coCPUs run a simple basic I/O system, which we developed for our own purposes. Notably, the coCPUs neither enjoy proper runtime support nor are able to access various system peripherals as with the first prototype. As a consequence, in this case also the parts of the application that need to access special devices must execute on the main CPU.

This goes also for both the root and sink, which are responsible for loading/unloading the pipeline and interacting with the rest of the system, e.g., perform some I/O with devices. For this reason they are executed via two separate *POSIX* threads on the main CPU. Last but not least, a pipeline stage, which will potentially execute on a coCPU, may not rely on special runtime functionality and/or peripherals as with the first platform.

3.5.3 CPU interconnect

The Microblaze coCPUs communicate with each other on both platforms using an already supported fast bus architecture, called FSL (Fast Simplex Links). This is a dedicated 32-bit wide unidirectional point-to-point communication channel, which does not need arbitration, provides hardware support to distinguish between data and control communication, and has a 64-byte FIFO depth for asynchronous access. The CPU transfers data directly from the FSL to the register file, and vice versa, using dedicated instructions. The current Microblaze implementation supports up to 8 read and 8 write FSL interfaces, making it possible to support rather complex pipeline topologies.

Unfortunately, in our second platform the main PowerPC CPU does not support FSL. For this reason the data transfer between the main CPU and the first-stage and last-stage coCPUs, or *border* coCPUs, is implemented using dual port Block RAMs (BRAMs), a special FPGA resource dedicated to implement RAM. One port is connected to the PowerPC PLB bus (and is mapped accordingly) while the other is connected to the Microblaze data memory bus. This allows the PowerPC to write/read directly to/from the Microblaze data memory. The PLB BRAM interface supports burst transfers for consecutive memory addresses and can achieve a throughput of up to $95MB/sec$, matching the speed of the FSL bus.

Therefore in the second platform the synchronization between the main CPU and the border coCPUs is achieved as follows. On the side of the Microblaze, a spinlock is located at a predefined memory address, which is polled to detect the readiness of data. The lock is released by the main CPU when data transfer finishes, i.e., when the root and sink threads are done writing and reading, respectively. To avoid polling on the main CPU, we have developed an FSL *Slave Interrupt Generator* peripheral which is interfaced to the border coCPUs and exports interrupt lines that are connected to the interrupt controller of the main CPU. The interrupts are generated by the border coCPUs using standard FSL instructions when they are ready to accept/deliver data from/to the main CPU. On the Linux side, we have developed a driver that registers service routines for the respective IRQs. The platform-specific *push* and *pull* implementation blocks on this signal before performing the data respective transfer. This version is invoked from within the right code segment, at the rightmost stage that executes within the root and the lefmost stage that executes within the sink, based on the current data transfer mapping.

3.5.4 Preprocessor support

Based on the pipeline structure defined by the programmer, the preprocessor generates the interconnection topology for the main CPU and coCPUs, along with the corresponding fast bus channel numbers and data transfer directions. Also, the high-level *push* and *pull* annotations are replaced with suitable calls for accessing the FSL in the context of independent stages (for the communication between coCPUs) and/or accessing the proper data memory addresses subject to the aforementioned synchronization scheme (for the communication between the main CPU and the border coCPUs).

The binaries for the different code segments are generated for the respective CPUs and runtime environments. More specifically, the binary for the sequential stage execution within the root and/or sink threads is generated for the main CPU and is linked to the proper Linux system image and libraries, including our system-level service. The binaries for the autonomous pipeline segments are generated for the Microblaze coCPUs and are linked with reference to their local address space and basic I/O system.

We have also developed a special platform mapper tool, which uses the initial platform description report to build a single deployment file, akin to the elf format, containing all the binaries as well as the description for the coCPU configuration and respective stage load. The tool determines the required local memory size for each coCPU and transforms the initial platform description file to a Microblaze Hardware Specification (MHS) file. This is then used by the Xilinx Platform Studio toolchain to build all the possible platform combinations and produce respective bitstreams.

3.5.5 Hardware reconfiguration and application loading

Application loading on both platforms is done using a custom program. The loader reads the MHS stored in the deployment file and passes respective information to the system service that is also responsible for managing the hardware reconfiguration. If the required coCPU network is not already installed, the system reconfigures and resumes with the requested configuration deployed on the FPGA, properly connected to the main CPU. The FPGA runtime reconfiguration approach and corresponding subsystem is based on existing support that we have built for the Microblaze uClinux [40] which we have ported to the PowerPC Linux environment (described in chapter 2).

When the loader is informed that the CPU network has been successfully installed, it reads from the deployment file the binary for each pipeline stage and copies it to the system service address space. The latter uses the binary images to load the respective coCPUs according to the desired configuration. This is done using a simple protocol supported by a first-stage loader we have developed, which is pre-installed on the coCPUs. Once loading of the coCPUs completes, the binary for the main CPU is loaded using the standard OS procedure, and the execution of the application commences or continues.

3.5.6 System service and load balancing issues

The required service functionality, as discussed in Section 3.4.3, is implemented as a system-level driver. The application loader is the first that connects to this driver in order register the application by passing the binary format header information along with the target coCPU binary images of each pipeline stage. Within the driver context, each application instance is associated to its configuration data (state and measurements) via the root thread id (note that the root thread remains the same even when changing between different task stuctures).

By default, the application is initially deployed to execute in strictly sequential mode, with the profiling version of the *push* enabled. This uses the linux kernel *task_t* cpu load to record the processing load of each pipeline stage. When profiling completes (currently this is done just for a single iteration), the driver automatically changes the data transfer mapping of the application to use the dummy *push* version. Note that the application code may ship with information about its expected stage load (our deployment file format allows for such information to be included and for the driver to be informed accordingly). However, this may be inaccurate for several reasons, e.g., derived from emulation and/or different input scenarios, so that ability of the driver to profile the application during the actual execution remains crucial.

Each time the application calls *ioctl* function, the driver accesses the linux kernel *kstat* data to determine the current main CPU usage. Together with the performance profile of each application pipeline, this information is used to decide about possible task restructurings to improve overall workload performance. Since this function is invoked in each iteration, it cannot afford to employ a complex algorithm. A platform-specific heuristic is used, taking into account the possible performance heterogeneity of all the platform CPUs. The main goal of this algorithm is to offload the main CPU as much as possible while keeping the pipeline coCPUs loaded and synched. To achieve the latter, for a given workload, the pipeline computations that are well balanced and usually require less coCPUs are fully deployed in order to maximize the respective coCPU utilization. Of course, this approach results in uneven distribution of resources and unfair response for some applications, but from the multitasking workload perspective the overall response is as good as it can get. Note, that in cases where there are not sufficient resources, if we choose to execute the arriving pipelined applications sequentially one after the other rather than concurrently, the overall response of the workload can be better than any other mixed execution combination because the coCPUs will be optimally exploited under the consequtive well balanced executions rather than when coCPU loads are assigned to the main CPU. This is because when the latter is overloaded it will be unable to feed the rest of the pipeline coCPUs with data on time, which will result in respective stalls. On the other hand, from a user perspective and in terms of multitasking usage scenarios, the sequential execution of arriving applications is highly unlikely to be acceptable, exactly as it happens with time-shared general purpose systems. It is all a matter of tradeoffs and different resource distribution algorithms can be designed depending on the platform target use.

On the other hand, when the main CPU is faster than the coCPUs other performance concerns arise as well. More specifically, in our second platform the main CPU runs at a 3 times faster clock rate than the coCPUs, has a more powerful instruction set and uses a separate 16KB data and instruction cache (it was not possible to use caches in the Microblaze softcores due to shortage of BRAM resources). As a rough conversion factor in terms of CPU performance, we use the Linux BogoMips [41] algorithm for both CPUs on the same platform, according to which the PowerPC is $3,71x$ faster than the Microblaze. Also, pipelined execution comes with additional memory allocation and data transfer overheads. Assuming an balanced load distribution, this means that up to 3 stages will run faster on the main CPU in the sequential configuration rather than on a coCPU pipeline. Of course, this situation changes as the main CPU gets loaded with additional tasks, in which case it might be beneficial to deploy (some of) the application pipeline stages on coCPUs.

To improve coCPU syncing and utilization, in the case where a full-fledged pipeline execution is chosen, as an optimization, the driver gives high-priority to both the root and sink threads, essentially treating them as user-level interrupt routines. This is done to avoid pipeline stalls, based on the assumption that the bulk of processing is performed by the pipeline stages whereas the root and sink are merely perform I/O with system peripherals. On the contrary, if even a single pipeline stage executes in the context of the root or sink, the driver resets the priorities to the default. It must be noted, however, that this special treatment of full-fledged pipelines may lead to the *starvation* of the rest of the application processes, if the pipeline matches the execution speed of the root and sink on the main CPU.

3.6 Proof-of-concept applications

We have used our framework to develop and test two applications: i) a pipelined version of *Tremor*, a fixed-point version of the *Ogg Vorbis* decoder [42] targetted at integer cores and ii) a custom ASCII-art rendering algorithm that operates on a NxN character frame and changes character values. Both programs are written in C and already run on Linux. Next, we describe how we partitioned the code to produce the different pipeline stages, and we separately discuss indicative measurements which were conducted on both our prototype FPGA-based platforms.

3.6.1 Profiling and partitioning of *Tremor*

As far as *Tremor* is concerned first step was to compile it with profiling extensions for a normal Linux environment and run it to decode a sample file. *GNU prof* was used to analyze the generated information, revealing that 80% of the CPU cycles are spent on the inverse discrete cosine transform (DCT) related functions. The rest of the CPU cycles are spent on file processing and soundcard interaction, but on our target platform coCPUs cannot be used to reduce the execution time of tasks that access system services and devices. The DCT-related processing comprises two main parts, the butterflies calculations and the bit reverse calculations, which can be naturally processed via two different pipeline stages. To better balance the workload, the first calculation can be further separated in two parts, giving a total of three stages. The lines of code added and the size of the transformed source is reported in table 3.1

Next, we changed the *Tremor* source code by removing function pointers (which are not properly handled by the preprocessor) and annotating the code to define the desired pipeline structure. The preprocessor was then used to generate executables for the emulation platform. The profiling information we got when running this code with a variety of input sample files indicated that the root and sink segments consumed 20% of the total workload. The three pipeline segments accounted for 28%, 25.4% and $26,6\%$, respectively. Moreover, these exchanged almost 50% less amount of data with each other than with the root and sink segments, because of data (de)compression. The reported communication to computation ratio is $1.8 * 10^{-2}$ for the root and sink program segments, $1 * 10^{-2}$ for the first and third pipeline stage, and $0.8 * 10^{-2}$ for the second pipeline stage.

Given this profile, the pipelined version of Tremor would be expected to achieve a $3.57x$ speedup on a homogeneous 4-processor system implemented on the first platform. On the other hand, taking the second platform CPU heterogeneity into account, if all 3 pipeline stages are deployed on coCPUs the expected speedup will be actually zero and equal to $1x$. Note that, if for example the second pipeline stage is deployed on a coCPU and the rest run on the main CPU the expected speedup will be $1.06x$ which means that dynamic load balancing has the potential to make a difference. To further investigate the heterogeneous platform we have executed tremor along with dummy workload.

Application Code Changes		
	Tremor	*ASCII-Art renderer*
Total lines of code	8244	1232
lines of added annotations	15	19
lines of additional generated code	135	147

Table 3.1: Code transformation details for *Tremor* and *ASCII-Art renderer*

3.6.2 Profiling and partitioning of *ASCII-art renderer*

In order to explore the dynamic load balancing effects on the homogeneous processor platform, we have also used the proposed framework to transform a C sequential version of a simple *ASCII-art* rendering algorithm and execute it concurrently with *Tremor*. This algorithm operates in a loop on an 100-character frame that resides in ram and the main computation can be easily divided to four equally loaded parts. The lines of source changes are reported in 3.1. The root and sink segments in this case comprise the 10% of the total workload. On our emulation platform the communication to computation ratio was $8 * 10^{-2}$ for the root and sink segments and $6 * 10^{-2}$ for the rest of the stages. Therefore, on a 5-processor system (master cpu and 4 coCPUs) the profiler indicated that the expected speedup would be around $3.3x$ compared to a highly optimized sequential application with the master CPU being practically unloaded. Note that, since 10% of the computation has to precede and cannot be accelerated, the theoretical upper acceleration bound with 4 equally loaded processors working concurrently is $3.6x$ compared to the optimized sequential version. The main reasons for the profiler deviation from the theoretical value are the computation communication ratio, which is not negligible and at least 6 times higher than the *tremor* partitioning case, and the fact that the stage load is slightly imbalanced. During the deployment on the target platform we expect the actual speedup to be slightly less than the profiler reported value.

