UNIVERSITY OF CAPE TOWN

MURAC: A unified machine model for heterogeneous computers

by

Brandon Kyle Hamilton

A thesis submitted in fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering & the Built Environment
Department of Electrical Engineering

February 2015

ABSTRACT

MURAC: A unified machine model for heterogeneous computers

by Brandon Kyle Hamilton

February 2015

Heterogeneous computing enables the performance and energy advantages of multiple distinct processing architectures to be efficiently exploited within a single machine. These systems are capable of delivering large performance increases by matching the applications to architectures that are most suited to them. The Multiple Runtime-reconfigurable Architecture Computer (MURAC) model has been proposed to tackle the problems commonly found in the design and usage of these machines. This model presents a system-level approach that creates a clear separation of concerns between the system implementer and the application developer. The three key concepts that make up the MURAC model are a unified machine model, a unified instruction stream and a unified memory space. A simple programming model built upon these abstractions provides a consistent interface for interacting with the underlying machine to the user application. This programming model simplifies application partitioning between hardware and software and allows the easy integration of different execution models within the single control flow of a mixed-architecture application.

The theoretical and practical trade-offs of the proposed model have been explored through the design of several systems. An instruction-accurate system simulator has been developed that supports the simulated execution of mixed-architecture applications. An embedded System-on-Chip implementation has been used to measure the overhead in hardware resources required to support the model, which was found to be minimal. An implementation of the model within an operating system on a tightly-coupled reconfigurable processor platform has been created. This implementation is used to extend the software scheduler to allow for the full support of mixed-architecture applications in a multitasking environment. Different scheduling strategies have been tested using this scheduler for mixed-archictecture applications.

The design and implementation of these systems has shown that a unified abstraction model for heterogeneous computers provides important usability benefits to system and application designers. These benefits are achieved through a consistent view of the multiple different architectures to the operating system and user applications. This allows them to focus on achieving their performance and efficiency goals by gaining the benefits of different execution models during runtime without the complex implementation details of the system-level synchronisation and coordination.

Contents

List of Figures

List of Tables

Listings

Abbreviations

AA	Auxiliary Architecture
API	Application Programming