3.6.3 Experiments on the homogeneous Platform

In the first experiment we measure the time for executing the sequential versions of the pipelined applications that were produced using the preprocessor and compare it to the original Tremor and *ASCII-art* programs. The transformed version in the case of tremor introduces an overhead of 2.6% compared to the original sequential version while in the case of the renderer it is 1.3%. The main

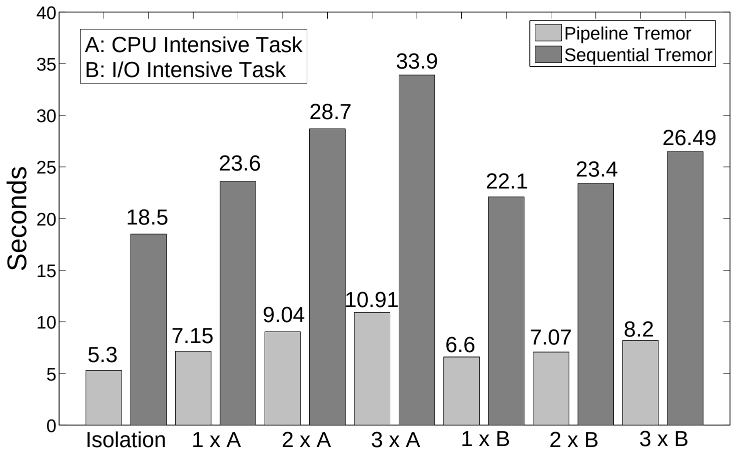


Figure 3.5: Performance of pipelined vs sequential *Tremor* for isolated & mixed executions

overhead is introduced by the context queue mechanism operation which in the case of tremor is more intrusive.

We have measured the performance of the original sequential version vs the pipelined version of *Tremor* for decoding an indicative input file holding 102500 data samples (the file resides on the flash). Firstly, we want to outline the performance impact of concurrent workload only on the main CPU and we use two types of dummy tasks: type A that performs CPU intensive computations, and type B that performs periodic I/O access to the flash (reads-writes for $50ms$, every $100ms$). Note that flash I/O is CPU driven, using polling, thus it also occupies the CPU. The execution time of each such task is set to $5secs$, approximately matching the execution time of the pipelined *Tremor* version. Figure 3.5 shows the times for executing Tremor in isolation as well as concurrently with one, two and three instances of each task type. The recorded utilization of the coCPUs (for the pipelined version) during these executions is shown in Figure 3.6.

When executing in an unloaded system, the pipelined version is $3.49x$ faster than the sequential version, which is close to the $3.57x$ speedup that was estimated using the profiling data of the emulation environment. The performance of the pipelined version drops when loading the main CPU with extra (dummy) work, with CPU intensive tasks having a more negative effect than I/O intensive tasks. This degradation is confirmed by the respective idleness of the pipeline stages; as the main CPU gets increasingly loaded, the number of pipeline stalls (failure to push data into or

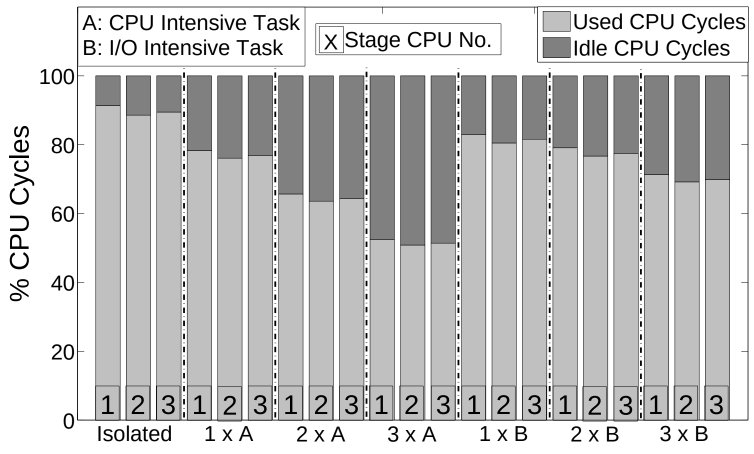


Figure 3.6: CPU utilization of the pipelined *Tremor* for isolated and mixed executions

pull out of the pipeline) grows as well, leading to more idle cycles in the coCPUs. Notably, the extra load on the main CPU also affects the performace of the sequential version, but (in our experiments) this impact is less noticeable than for the pipelined version because dummy tasks finish before the completion of the sequential version, which can fully exploit the main CPU thereafter. Overall, the pipelined version outperforms the sequential version as expected, but it is also sensitive to other tasks executing on the main CPU.

Taking a second look at the results of the pipelined version in the case of isolated execution, it can be seen that the idleness of the coCPUs closely follows the load distribution recorded using the emulation environment for an unloaded system. However, contrary to the profiling data which indicate that the first stage is the bottleneck of the computation, i.e., works at full speed, in reality the first stage coCPU remains idle for 8.6% of the execution time. Further investigating into this matter, we confirmed that the *root* thread is burdended with extra overheads such as time consuming memory allocation on our MMU-less platform and access to the flash (concurrently to the *sink* thread), which are not captured in the emulation environment. As a consequence, the main CPU becomes the actual bottleneck and cannot feed the pipeline fast enough, leading to the idleness of the first stage coCPU. This also explains the difference between the estimated vs actual speedup. Of course, this situation deteriorates as the load on the main CPU increases.

The actual performance of the *ASCII-art* renderer when executed in isolation, on the 5-core plat-

	Pipeline Application Stages (% of the Total Workload)							Total
Configs	$Trm1$ (14%)	$Trm2$ (12.7%)	$Trm3$ (13.3%)	$Art1$ (11.2%)	$Art2$ (11.2%)	$Art3$ (11.2%)	$Art4$ (11.2%)	Main ld (X%)
Cfg 1	coCPU	coCPU	coCPU	coCPU	Main	Main	Main	48.6%
Cfg 2	coCPU	coCPU	Main	coCPU	coCPU	Main	Main	50.1%
Cfg 3	Main	Main	Main	coCPU	coCPU	coCPU	coCPU	55%
Cfg 4	Main	coCPU	coCPU	coCPU	coCPU	Main	Main	51.4%

Table 3.2: Possible coCPU assignment configurations that were considered for the concurrent execution of *tremor* and *ASCII-art* renderer.

form, verifies the previously discussed observations with *tremor*. Originally the highly optimized sequential version can process 83 frames per second while the accelerated version managed to process 265 frames per second. The actual speedup is $3.19x$ compared to the highly optimized version. In this case, the main CPU executes 10% of the computation load at the beginning of execution and then it remains practically unloaded.

In the context of a simple ASCII-based game, we want to execute *tremor* and *ASCII-art* renderer concurrently. We started both pipeline versions with the *ioctl* call enabled during each iteration. The Suzaku Xilinx Spartan FPGA, unfortunately cannot host more than 5-cores due to scarce *Block RAM* resources. Therefore, the dynamic load balancing system support tries to offload the main CPU following the previously discussed simple resource distribution algorithm. In table 3.2 we present all the configurations that were considered by our simple algorithm. Note that the load percentages for each pipeline stage and the main cpu in this table are calculated with the total workload as a reference. The choice was configuration 1 where *tremor* stages are fully deployed and the remaining coCPUs are assigned to the rest of the system. In order to have have a common metric reference, we configured the iterations *ASCII-art* execution to last exactly the same time as the *tremor* sample file sequential processing.

Figure 3.7 shows the execution times on a 5-core RPPA for all the configurations given in Table 3.2. The execution times of the isolated sequential and pipelined executions and concurrent sequential executions of both programs are given as an additional reference. Figure 3.8 depicts the utilization of the coCPUs for all configurations given in Table 3.2 as well as the isolated pipelined executions on an idle system. Note that the recording of the utilization for each coCPU in this figure takes place until the completion time of the application that is using it in each configuration. As it

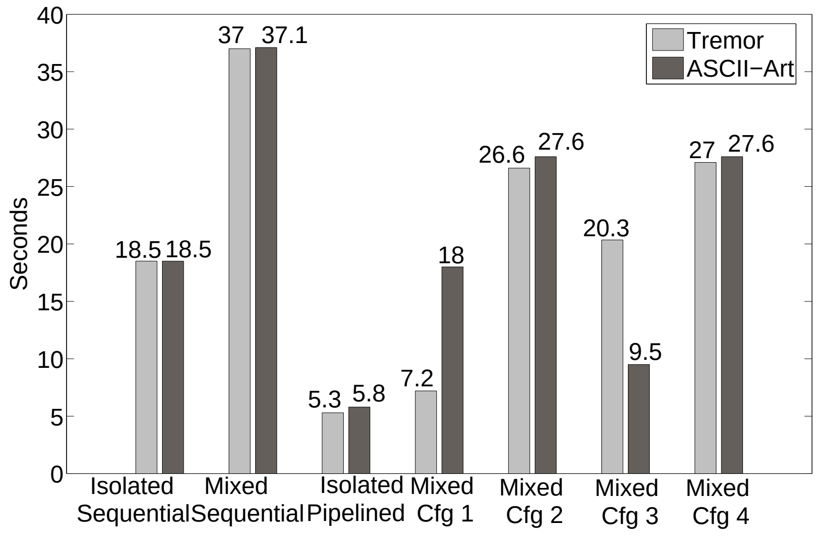


Figure 3.7: Performance of *Tremor* and *ASCII-Art* renderer in isolated and mixed execution scenarios for first three configurations of table 3.2

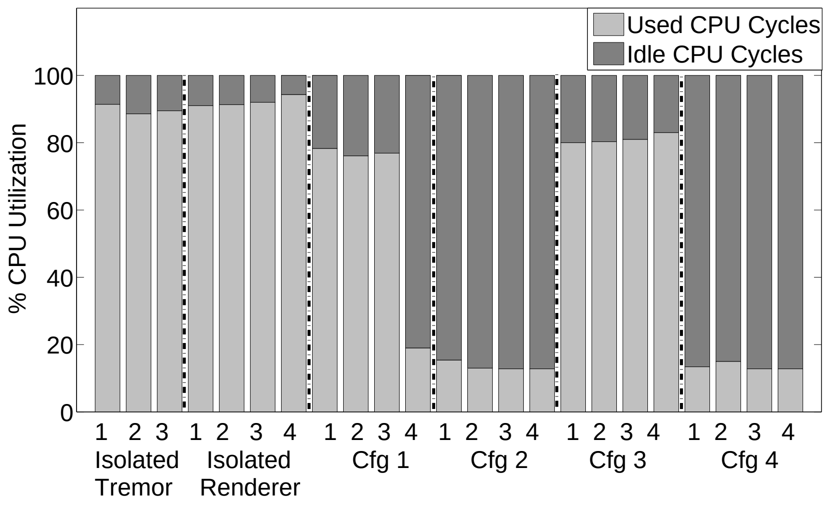


Figure 3.8: CPU utilization of Tremor and ASCII-Art

can be seen, the first configuration (chosen by our monitoring service) performs better for the given workload. It is interesting to note that the performance of the second and fourth configuration, where the coCPUs are evenly distributed between the two computations, results in a poor performance which is actually quite close to the mixed sequential execution scenario (without using any coCPUs at all). This is because the main CPU cannot service any of the two pipelines fast enough, so the coCPUs are underutilized resulting into a degraded performance.

3.6.4 Loading and Transition Delay

To get a feeling of the overhead caused by our explicit loading process (vs having the application code pre-installed on the coCPUs as a part of the FPGA configuration bitstream), we measured the time it took for the application loader to deploy the pipelined version *Tremor* on the first prototype platform, once the FPGA is appropriately configured. The delay recorded for tremor loading was $1190 usecs$ while for ASCII-art renderer it was close to $810 usecs$. As a rule of thumb, on Xilinx Microblaze $100 Mhz$ platform the observed mean overhead was around $100 usecs$ per KB per coCPU for up to five coCPUs. Transition delay to a new coCPU assignment configuration in all cases depended on the amount of KBytes that had to be (re)loaded on the coCPU network and never exceeded 2 msecs which is negligible.

It is also possible to give a lower bound for coCPU network loading, as follows. The loading of each word needs 1 instruction to increment the memory pointer, 1 to store data, 1 to increment the received word counter, and one to check if more data is expected. This adds up to a total of 5 instructions which can be executed in 5 cycles on the Microblaze (taking advantage of the datapath pipeline). A word can be transferred between two CPUs in a single cycle, so $(N - 1)$ cycles are needed for a word to reach the Nth coCPU down the pipeline. Assuming that the binary for the ith coCPU is Bi words long, the total CPU cycles required to load the entire application can be calculated using the formula:

$$Cycles = \sum_{i=1}^{N} (5 + i) * B_i$$

For the case of *Tremor* and 3 CPUs, this adds up to 1000 *usecs*. Notably, this is considerably less than the value reported by the runtime loader. We assume this deviation is due to operating system activity during the loading process as well as the system calls used to get the timestamps and calculate the time difference, which may introduce a significant overhead when measuring small time intervals.

At this point we should clarify that we pre-configure our platform with the maximum number of coCPUs needed in each platform case, with all possible interconnections, and we let the driver ignore the FPGA reconfiguration parameter in the load balancing algorithm. Thus, the measured times include the runtime binary loading to the coCPU network, but not the time needed to reconfigure the FPGA. Despite that the latter can also be done at runtime, this involves writing the bitstream to an external flash which is unacceptably slow and would have distorted the results of the experiments presented here.

3.6.5 Measurements on the heterogeneous Platform

To capture the various aspects of selectively deploying pipeline stages at runtime we have conducted experiments for different scenarios on a heterogeneous platform only for *Tremor*. In all cases, the Tremor application is supplied with the same input file that we used for the previous platform holding 102500 data samples. Time is measured again using the standard Linux *gettimeofday* for response times. Power consumption is measured using a computer controlled digital multimeter.

In a first experiment we measure the time for executing the sequential version of the pipelined application that is produced using the preprocessor and compare it to the original Tremor program. The original needed $637msecs$ while the transformed sequential version completed in $656msecs$, denoting an overhead of 2.9%. Note that the latter also performs an *ioctl* call in each iteration.

In a second experiment we measure the execution time of the pipelined Tremor for all possible configurations (in an idle system) and compare it with the strictly sequential stage execution. Figure 3.9 shows the results along with the theoretically derived values. Note that the theoretical estimates fail to capture the memory allocation and data transfer overheads of the pipelined version, which turn out to be quite significant in practice.

In a third experiment, the pipeline Tremor is executed on an idle system, under the configuration control of our driver. The driver profiles the application pipeline under sequential execution, and decides to keep this configuration. This is indeed the best choice, as it can be inferred from the previous results.

In a fourth experiment, the performance of Tremor is compared in the presence of another (dummy) application which performs a certain number of floating point operations that require $200msec$ on an idle system. The dummy application is started when Tremor has performed half of its iterations (has processed the half input). Figure 3.10 depicts the results, in terms of the response

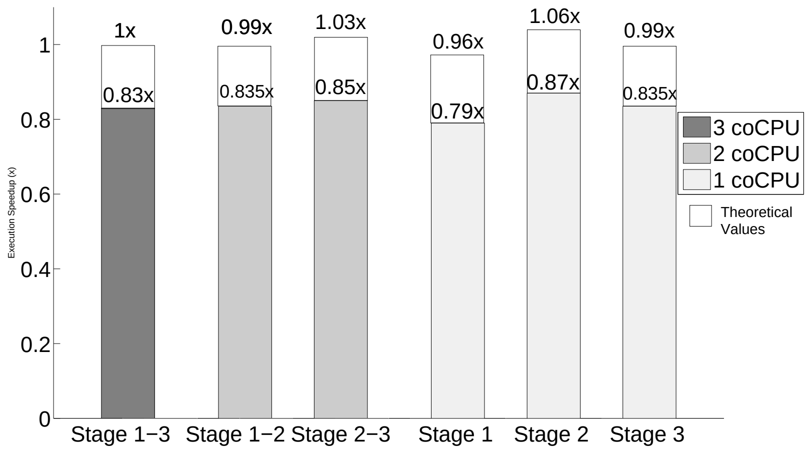


Figure 3.9: Performance of different configurations

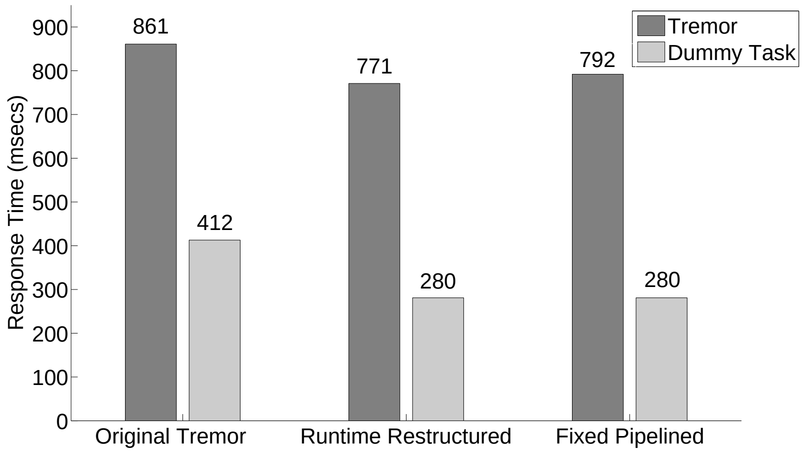


Figure 3.10: Load balancing performance

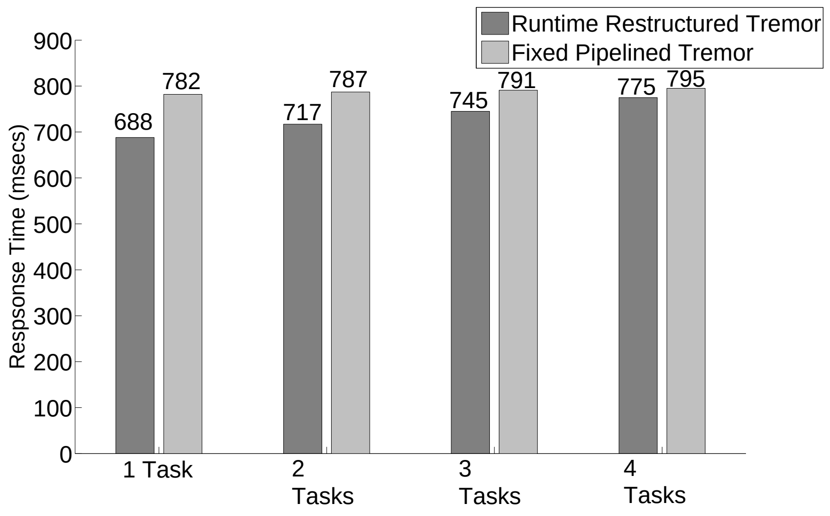


Figure 3.11: Load balancing performance for different number of consecutive task arrivals

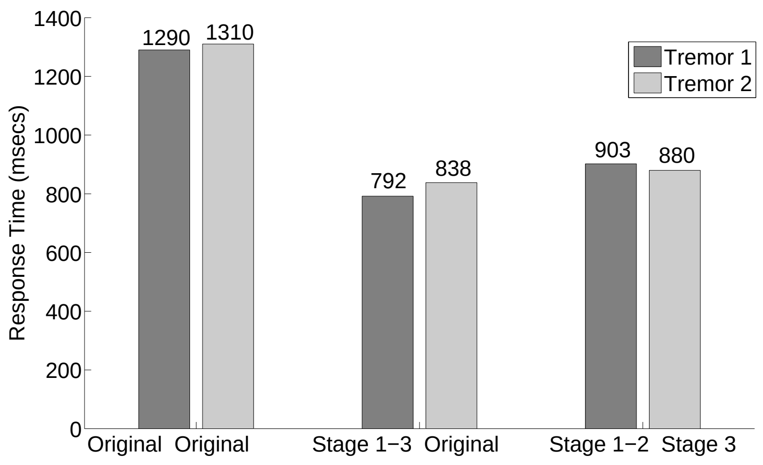


Figure 3.12: Concurrent execution of 2 Tremor instances

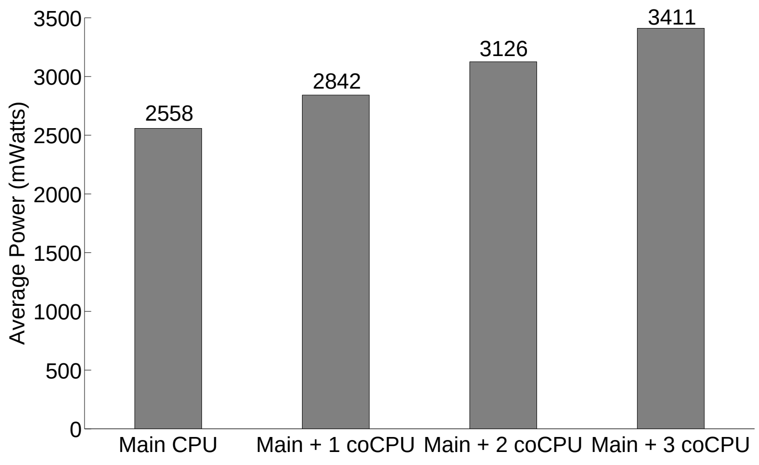


Figure 3.13: Average Power Consumption

times, for the original and pipelined Tremor version. The latter is shown with task restructuring and load balancing enabled as well as for a fixed full-fledged pipeline configuration. In the former case, Tremor starts to execute in sequential mode to take advantage of the main CPU, switches to a full-fledged pipeline mode when the dummy application is introduced, and back again to the sequential execution when the dummy application terminates. As it can be seen, the pipelined versions result in better individual and overall execution times. Moreover, the dynamic version performs better than the fixed pipeline version, even if only marginally in this case, despite the overhead for switching between different task structures and pipeline configurations. To further outline the potential and the benefits of dynamic load balancing, we have developed a shorter version of floating point tasks that needs $50msec$ to complete on an idle system. In the respective execution scenario, one or more of these tasks arrive consequtively during the Tremor execution. The next task always arrives after the previous one has finished its execution and at least $100msec$ have elapsed since then. In figure 3.11 we present the performance results for runtime restructured and fixed pipelined Tremor that execute concurrently with 1,2,3 and 4 tasks arriving consequtively within its execution. Despite the switching overhead, dynamic load balancing optimally exploits the faster system resource, which is the main CPU in our platform, to improve the overall workload completion.

In the last experiment, two pipelined Tremor versions are executed concurrently. The driver, following the heuristic described in Section 3.5.6, chooses the full-fledged pipeline execution with 3 stages executing on coCPUs for one instance and the fully sequential execution of the other. The performance achieved is shown in Figure 3.12, compared to a different deployment scenario where the first two stages of one instance and the third stage of the other are executed on coCPUs. The performance of the concurrent execution of two instances of the original Tremor program is also shown as a reference.

Power consumption is directly related to the number of the deployed coCPUs, as shown in Figure 3.13. Obviously, the ability to deploy coCPUs on demand, only if these can be used to boost application and/or overall system performance, can save a significant amount of power.

3.7 Related Work

An ideal target for our framework would be the Ambric RPPA architecture [7] integrated with a master CPU that can run a full-fledged OS. Currently Ambric uses a structural object programming model. The programmer separates the application into high-level processing objects which can be developed independently and can execute asynchronously with each other, at their own speed, on their own dedicated processor core. Then the programmer defines how the objects should be interconnected to form a computational model close to a Kahn Process Network [1]. The development toolchain provides an emulation environment for debugging and performance estimation, as well as tools to compile objects, configure and load the PPA network at runtime. Our framework extracts programmer identified pipeline flows out of existing sequential applications and each independent stage could be mapped to an Ambric object with proper backend changes. This way our approach could be used to quickly transform existing codebase for the Ambric architecture rather than coding everything from scratch. Moreover, the authors consider Ambric an application specific processor and compare it to DSPs and FPGA accelerator designs. Since Ambric building blocks are processor cores, we believe that the integration of PPA functionality in an OS context is also important. Cell processor [43] could have been another possible target for our framework. But in this case, given that all cores, apart from private local memories, share the main system memory, the code transformation approach can be explored in the context of more appropriate programming models [44] instead of distributed memory pipelines.

Another key issue when designing a multi-core platform is the selection of the application target group; it has been shown that applications perform poorly when executed on platforms customized to speedup a different type of computations [45]. This decision is particularly important for non-uniform memory designs that also use distributed memory, where the processor interconnections can be customized to achieve higher data transfer rates. Note that the problem of "overspecialization" is the main reason why the processor pipelines/arrays proposed in the 70s and 80s have not been considered for a wider range of computations. Reconfigurable hardware creates new opportunities to that end, namely, by deploying on demand application-specific accelerators at runtime [40]. Customized multi-core configurations, in the form of a coprocessor graph, have been already used in digital image processing [46], software radio [47] and network applications [48]. However, in all these cases, both the platform design and the program development were done by hand and were tuned to work in conjunction with each other.

Writing parallel applications for a multi-core target platform is a difficult task. There is substantial work on compilers that support imperative programming languages such as C, trying to exploit instruction-level parallelism and/or to identify independent threads of an execution at a higher level, without application-level support [49, 50]. In general, however, little parallelism may be achieved this way. Programming models and languages for capturing and exploiting parallelism at the application level have also been proposed. For instance, StreamIt [51] introduces a high-level model which enables the compiler to automate tasks such as partitioning, load balancing, layout and memory management. This is currently being used to program the RAW architecture [52]. On the other hand, message-passing [53] has become widely accepted as a portable style of parallel programming, but inherits costly semantics related to message queuing and selection options which are often not required. POSIX threads [54] can also be used to explicitly capture parallelism at the level of the application, but lack support for data parallelism and require programming at a lower abstraction level than most developers would prefer. Despite their drawbacks, these approaches have been used on both shared memory and distributed memory architectures [55, 56].

Taking a different approach, *OpenMP* [13] is a specification for a set of compiler directives, library routines, and environment variables which can be used to specify the desired level of parallelism. Although *OpenMP* is typically used in shared memory systems, some extensions have been proposed for distributed memory systems as well [57]. Inserting annotations to build parallel versions of a sequential program has the advantage of code reuse and simplicity; in many cases the desired transformation requires minimal changes to the source code. The same annotations and library calls can be substituted by the *OpenMP* preprocessor with radically different versions or codeblocks aimed for different platforms. Moreover, the programmer can specify and experiment, e.g., via profiling, with different partitions in a straightforward way. The main drawback is that the programmer may have to take low-level parallelization decisions, which can be quite awkward and/or may hinder portability.

In [58] code annotations are proposed to achieve coarse grained pipelined parallelism. A dynamic analysis tool is used to help the programmer balance pipeline stage workload, and the underlying runtime support is used to fork the pipeline stages as different private memory processes that communicate via *Unix* pipes. Pipeline parallelism is also exploited in [38] using the techniques of Decoupled Software Pipelining [59], also employing thread-level speculation to opportunistically execute multiple loop iterations in parallel and extract this way parallelism from previously unparallelizable loops. A special annotation is used to indicate commutative functions/actions and a hardware-based mechanism called versioned memory, similar to our software context-management queue, is used to enable shared memory thread-based execution support instead of processes. Both approaches, assume an isolated environment that runs on a shared memory homogeneous multiprocessor system where all the resources serve one pipelined application at a time so they are not considering any dynamic load balancing attributes and mechanisms.

On the other hand, application-level dynamic load balancing approaches have been proposed to improve execution of parallel applications on heterogeneous large scale SMP clusters. In [60] OpenMP extensions are proposed that introduce dynamic load balancing attributes. These extensions are appropriately placed inside loops that have been partitioned to exploit data parallelism. Each computation iteration that is assigned to a different thread is profiled by appropriate application-level support that is added by the compiler. This way every next iteration, the data load is distributed according to the previous profiling feedback. Apart from the fact that this approach can be only used on applications that exploit data parallelism and requires the programmer to explicitly place load balancing annotations, it also does not use a central authority to decide application partitioning. As we have demonstrated in the fourth experiment of the section 3.6, this is of particular importance when two or more transformed applications are executed concurrently.

In the same spirit several dynamic task partitioning methods have been proposed [61, 62] that deal with the partitioning of unstructured mesh problems like Computational Fluid Dynamics. In [61] the sub tasks of a computation are developed using an appropriate programming model and a software framework called Data Movement and Control Substrate(DCMS)[63], which features an extension called Mobile Object Layer(MOL)[64] that enables transparent task migration between processors at runtime. The authors have extended the MOL concept and have added appropriate load balancing routines that communicate with respective runtime support. The system features a plugin interface so the programmers may define their load balancing algorithms depending on their target cluster platform. While the main principles of this design are close to ours, tasks are just migrated and not restructured and, moreover, explicit load balance related calls must be added and invoked respectively at the application level.

Task dynamic partitioning support that enables load balancing at runtime has been proven to achieve good performance on Multiple Instruction Multiple Data (MIMD) targets. In [65] the authors have implemented in the Mul-T[66] runtime system support that enables parallel tasks to execute either independently or in the context of a few system tasks based on the resource availability. As in our approach, the programmer is expected to identify and expose parallelism without worrying about the granularity while the system limits parallelism to meet platform capabilities. Their implementation futures Lisp constructs that must be used to define independent execution entities and is tightly integrated with Mul-T runtime design and their target platforms ALEWIFE and Encore Multimax. This approach is primarily targetted at parallel algorithms that are naturally expressed at a fine level of granularity which is typical workload for MIMD architectures and does not consider pipeline applications. Moreover, the partitioning decisions are application driven and they no central coordination authority is employed to deal with the case of applications that execute concurrently.

In [67] a different approach is proposed to improve load balance performance. The authors identify as a basic parallel application performance problem the assumption that all platform processors are fully available to perform a given computation. To solve this, they present a user-level task scheduler that schedules tasks onto a fixed collection of processes which the operating system kernel further schedules onto a fixed collection of processors. The proposed scheduler implements the work stealing algorithm[68] where each process is assigned a pool of independent tasks to execute one-by-one. When a certain task pool is depleted, the process steals threads from the neighbour pool.

Programming models and languages for capturing and exploiting parallelism at the application level have also been proposed. For instance, StreamIt [51] introduces a high-level model which enables the compiler to automate tasks such as partitioning, static load balancing, layout and memory management. This is currently being used to program the RAW architecture [52]. In this case, to our knowledge, dynamic load balancing has not yet been considered.

With the advent of softcore instruction-set processors that can be deployed on reconfigurable hardware new multicore processing opportunities arise by enabling the building on demand as well as the runtime deployment of application-specific cpu types and platform configurations[40]. Such customized multi-core configurations, in the form of a coprocessor graph, have been already used in digital image processing [46], software radio [47] and network applications [48]. However, in all these cases, both the platform design and the program development were done by hand.

3.8 Summary

We have presented load balancing support as part of a framework we have developed to transform sequential applications for pipelined execution. Our primary motivation for this work is to explore the potential of parallel processing on RPPA configurations that can be used in the context of multitasking. To that end, we preferred pipelined parallelism because all computations can be transformed, to a certain extend, for pipelined execution and moreover, for obvious reasons, it is a more suitable concept for the use of heterogeneous cores on-chip.

Embedded systems require flexibility at all design levels, and therefore reconfigurable hardware designs combined with a general purpose runtime are good candidates even for commercial solutions[20]. Our vision is to have such a system where all userland applications have the ability to deploy stages of execution to dedicated and appropriately customized processor cores and, to that end, support for dynamic load balancing and corresponding selective stage scheduling is of major importance because it enables effective resource usage.

Our approach is in line with the principle of letting the programmer specify the maximum parallelism at the application-level, while making system responsible for deploying the code in the best possible manner, depending on the underlying platform characteristics and the current execution context. To that end, we have strived for a generic design that can be ported/applied to radically different platforms using appropriate backends. Nevertheless, the dynamic load balancing support that we have presented here can only be realized via offloading of stages onto the main CPU. Groups of sequential independent stages cannot be assigned to the same coCPU (only one is allowed) and cannot be migrated in this manner at runtime. This inefficiency is because the source-level restructuring is very complex cannot be efficiently extended to support more radical dynamic load balancing scenarios. In the next chapter we present a new model that has been designed to inherently support dynamic load balancing.

Chapter 4

PipeIt Framework for RPPAs

This chapter presents the *PipeIt* framework for developing pipelined applications targeted at tightly-coupled PR processor arrays on-chip. Contrary to our previous openMP-style approach, *PipeIt* programming model design is targeted at the development of highly optimized applications for RP processor arrays and as a result has a steeper learning curve for developers with experience on the sequential programming style. The framework includes a component programming and wiring model, a runtime environment, and a corresponding toolchain. It enables the programmer to develop applications in a high-level manner, structuring the code at the finest possible/meaningful level of granularity, without caring about how this will actually be deployed and executed. The runtime environment dynamic loading mechanisms can take advantage of the available *PipeIt* stub to evenly distribute the pipeline stages among available resources and dynamically change this arrangement on demand when another *PipeIt* application needs to be executed concurrently. As a proof of concept, we discuss the development of a fine-grained pipeline for the computation of Secure Hash Algorithm (SHA1), its integration with *Crypto* library, and execution combinations of this code on a Suzaku-based RPPA platform in the multi-tasking context of a uClinux environment.

4.1 Introduction

The promising performance gains combined with less physical limitations and better financial scalability, have motivated researchers to improve all aspects of multicore computing. While high-end computing remains the most obvious area of application, multicores are rapidly becoming mainstream in conventional user desktops as well, even in embedded system solutions [7].

There is a variety of approaches at the architecture level, ranging from loosely-coupled processors interconnected via ethernet on different nodes, on the same board or even the same chip, to tightly-coupled processors on a chip that are typically assigned with dedicated tasks of a larger computation and are connected together via a bus or high end dedicated links without any arbitration [7]. Platform differences in conjunction with the wide polymorphism in terms of applications that seek for parallelization opportunities also lead to a variety of partitioning and communication schemes. In turn, these can be supported by different tools. Of course, the type of computation at hand may naturally favor a certain scheme, making it more appropriate for extracting the most amount of parallelism. Hybrid system configurations, for example a cluster of Cell Blades [43] where each node features several Cell processors which in turn feature several cores on chip, introduce additional challenges. The effective partitioning and deployment of an application on such a target platform requires the synergistic use of more than one programming approaches.

Another important aspect is whether a multicore system is used in a dedicated fashion, or in the context of an open computing environment. In the former case, the optimal partitioning of the computation that will lead to the best possible performance (given the available system resources) can be decided at the design phase. On the contrary, in the latter case, new tasks may appear at any point in time during execution and the available resources must be used opportunistically to boost overall performance, necessitating their redistribution each time a new task arrives.

In this Chapter we present the *PipeIt* framework which provides support for building and deploying application-specific pipelines on tightly-coupled distributed memory PR processor arrays[7] in the context of a general-purpose computing environment. *PipeIt* includes a component and wiring model, a runtime backend that is appropriately customized for and integrated with the execution environment, the platform processors, and a corresponding front-end compiler that generates the ultimate source code and build scripts, so that a regular toolchain can then be used to produce proper executables. Finally, a custom loader that is aware of the target platform available resources is used to deploy *PipeIt* application stages at runtime. This initial arrangement can be changed at any point in time. *PipeIt* is designed for open, best-effort computing systems, where the workload is not a priori known.

The main additional contributions of this programming model are: i) the modular application design approach that enables the reuse of basic/common pipeline structures; ii) support for seamless pipeline execution, while the underlying runtime may use a provided stub to dynamically reassign stages to available cores on demand and thus rearrange the dataflow inside the PPA according to the current execution context with minor overhead; iii) the ability to invoke a pipelined computation via a simple library call from within a conventional application; iv) a prototype implementation of all the development tools together with an emulation environment for debugging and accessing expected performance of the pipelined computation; and v) a proof-of-concept implementation of a well-known application on an FPGA-based PPA prototype.

4.2 Application Domain and Target Platforms

With *PipeIt* we wish to support the development of pipelined computations that perform CPU-intensive data transformations and operate on data block streams. Indicative cases are block cipher algorithms for encryption or authentication, data (de)compression algorithms that are widely used in data storage, and encoding algorithms for video or audio.

The target platform is distributed memory, tightly-coupled RPPA systems on-chip [7] aimed for general purpose everyday use, such as desktops or embedded devices. These platforms typically feature reconfigurable, ultra fast and dedicated interconnections that introduce a small overhead for data block transfer. However, the local memories of processing elements can host only a few Mbytes of data. This configuration does not allow the PPA cores to be used in the typical time-shared manner like most shared memory multicore platforms. Therefore, PPAs are currently used as dedicated coprocessors that may carry out one large computation at a time that is typically divided in an a priori known number of tasks which are statically assigned to an array processor. We believe that RPPAs with appropriate support may also be used as a main processing element in a dynamic environment to carry out general purpose workload.

We assume that either an external processor or a special processor on the RPPA plays the role of the platform *master*. This distinct processor is interfaced to all platform peripherals, runs a proper OS/runtime, and configures the (rest of the) arrays to setup application pipelines as desired. It may also be responsible for pushing data into and pulling data out of the pipeline during execution, in case this cannot be done by any ordinary array processor, e.g., if array processors have no access to the main memory and I/O peripherals.

In the general case, several pipelined applications may execute concurrently to each other. Pipelined applications may also execute concurrently to other conventional (sequential) applications, which run on the master processor. Notably, some sequential applications may contain parts that are implemented as pipelined computations and which can execute on the processor array, subject to resource availability.

4.3 The *PipeIt* Framework

To enable structured development of pipelined applications, *PipeIt* adopts a component model. Each component represents a pipeline stage that ideally should be executed using a separate processor. Components have a fixed number of input and output ports. They can be wired together in the form of a directed graph, linking together output ports with input ports in a point-to-point fashion according to the desired data flow. The wiring of components is practically orthogonal to their implementation, and it is specified in a separate so-called configuration file. Basic checking is done to make sure that interconnected ports provide/expect data objects of the same size (in principle it is possible to add higher-level type checking too, but we have not done this).

During execution, each component blocks until data is available at its input, processes data, and writes data to its output, in an endless loop. The only component that has no input ports, i.e., does not wait for data to arrive from another component, is the pipeline entry point, called *root*. The *root* component typically reads data from an external source, such a file, special memory location or a special device. Similarly, the only component that has no output ports, i.e., does not send data to another component, is the pipeline exit point, called *sink*, which typically writes data to an external destination. The root and sink both run on the master processor (using two different threads), enabling the seamless integration of the pipelined computation with the rest of the application and

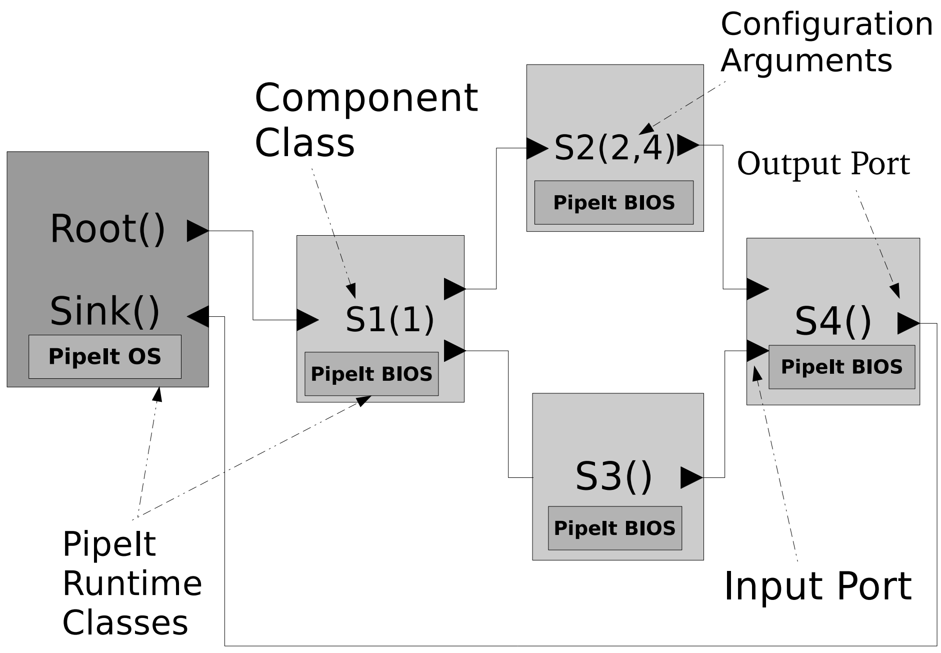


Figure 4.1: An indicative *PipeIt* configuration

system.

Figure 4.1 illustrates a sample *PipeIt* pipeline. Notably, it is possible to have branches in order to introduce data parallelism inside the pipeline or to support different, mutually exclusive, processing paths.

The performance of a pipelined computation is obviously dominated by its slowest stage. *PipeIt* provides an emulation environment for executing pipelined computations to get feedback about the amount of communication and processing performed by each component. This information can be used to restructure the pipeline (components and wiring) in order to achieve a better balance. The developer is free to introduce *as many* components are needed to strike a good compromise between the desired level of granularity and an attractive computation-to-communication ratio. Obviously, the bigger the pipeline length, the greater the potential for accelerating the computation (provided that all components perform a comparable amount of processing).

At design time, one naturally assumes an unloaded system. In fact, it is not even necessary to respect the resource limitations of the target platform, even if these are known; still, having too many components compared to what the system can actually support is non optimal because it introduces some extra communication overhead. Of course, at the time of execution, there may not be as many processors available, either because the system does not have them in the first place or because some processors are already being used for other applications (note that the latter is impossible to know in advance). To deal with this problem, *PipeIt* provides support so that the pipeline can be dynamically balanced, arranging for some components to be executed in a co-located fashion on the same processors, trying to evenly distribute load among the available PPA cores. When resources are freed, the reverse restructuring may occur, i.e., co-located components can be distributed for execution on separate processors, again, provided this will result in a performance boost. Obviously, the dynamic load balancing policies that can be used may vary according to usage scenarios. *PipeIt* provides support to the underlying runtime so that the PPA cores may be redistributed among pipeline stages and therefore makes seemless pipeline runtime load balancing possible. Exploring dynamic load balancing policies is beyond the scope of this dissertation.

The following subsections discuss the most important aspects of the *PipeIt* framework in more detail.

4.3.1 Component and communication model

A *PipeIt* component is coded as a C++ object, in a separate file with the same name. Each component class must be defined as a subclass of a *runtime type* class, which features two virtual functions, *config* and *exec*, that must be overloaded.

The *config* function is called only once, before the actual execution commences, and must be used to declare the ports of the component and initialize its internal state. Different configuration strings can be passed to each component, making it possible to implement very flexible initialization schemes, and allowing for component classes to be easily reused in the same or different *PipeIt* applications.

The *exec* function must contain the component's data transfer and stage processing code. It is invoked repeatedly during execution from within the *PipeIt* runtime, in an endless loop.

Data transfer is performed using the *input* and *output* functions. These are inherited from the base runtime type class, and are transparently mapped to the communication primitives of the respective processor runtime environment. Ports are addressed using a simple numbering scheme which is mapped by the *PipeIt* framework to the target-specific addresses. In a nutshell, components receive and send data using the abstract *PipeIt* primitives and port ids without caring about the underlying implementation or the physical location of neighbor components.

Memory allocation of data buffers must be done using the *pipeIt_malloc* function provided by the runtime. This is necessary because *PipeIt* needs to control data access and transfers in order to perform component migration safely during pipeline restructuring. For the same reason, static declarations of data transfer blocks are not allowed.

As an example, we give the code of a simple *PipeIt* component that receives an integer from its (single) input port, increments it, and forwards it to its (single) output port:

```
int * data;

class IncInt : public PipeItOs
{
  public:

     IncInt() {;}

    /* overloaded functions follow */
    void config(int argc, char *argv[]) {
      data = pipeit_malloc (sizeof(int));
      pipeit_add_input (0, sizeof(int));
      pipeit_add_output (0, sizeof(int));
    }

    void *exec(void *d, int size) {
      pipeit_input (0, &data, sizeof(int));
      *data = *data + 1;
      pipeit_output (0, &data, sizeof(int));
    }
};
```

Data is passed from one component to another by writing it into the proper output port and reading it from the proper input port, respectively. However, some variables may need to be accessed *only* between the root and sink components, hence need not travel through the entire pipeline. A separate mechanism, called the *context queue*, is employed to keep track of these values in synch with the pipeline. Before writing data into its output, the root adds a context entry with the proper values, and, conversely, the sink removes the next context entry before attempting to read data from its input. In our current implementation, where the root and sink both execute on the master processor via two different threads, we use a shared FIFO queue. This is similar in concept with the so-called versioned memory introduced for shared memory multiprocessors [38]. Obviously, a different implementation would be needed if the root and sink were to execute on different processors with separate memories.

4.3.2 Runtime classes

PipeIt introduces runtime support mainly for two reasons: i) to confine the programmer during the development of a component to the execution context and available resources of the target environment; and ii) to be able to perform dynamic pipeline restructuring at runtime in a seamless way.

There are two radically different execution environments, the master processor which runs a proper operating system, and ordinary RPPA processors running a small basic I/O system (BIOS) that is a custom implementation. In both cases *PipeIt* adds a thin layer providing a set of generic data transfer primitives, optimized for the respective environment. This allows components to be implemented in an abstract fashion without really caring about their mode of execution.

Similarly, there are 3 different *runtime type* classes, i) *PipeItOS*, ii) *PipeItBIOS* and iii) *PipeItOSLib*, which can be used to develop components. Each reflects a different flavor of the generic *PipeIt* runtime support, discussed in more detail in the sequel.

The PipeItOS class is used for components that will execute on the master processor, having access to the full functionality of a proper OS. This runtime class is used for the root and sink components, which may contain system calls and access peripherals. The code generated in this case has the form of an autonomous executable for the master processor linked against the OS runtime, using a separate *POSIX* thread for running each component.

The PipeItBIOS class is aimed at components that should run on ordinary array cores on top of the *PipeIt* BIOS. In this context the programmer may only perform CPU intensive computations, read data from input ports and write data to output ports. Erroneous attempts to use a (non-existing) runtime feature will cause the compilation of the component to fail. The default for such components is for them to execute on a dedicated array core. However, as a result of pipeline restructuring, several such components can be placed on the same array core or even on the master processor.

Finally, the PipeItOSLib class has a similar functionality to PipeItOS, but it does not result in the generation of an autonomous executable. Instead, it produces code that enables the pipelined computation to be invoked from within an external application context, much like a library. In this case, the root and sink components must establish an appropriate communication with the application, based on the arguments of the *config* and/or *exec* functions. In principle, any mechanism and protocol can be used for this purpose (*PipeIt* makes no assumptions about this). As a single convention, the application must invoke a routine following a simple naming convention, which initializes and spawns the *POSIX* threads for running the root and sink, before attempting to communicate with them.

4.3.3 Configuration language

The wiring of each *PipeIt* computation is specified in a separate configuration file, using a simple syntax. Configurations are expressed in terms of 3 main elements: component declarations, port connections and composites.

Components are declared by the class names used in the respective implementation files. A configuration string may optionally be used to convey initialization information to the instance being declared. Configuration strings are not interpreted by the *PipeIt* framework; they are passed "as is" to the respective components, via a call to the *config* function. If a component is referenced only once, there is no need to explicitly declare it, and the class name can be used instead.

Input ports are denoted by placing the respective number in brackets on the left hand side of a component name, while output port numbers are placed on the right. Each connection is denoted by a right arrow, starting from an input port and pointing to an output port.

To enhance the structure of complex computations, and to enable the reuse of common substructures (in the same of even different configuration files), configurations have to be grouped into so-called composites. A composite can have an arbitrarily complex structure. To the outside, it merely exports its input and output ports, i.e., appears just like an ordinary component. There is currently no support for nested composition.

Composites have a so-called execution type, for which there are two options. The *PipeItMaster* type is used for composites that will run on the master processor; their components must extend the PipeItOS or PipeItOSLib class. The *PipeItArray* type is used for composites that should run on ordinary array cores, and all of their components must extend the *PipeItBIOS* class. A configuration file must have exactly one *PipeItMaster* composite, which contains the root and sink of the pipeline. It can have one or more *PipeItArray* composites, depending on the complexity of the computation. Below we present a formal description of the *PipeIt* language:

```
/*Component Declarations*/
name :: classname(config);
/*multiple*/
name1, name2, ..., nameN :: classname(config);
/*or standalone (demonstrated at Connections)*/
classname(config)

/*Connections*/
classname1 [portID] -> [portID] classname2;

/*Composites*/
PipeItArray name1 {
    input[portID] -> ...
       Component Connections ...
    -> [portID]output;
}

PipeItMaster master {
    output[portID]-> ...
       PipeItArray Composites ...
    -> [portID]output;
}
```

and small example that demonstrates the regular use:

```
PipeItArray IncIntTwice {
  inc :: IncInt();
```

```
   input[0] -> [0]inc;
   inc[0] -> [0]IncInt()[0] -> output[0];
};

PipeItMaster MyApp {
   r :: MyRoot("42");
   s :: MySink();
   r[0] -> output[0];
   input[0] -> [0]s;
};

MyApp[0] -> [0]IncIntTwice;
IncIntTwice[0] -> [0]MyApp;
```

This application is based on two composites to increment an integer value twice. The *IncIntTwice* composite of type PipeItArray uses two appropriately connected instances of the *IncInt* class (defined previously) to do the job. Once instance is declared explicitly while the other is declared implicitly, via the class name. The *MyApp* composite of type PipeItMaster contains a *MyRoot* and a *MySink* component (code not shown here). Note that local (intra composite) connections are declared within the respective scope, while global (inter composite) connections are defined at the end of the configuration file. The keywords *input* and *output* are used to refer to the input and respectively output ports of the composite. In this case, the computation takes its input via the configuration string passed to the *MyRoot* component, hence the *MySink* component will receive the value 44.

4.3.4 Dynamic load balancing support

Ideally, the developer specifies the pipeline structure based on the assumption that all the components (stages) of the pipeline will be executed on a dedicated array core. Unfortunately, this is not guaranteed in an open, general purpose computing system. On the one hand, the actual target platform may not have as many array cores. On the other hand, some of the array cores may be used by other programs when the application starts running. It is also impossible to predict what tasks will arrive or finish during its execution. As a consequence it may not be feasible to keep the full-fledged pipeline structure, as this was originally defined by the programmer.

To enable the execution of pipelines, as efficiently as possible, *PipeIt* comes with built-in support that enables dynamic load balancing. Specifically, the *PipeIt* runtime can seamlessly place components on the same or different processors to evenly distribute the processing load on the available resources. In order to keep things simple and efficient, pipeline restructuring obeys the following rule: if two components are to be placed on the same processor, then every component between them must also be placed on that processor. This is because components can be only executed sequentially, not in a time-shared manner. Notably, the most extreme option is to place all components on the master processor. To illustrate this, Figure 4.2 shows all allowed configurations for a pipeline with 4 components, including the root and sink.

Appropriate *PipeIt* support has been developed to perform this restructuring in a transparent fashion and to achieve an efficient execution of co-located components. The ultimate partitioning decision has to be made by the underlying runtime. On each processor, *PipeIt* uses a simple scheduler to execute all co-located components sequentially. The components to be executed locally are specified using a so-called *ComponentExecutionMap*. The map of each processor is set by special profiling code that runs on the master processor (discussed in the sequel) and is sent to the respective processor using a simple dissemination protocol.

No actual data transfer is performed between co-located components. Instead, both ends of each local link share a data buffer which is accessed from within the respective I/O calls (no synchronization is needed since execution is sequential). The I/O behavior of each component is controlled transparently via a so-called *IOMap*, which indicates whether the *input* and *output* calls should perform a remote or local/virtual data transfer. The same execution and data passing model is also supported in the context of the root and sink threads, allowing for components to be placed on the master processor and for them to be executed in the context of either thread, as needed. Moreover, in the case where it is decided that all components have to execute on the master processor, an optimized version of the sequential execution of the computation may be explicitly coded by the developer. This way the overhead of the indirect *PipeIt* scheduler invocations which is increased along with the finer component granularity will be avoided for the sequential execution.

The initial structure of the pipeline is established as follows. When the computation is first

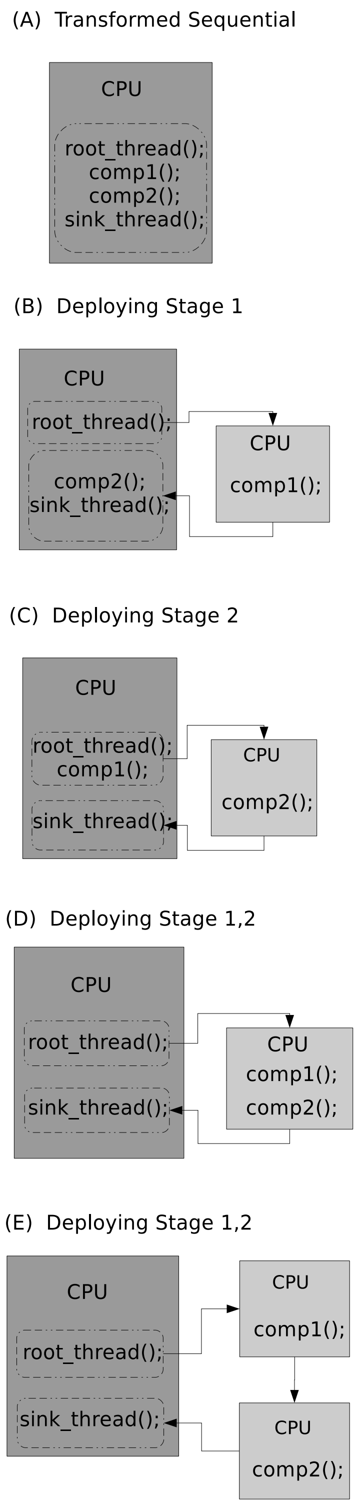


Figure 4.2: Configurations for a pipeline with 4 components

deployed, *PipeIt* runs the pipeline for a number of iterations (defined by the user during compilation) sequentially on the master processor. During this initial execution phase, an appropriately instrumented version of the *exec* call is employed, through which *PipeIt* collects information that can be used to estimate the processing overhead of each component. Next, the optimal component placement scheme that will be used given the available number of processors that is provided by the runtime is decided. Then, the required processors are allocated and the application code along with the corresponding *ComponentExecutionMap* and component *IOMaps* are loaded, and the pipeline is ready to accept data.

The pipeline can be restructured at any point in time during execution. This makes it possible to adapt to changing workload conditions, exploiting processors that become available or releasing processors for the benefit of other applications. Moreover, if the master processor gets loaded (e.g., with other conventional applications) and some pipeline components are executing on it, these can be offloaded to array processors if available. In order to guide load balancing, a corresponding system service must be inquired periodically to provide the most appropriate pipeline structure. Since frequent monitoring introduces considerable overhead, the rate can be explicitly enabled by the programmer when compiling a *PipeIt* application. If the system decides that a new structure is beneficial, *PipeIt* updates the *ComponentExecutionMap* and component *IOMaps*, pushes this information downstream (instead of actual data), and proceeds with the normal pipeline execution.

Figure 4.3 shows two indicative configurations for an application with 5 components, together with the corresponding data transfer and execution mappings. In first case (A) all components execute sequentially on the master processor, and communication is done using shared buffers. In the second case (B) the third component is set for execution on a array processor, and the output mapping of the second component as well as the input mapping of the fourth component are set to invoke the appropriate communication primitives to send/receive data between the master processor and the array processor. All other communication remains local, within the context of the root and sink component, respectively. In the same spirit, (C) depicts the deployment of second and third component on distinct array processors. The mappings and memory use are accordingly adjusted.

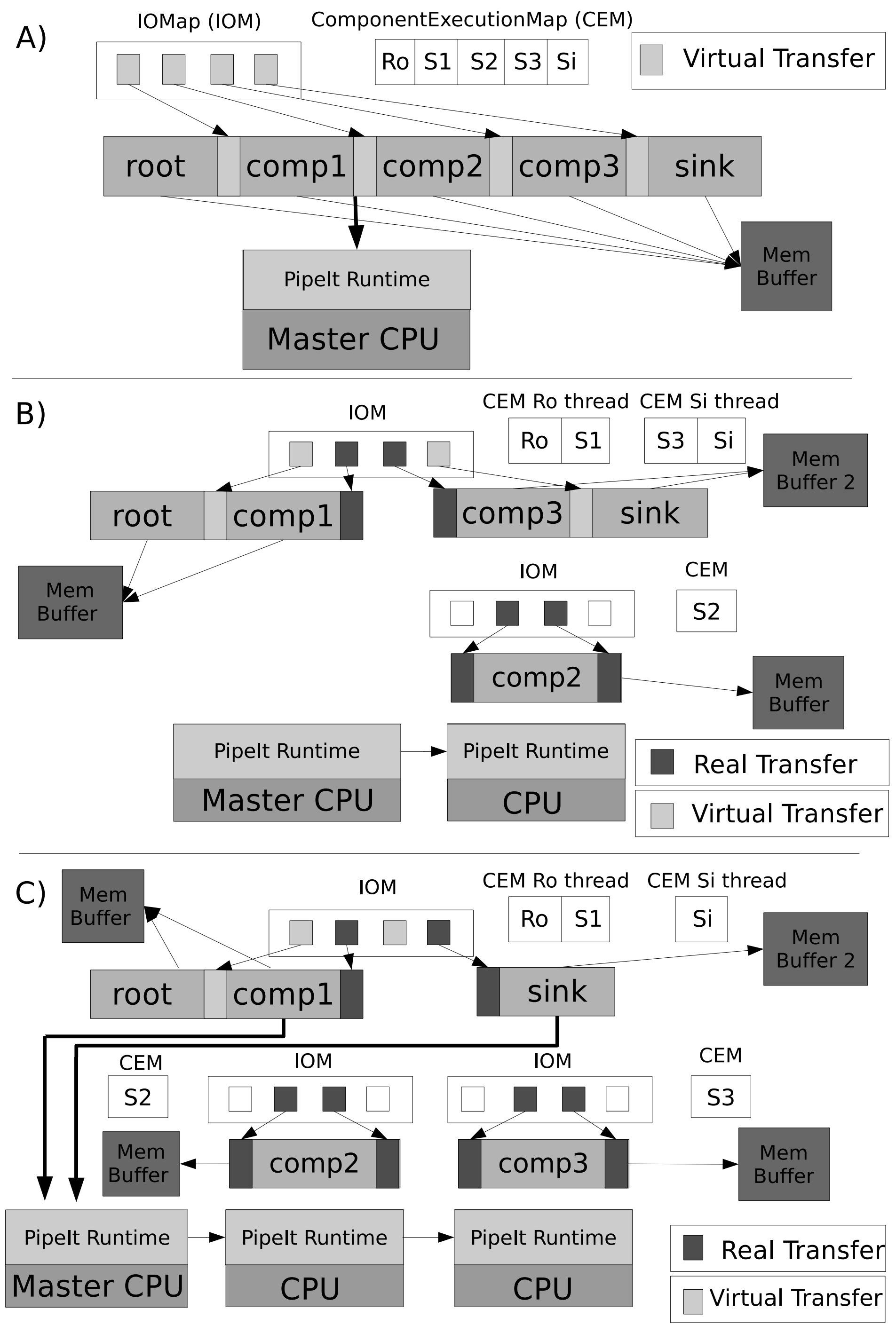


Figure 4.3: Structures for supporting pipeline restructuring

4.3.5 Application development and tools

The developer must first provide at least the skeleton for each component (a separate file for each component class) that will be employed, and then write the application configuration file. The *PipeIt* compiler parses the file, creates the appropriate data structures used to configure all aspects of the pipeline structure, and generates corresponding flavours of *pipeit.h* files, to be included by convention in each component implementation. These header files contain static declarations of various required variables, including the transfer bitmaps and profiling structures, as well as support structures for the mapping of the port numbers to the actual platform-specific addressing primitives that will be used by each component. If the target platform features hardware-level reconfigurable interconnections between processors, the compiler must be changed to produce the necessary configuration support.

The next step is to create *makefiles* that will be used to invoke the regular C++ toolchain for the target platforms as well as for the *PipeIt* emulation environment (see below). As a final step, the executables are placed in different folders depending on the platform target, and the scripts for deploying the application are generated. From that point onwards, the regular *gnumake* environment will take over.

The default configuration produced by the toolchain is to produce code for all components to execute on the master processor environment, and for all components except the root and sink to execute on an ordinary array processor environment. The compiler also accepts a hint in terms of meaningful component co-location for the case where there are not enough array processors or array local memory is not enough to host copies of all components; in the latter case, different executables will be generated for different sets of array processors.

The *PipeIt* toolchain can be used to generate executables for emulated execution on a Linux host. This is achieved by switching the backends of the runtime classes so that they can execute on a usual Linux runtime environment. In essence, the master and array processors are emulated using independent processes while component interconnections are emulated via unix named pipes. When the application is started in emulation mode, the user must specify the desired number of processors to use (in the extreme case, as many as components).

Running an application in emulation mode considerably simplifies debugging. Moreover, it enables the use of sophisticated profiling tools like *gprof* to decide a good partitioning of the computation which will also take advantage of the dynamic load balancing support. Another major motivation for using the emulation mode is to assess the expected performance on the target platform for various configurations. Of course, if the emulation host has a radically different architecture from the target platform, it might not be possible to make an accurate estimation, but the profiling results may be further evaluated if the developer knows the performance difference factor between emulation and target platforms. Of particular importance is also the estimation of the computation-to-communication ratio, for both local/virtual and real data transfers, which becomes increasingly relevant as the pipeline granularity (and length) increases.

4.4 Proof of Concept Prototype and Applications

As a proof of concept, we use a custom RPPA system, implemented on a FPGA as a system-on-chip, for which we have built appropriate backend support into the *PipeIt* framework. In this section, we discuss our prototype as well as indicative experiments that have been performed using pipelined versions of well-known applications.

4.4.1 The platform

Our prototype PPA platform is implemented, again, on the Atmark Techno Suzaku [20], which features a Xilinx Spartan 3 FPGA along with off-chip 16MB RAM, 8MB flash, a MAC core and a configuration controller. For all processors (master and array cores) we use the *Xilinx Microblaze* soft processor, a classic 32-bit RISC architecture, with 32 general purpose registers and an orthogonal instruction set, having a 3-stage instruction pipeline with delayed branch capability for improved instruction throughput. The platform was developed using the Xilinx Platform Studio toolchain version 6.3.

The entire system is implemented on the FPGA. The master processor is interfaced to all platform peripherals and is responsible for running a customized version of the *uClinux* embedded operating system [39], achieving 25.29 *BogoMIPS*. The array processors have access only to their local memory and are connected to each other and the master processor using FSL links. Using special support which we have developed in previous work [40], it is possible to dynamically reconfigure the system to instantiate different configurations (in terms of numbers of processors or topologies). In our experiments, we use a pre-installed configuration with 5 processors (1 master and 4 array cores) with the appropriate interconnections, which takes up all the resources of the FPGA.

As we have outlined in Section 4.3.4, the *PipeIt* runtime needs to know the available processors in order to decide about the component placement scheme. For this purpose we have developed a simple runtime service that keeps track of the available PPA cores as well as the utilization of the master processor, and provides this information to the *PipeIt* runtime on demand via the *proc* filesystem. The service does not employ any sophisticated load balancing policy because we don't intend to investigate load balancing policies in this work.

4.4.2 Applications

As a proof of concept application, we have implemented a *PipeIt* version of the Secure Hash Algorithm. We also reused this implementation, in library mode, to develop an application that performs the HMAC authentication. In turn, we have integrated this functionality in the *OpenSSL* library and deployed it on our target.

Looking at the sequential code of SHA1 it is straightforward to come up with an appropriate pipeline structure. The code invokes sequentially 80 times 4 different types of functions that perform the same amount of computation on the given data block. Hence, the obvious pipeline structure comprises 6 components, namely *root*, one component for each function type, and *sink*. Below, we list a simplified version of the source code of the component that is responsible for performing the processing of the first function:

```
class R0: public PipeItBIOS
{
  public:
    R0(){;} ~R0(){;}

    struct Data  * d;

    void config(char * args) {
        d=pipeit_malloc (sizeof(struct Data));
        pipeit_add_input (0, sizeof(struct Data));
        pipeit_add_output (0, sizeof(struct Data));
        parse_args (args);
    }

    void exec(void)  {
        pipeit_input (0, (void *)d, sizeof(struct Data));
        R0Calc (d+arg1, d+arg2, d+arg3, d+arg4, d+arg5, offset);
        pipeit_output (0, (void *)d,sizeof(struct Data));
    }

};
```

The configuration file, for linking together 80 such components as well as the root and sink, properly configured and in the desired order, is as follows (in this case component declarations are implicit, instead class names are used directly for brevity; this is in fact valid configuration syntax which is quite useful if a component is referenced at only one location):

```
PipeItArray SHA1 {

  input[0]
  -> [0]R0("a b c d e 0")[0]
  -> [0]R0("e a b c d 1")[0]
   ...
  -> [0]R0("a b c d e 15")[0]
  -> [0]R1("e a b c d 16")[0]
  -> [0]R1("d e a b c 17")[0]
   ...
  -> [0]R2("e a b c d 31")[0]
   ...
  -> [0]R4("b c d e a 64")[0]
  -> [0]R4("a b c d e 65")[0]
  -> [0]R4("b c d e a 79")[0]
  -> output[0];

};
```

```
PipeItMaster SHA1App {

  SHA1Root() -> output[0];
  input[0] -> [0]SHA1Sink();

};

SHA1App[0] -> [0]SHA1;
SHA1[0] -> [0]SHA1App;
```

We used the *PipeIt* toolchain to build the SHA1 transformer. We initially run a simplified version of the pipeline, with only 6 components, in emulation mode. As a data input we used a variety of statically declared blocks in an endless loop. The profiling data indicated that the overhead of local I/O corresponds to 14% of the total amount of processing. This was expected, because each component performs just few operations. Actually, this overhead becomes even more notable if compared to the *PipeIt* SHA1 implementation to the original implementation of the *Crypto* library, which used loop unrolling and inline invocation.

At this point, one could revisit the defined pipeline structure to create a coarser partitioning which would be less I/O intensive. On the other hand, this would reduce flexibility in terms of load balancing. The final decision is for the developer to make and is all a matter of tradeoffs.

The HMAC computation can be expressed as a function of the SHA1 computation, as follows:

$$hmac = sha1((K \oplus opad)||sha1((K \oplus ipad)||msg))$$

where sha1 is the cryptographic hash function, K is a secret key padded to the right with extra zeros to the block size of the hash function, *msg* is the message to be authenticated, $||$ denotes concatenation, $\oplus$ denotes exclusive or (XOR), the outer padding *opad* equals $0x5c5c5c..5c5c$ and inner padding *ipad* equals $0x363636..3636$ (both are two one-block-long hexadecimal constants). Despite the fact that two consecutive SHA1 computations are needed, it is the inner invocation that actually performs the heavy work to create the message digest. The outer invocation performs a single iteration because by convention *opad* size is equal to one SHA1 computation blocksize. For the HMAC implementation we use a different configuration file that *reuses* the previously described

SHA1 implementation as follows:

```
PipeItArray SHA1 {
  input[0] .... -> output[0]
};

PipeItMaster HMACApp {
  HMACLibRoot() -> output[0];
  input[0] -> [0]HMACLibSink();
};

HMACApp[0] -> [0]SHA1;
SHA1[0] -> [0]HMACApp;
```

We have integrated the *PipeIt* version of HMAC into *Crypto* library so that it can be used from well known applications like secure copy (scp).

Note that several applications might wish to execute the same type of a *PipeIt* computation (e.g., HMAC) at the same time. If there are enough resources, several different pipelines will be deployed on the PPA. Else, some computations will be performed on the master processor, using a fully “collapsed” pipeline structure. This is what happens in our prototype platform, which has very limited resources indeed.

4.4.3 Experimental results

We firstly deployed the *PipeIt* version of SHA1 on the our prototype platform, as an autonomous executable. Then we created a simple application that calls the *PipeIt* version of HMAC via the *Crypto* library. The HMAC keys are set once, at the beginning of the computation. Both applications feed a variety of predefined blocks of data in an endless loop.

For both applications, we have performed measurements for the case where: i) the pipelined computation runs only on the master processor; ii) 4 processors are used to run the pipelined computation, whereas the master merely runs the root and sink components; and (iii) 5 processors including the master are used to run the pipelined computation. In all cases, the system was unloaded, i.e., there were no other active applications running at the same time. (iv) Two instances of the SHA1 computation are executed concurrently. The 4 CPUs are evenly distributed among

Computation	original	*PipeIt* seq	*PipeIt* 4 CPUs	*PipeIt* 5 CPUs
SHA1	$55.2Kb/s$	$43.1Kb/s$	$156kb/s$	$192.7kb/s$
HMAC	$54.1kb/s$	$42.8kb/s$	$154.9kb/s$	$191kb/s$
Simultaneous Execution of 2 Instances				
SHA1	Instance 1	$76kb/s$	Instance 2	$76kb/s$

Table 4.1: SHA1 and HMAC performance on the prototype platform

these two instances (each one is assigned 2 CPUs) while the main CPU is occupied with root and sink components only. Notably, the second instance arrives after the execution of the first one has started using all the available CPUs. In this experiment the platform is reconfigured at runtime to form different CPU interconnections using the support we have previously developed[40]. The *PipeIt* execution transition delay for the pipeline restructuring in this case was 580 usecs. The results are depicted in Table 4.1, including the performance of the original (sequential) code as a reference.

The first observation is that the sequential execution of pipelined computations introduces a notable overhead of about 28% compared to the original sequential code. This is because *PipeIt* explicitly invokes each component in a loop, and cannot optimize code, e.g., by using inline functions. The second observation is that the speedup achieved when using 5 processors is $4.45x$ compared to the sequential execution of the pipeline, and $3.5x$ compared to the highly optimized original version. The performance of HMAC closely follows that of SHA1, which is expected since the HMAC relies on SHA1, being slightly slower, as expected.

Notably, the $4.45x$ speedup obtained when using 5 processors is far away from what should be theoretically possible. This is because the pipeline configuration used in this case also suffers from the execution and communication overhead imposed by the *PipeIt* runtime for co-located components. More specifically, for SHA1, 12 components are co-located with the root and sink while each other array core is assigned 17 components. This overhead obviously increases as less processors are used, and the number of co-located components increases, as demonstrated by the performance figures for the case of using just 4 processors (where each processor is assigned 20 components, and the master just the sink and root).

It is possible to boost performance if one revisits the pipeline structure of SHA1, to adapt the number of components to the actual platform capabilities, by introducing fewer and more heavy-weight components. To verify this, we created a second version of SHA1 with 6 components where each one contains an integrated optimized version of the code of co-located components for the case of 5 processors. This *PipeIt* program achieved a throughput of $253.9kb/s$ which roughly equals to a $4.6x$ performance improvement over the original highly optimized sequential code. There is still some inefficiency, due to the actual data transfer overheads between the processors and the scheduling overhead of the root and sink threads on the master processor, which partly reflects the limitations of our prototype platform. As a downside, this pipeline is very coarse-grained and cannot result in a speedup greater than $5x$, even if the underlying platform features many unused cores while the initial fine grained partitioning version may theoretically reach $80x$ on an 80-core array. Having fewer pipeline components/stages also limits the options of the runtime in terms of load balancing. Obviously, a reasonable tradeoff must be done at design time by the developer, with some class of target platforms in mind.

Having the *PipeIt* version of HMAC integrated in the *Crypto* library made it trivial to exploit this implementation from within an *existing* and *conventional* application like *scp* (version 2 of the SCP protocol). To get a feeling about the performance impact, we copied over Ethernet a $10MByte$ file from a PC connected on the same switch as the Suzaku (so that the network throughput does not become the bottleneck). With the original SHA1 implementation, file transfer was done at $18.1Kb/s$. The *PipeIt* version reached $22.4kb/s$, an improvement of $1.23x$. In this case, during the initialization of the HMAC code that takes place at the beginning of *scp* the SHA1 computation is arranged to evenly execute on all 5 processors (including master). When processing starts, the master gets occupied with the execution of the conventional part of secure copy, so our simple runtime service, because of the increased main processor usage, the next time is inquired it instructs *PipeIt* support to rearrange the SHA1 computation to use just 4 processors (with the master running only the root and sink). Of course, the end effect at the application level is not analogous to the speedup that is obtained at the level of SHA1 (for the same case; see Table 4.1). This is because the message authentication process is only a part of the secure copy, which also requires AES encryption and appropriate network stack handling.

To verify that dynamic load balancing works better when the pipeline has a finer granularity, we also used the coarse-grained pipeline version of SHA1, described previously. For this configuration, file transfer speed never got beyond $20.4kb/s$, reducing the previous improvement by roughly 50%. As it turns out, given that the master processor is heavily loaded due to the execution of the conventional part of secure copy, *PipeIt* has to place two (out of five) components on one of the four processors. This “balancing” results in a far more uneven distribution of the computation than when using the fine-grained pipeline, in which case each processor is assigned 20 components.

The total size of the *ComponentExecutionMap* and *IOMap* for the case of fine grained SHA1 *PipeIt* computation is 80 bytes. On our target platform, we have measured that each array processor needs $110usecs$ to change to a new configuration for this computation which is negligible.

4.5 Related Work

Ambric RPPA architecture [7] integrated with a master CPU that can run a full-fledged OS would be an ideal high-performance target for our framework. Currently Ambric uses a structural object programming model. The programmer separates the application into high-level processing objects which can be developed independently and can execute asynchronously with each other, at their own clock speed, on their own dedicated processor core. Then the programmer defines how the objects should be interconnected to form a computational model close to a Kahn Process Network [1]. The development toolchain provides an emulation environment for debugging and performance estimation, as well as tools to compile objects, configure and load the processor network at runtime. In this approach, the authors consider ambric an application specific processor and compare it to DSPs and FPGA accelerator designs. The framework addresses only the development on the array processors which are configured to carry out a computation and assumes that providing and collecting processed data which implies the integration with a target platform and a master processor should be handled by external applications. Since ambric building blocks are processor cores, we believe that the integration of array functionality in an OS context is also important. To that end, our proposed load balancing approach could help to efficiently accommodate concurrently running applications on ambric processor improving this way its performance and response in a general purpose, multi-tasking environment. Cell processor[43] could have been another possible target for our framework, but in this case all processor cores, apart from private local memories, share the main system memory. Therefore, for cell targets, dynamic load balancing should be explored in a more appropriate programming context. We plan to introduce *PipeIt* support for Cell in the future.

PipeIt architecture design approach has been heavily influenced by the click modular router[12] network packet processor framework which is employed to abstract complexity for a different application domain. Click features *C++* objects that are called *elements* that also have input and output ports to receive network packets on which they perform a specific type of operation. Click employs a configuration language as well, that specifies a network of interconnected elements and to create different packet processing configurations and features a runtime to execute them appropriately. These configurations, contrary to *PipeIt* approach are loaded and can be changed at runtime to dynamically modify the packet processing behaviour. What is more important is that the click framework and the corresponding programming model which is close to *PipeIt*, has been heavily adopted by the network systems community as fast to learn and easy to use. As a result the number of network systems researchers that use click is impressively growing[69].

StreamIt [51] language introduces a high-level model for a broader application domain than *PipeIt* which includes all types of applications that use a stream as an abstraction. In this approach the compiler can automate tasks such as partitioning, static load balancing, layout and memory management and is currently being used to program the RAW architecture[52]. In *StreamIt* there is no notion of configuration, the components are hardwired and explicitly interfaced to a neighbour. Moreover, the compiler produces a highly optimized executable aimed to run on dedicated resources on the target platform and not in a general purpose context. In this case, to our knowledge, load adaptation to available resources that may change at runtime has not yet been considered.

The approach to define components and wire them in a configuration file has also been introduced to nicely abstract the event-driven nature of the resource constrained embedded systems. More specifically, network embedded systems C *nesC*[14] is a component-based, event-driven programming language used to build applications for the TinyOS, an operating environment designed to run on embedded devices that are used in distributed wireless sensor networks. *nesC* is built as an extension to the C programming language with components ”wired” together to run applications on TinyOS.

In [58] code annotations that enable source-level restructuring are proposed to achieve coarse grained pipelined parallelism. A dynamic analysis tool is used to help the programmer balance pipeline stage workload, and the underlying runtime support is used to fork the pipeline stages as different private memory processes that communicate via *Unix* pipes. Pipeline parallelism is also exploited in [38] using the techniques of Decoupled Software Pipelining [59], also employing thread-level speculation to opportunistically execute multiple loop iterations in parallel and extract this way parallelism from previously unparallelizable loops. Both approaches, while they can be used for a processor array target assume an isolated environment that runs on a shared memory homogeneous multiprocessor system where all the resources serve one pipelined application at a time. General purpose execution context is not considered.

In the same spirit several dynamic task partitioning methods have been proposed [61, 62] that deal with the partitioning of unstructured mesh problems like Computational Fluid Dynamics. In [61] the sub tasks of a computation are developed using an appropriate programming model and a software framework called Data Movement and Control Substrate(DCMS)[63], which features an extension called Mobile Object Layer(MOL)[64] that enables transparent task migration between processors at runtime. The authors have extended the MOL concept and have added appropriate load balancing routines that communicate with respective runtime support. While the main principles of this higher-level approach are close to ours, tasks are just migrated and not restructured and platform targets are not tightly coupled systems-on-chip; They must feature CPUs that can operate in the traditional time shared manner. Moreover, load balance calls must be added and invoked respectively at the application level.

Task dynamic partitioning support that enables load balancing at runtime has been proven to achieve good performance on Multiple Instruction Multiple Data (MIMD) targets. In [65] the authors have implemented in the Mul-T[66] runtime system support that enables parallel tasks to execute either independently or in the context of a few system tasks based on the resource availability. As in our approach, the programmer is expected to identify and expose parallelism without worrying about the granularity while the system limits parallelism to meet platform capabilities. Their implementation futures Lisp constructs that must be used to define independent execution entities and is tightly integrated with Mul-T runtime design. This approach is primarily targetted at parallel algorithms that are naturally expressed at a fine level of granularity which is typical workload for MIMD architectures and does not consider pipeline applications. Moreover contrary to our work, the partitioning decisions have to be explicitly coded and are application driven.

In [70] a different approach is proposed to improve load balance performance. The authors identify as a basic parallel application performance problem the assumption that all platform processors are fully available to perform a given computation. To solve this, they present a user-level task scheduler that schedules tasks onto a fixed collection of processes which the operating system kernel further schedules onto a fixed collection of processors. The proposed scheduler implements the work stealing algorithm where each process is assigned a pool of independent tasks to execute one-by-one and after it depletes it can steal tasks from a neighbour pool.

4.6 Summary

In this chapter we have presented *PipeIt*, a framework to develop pipeline applications for embedded RPPA targets. We have preferred to exploit pipelined parallelism because all computations can be transformed to a certain extend for pipelined execution and, moreover, it is a more suitable concept for the use of distributed memory, possibly heterogeneous, PR system on-chip.

Embedded systems require flexibility at all design levels, and therefore reconfigurable hardware designs combined with a general purpose runtime are good candidates even for commercial solutions[20], [7]. Our vision is to have such a general purpose system where all userland applications have the ability to deploy stages of execution to dedicated processor cores and, to that end, support for dynamic load balancing and corresponding selective component scheduling is of major importance because it enables effective resource usage.

Our approach is in line with the principle of letting the programmer specify the maximum parallelism at the application-level, while making system responsible for deploying the code in the best possible manner, depending on the underlying platform characteristics and the current execution context. To accomplish this, we have strived for a generic design that can be ported/applied to radically different platforms using appropriate backends. We are currently working on *PipeIt* towards two different directions: i) increasing the codebase by transforming to *PipeIt* applications more ciphers from the *Crypto* library and ii) we are extending the concept and building backend support to take advantage of the Cell Broadband Engine architecture features.

Chapter 5

Conclusion and outlook

We have addressed challenges at the operating system and application programming level for the emerging hybrid reconfigurable parallel architectures. We regard as particularly important contributions the abstraction of runtime hardware reconfiguration as a typical system service and the synergistic mechanisms between the OS and the development toolchain that enable dynamic load balancing of workload. With this support hardware/software codesigned and massively parallel applications can now be used in typical multitasking context, where an application may arrive at any point in time and share the platform resources contrary to what is currently possible with dedicated application specific setups. Moreover we demonstrate that operating system support remains relevant in the context of emerging massively parallel platforms because it can facilitate mechanisms to abstract performance asymmetry of the target platform processing elements. In most cases this support allows the developer to partition the application without worrying about the target platform available resources because the same partitioning can be effective on a variety platforms with different number of elements and performance. Our dynamic load balancing approach has been explored and implemented for distributed memory systems. We believe that the dynamic profiling concept can be also used on shared memory applications and the design should be revisited for the respective application domain.

Parallel reconfigurable platforms can also be used to improve availability and fault tolerance. Operating systems should be further changed to detect faults, migrate and replay application execution. Research on these topics is already active but it is primarily being addressed in virtual machine context.

Bibliography

[1] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP'74*, pages 471–475, 1974.

[2] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.

[3] J. R. Hauser and J. Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 12, Washington, DC, USA, 1997. IEEE Computer Society.

[4] Timothy John Callahan. *Automatic compilation of c for hybrid reconfigurable architectures*. PhD thesis, 2002. Chair-Wawrzynek,, John.

[5] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *MICRO*, 2003.

[6] Nvidia Corporation, http://www.nvidia.com/cuda. *Nvidia CUDA*.

[7] Michael Butts, Anthony Mark Jones, and Paul Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. In *FCCM '07*, 2007.

[8] picoChip Inc, http://www.picochip.com. *picoChip Multicore DSP*.

[9] IntellaSys Inc, http://www.intellasys.net. *Intellasys SEAforth-40*.

[10] Dean Truong, Wayne Cheng, Tinoosh Mohsenin, Zhiyi Yu Toney Jacobson, Gouri Landge, Michael Meeuwsen, Christine Watnik, Paul Mejia, Anh Tran, Jeremy Webb, Eric Work, Zhibin Xiao, and Bevan Baas. Hardware and applications of asap: An asynchronous array of simple processors. In *HotChips 2008*, August 2008.

[11] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. Paro: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In *ARC '08*, 2008.

[12] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3), 2000.

[13] Timothy G. Mattson. How good is openmp. *Sci. Program.*, 11(2):81–93, 2003.

[14] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

[15] Yani Matsumoto and Antoin Masaki. Speed improvement of FPGA by mixing multiple-gate-width routing switches. In *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, pages 14–22, 2005.

[16] Lesley Shannon and Paul Chow. 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*, pages 190–199, 2004.

[17] Xilinx Inc. *Two Flows for Partial Reconfiguration: Module Based or Difference Based.*, 2003. Aplication Note XAPP290.

[18] Bryan Fletcher. FPGA Embedded Processors: Revealing True System Performance. In *Embedded Systems Conference San Fransisco*, number ETP-367, 2005.

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

[20] Atmark Techno Inc, http://www.atmark-techno.com/en/products/suzaku. *Suzaku Series*.

[21] Nigel Cunningham. *Suspend2 project*. http://www.suspend2.net/.

[22] Greg Ungerer. netflash utility. http://docs.linux.com, October 2002.

[23] Williams J Lu Y, Bergmann N. Dynamic loading of peripherals on reconfigurable system-on-chip. In *SPIE Microelectronics: Design, Technology, and Packaging II*, volume 6035, 2005.

[24] Greg Stitt, Frank Vahid, Gordon McGregor, and Brian Einloth. Hardware/software partitioning of software binaries: a case study of h.264 decode. In *CODES+ISSS*, pages 285–290, 2005.

[25] Miljan Vuletic, Laura Pozzi, and Paolo Ienne. Programming transparency and portable hardware interfacing: Towards general-purpose reconfigurable computing. In *ASAP*, pages 339–351, 2004.

[26] Matthias Dyer, Christian Plessl, and Marco Platzner. 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*, pages 292–301, London, UK, 2002. Springer-Verlag.

[27] Edson L. Horta and John W. Lockwood. Parbit: A tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (fpgas). Technical Report WUCS-01-13, Washington University Department of Computer Science, 2001.

[28] Anup Kumar Raghavan and Peter Sutton. 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*, page 192, Washington, DC, USA, 2002. IEEE Computer Society.

[29] Inc Atmel. *Field Programmable System Level Integrated Circuits (FPSLIC)(2002)*. http://www.atmel.com/products/FPSLIC/.

[30] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. A Self-Reconfiguring Platform. In *Proceedings of Field Programmable Logic and Applications*, pages 565–574, 2003.

[31] Reetinder P. S. Sidhu and Viktor K. Prasanna. 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*, pages 698–709, London, UK, 2002. Springer-Verlag.

[32] Ryan J. Fong, Scott J. Harper, and Peter M. Athanas. A versatile framework for fpga field updates: An application of partial self-reconfiguation. In *IEEE International Workshop on Rapid System Prototyping*, pages 117–123, 2003.

[33] John Williams and Neil Bergmann. 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*, pages 163–169, 2004.

[34] Vincent Nollet, Jean-Yves Mignolet, Andrei Bartic, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Hierarchical run-time reconfiguration managed by an operating system for reconfigurable systems. In *Engineering of Reconfigurable Systems and Algorithms*, pages 81–87, 2003.

[35] Christian Haubelt, Dirk Koch, and Jürgen Teich. Basic OS Support for Distributed Reconfigurable Hardware. In *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 18–22, Samos, Greece, July 2003.

[36] Xilinx Inc. *Architecting Systems for Upgradability with IRL*, 2001. Aplication Note XAPP412.

[37] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[38] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David I. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture*, 2007.

[39] John Williams. *The Microblaze-uClinux kernel port Project*. http://www.itee.uq.edu.au/ jwilliams/mblaze-uclinux/.

[40] Dimitris Syrivelis and Spyros Lalis. System- and application-level support for runtime hardware reconfiguration on soc platforms. In *USENIX ATC '06*.

[41] Wim van Dorst. *BogoMips mini-Howto*. http://tldp.org/HOWTO/BogoMips/.

[42] The xiph open source community. *Tremor Ogg Vorbis decoder*. http://wiki.xiph.org/index.php/Tremor.

[43] Michael Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, 2006.

[44] Michael Gschwind, David Erb, Sid Manning, and Mark Nutter. An open source environment for cell broadband engine system software. *Computer*, 40(6):37–47, 2007.

[45] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Comput. Archit. News*, 33(2):506–517, 2005.

[46] Sun Wei. *A FPGA-based Soft Multiprocessor System for JPEG Compression*. www.opencores.org.

[47] Kevin Skey, John Bradley, and Karl Wagner. A reuse approach for fpga-based sdr waveforms. In *MILCOM 2006*, pages 1–7, 2006.

[48] Kaushik Ravindran, Nadathur Rajagopalan Satish, Yujia Jin, and Kurt Keutzer. An fpga-based soft multiprocessor system for ipv4 packet forwarding. In *15th International Conference on Field Programmable Logic and Applications (FPL-05)*, pages pp 487–492, 2005.

[49] Wei Du, Renato Ferreira, and Gagan Agrawal. Compiler support for exploiting coarse-grained pipelined parallelism. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 8, 2003.

[50] Mary H. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 49, 1995.

[51] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.

[52] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[53] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[54] F. Mueller. Pthreads library interface.

[55] B. Dreier, M. Zahn, and T. Ungerer. Parallel and distributed programming with pthreads and rthreads. In *Proc. of the 3rd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 34–40, 1998.

[56] T. Loos and R. Bramley. MPI performance on the SGI Power Challenge. pages 203–206, 1996.

[57] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending openmp for numa machines. *Sci. Program.*, 8(3), 2000.

[58] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. *40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, 2007.

[59] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[60] Yoshiaki Sakae, Satoshi Matsuoka, Mitsuhisa Sato, and Hiroshi Harada. Towards dynamic load balancing using page migration and loop re-partitioning on omni/scash. In *In Proceedings of The Fourth European Workshop on OpenMP (EWOMP 2002)*, 2002.

[61] Kevin Barker, Andrey N. Chernikov, Nikos Chrisochoides, and Keshav Pingali. A load balancing framework for adaptive and asynchronous applications. *IEEE Trans. Parallel Distrib. Syst.*, 15(2):183–192, 2004.

[62] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997.

[63] Kevin Barker, Ni Chrisochoides, J Dobbelaere, D Nave, and K Pingali. Data movement and control substrate for parallel adaptive applications. *Concurrency Practice and Experience*, 14:77–101, 2002.

[64] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel. Mobile object layer: a runtime substrate for parallel adaptive and irregular computations. *Adv. Eng. Softw.*, 31(8-9):621–637, 2000.

[65] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy Task Creation: A Technique For Increasing The Granularity Of Parallel Programs. Technical Report MIT/LCS/TM-449, 1991.

[66] D. A. Kranz, Jr. R. H. Halstead, and E. Mohr. Mul-t: a high-performance parallel lisp. *SIGPLAN Not.*, 24(7):81–90, 1989.

[67] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129. ACM, 1998.

[68] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, 1994.

[69] Eddie Kohler. *The Click Modular Router Project*. http://read.cs.ucla.edu/click/.

[70] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98*, pages 119–129. ACM, 1998.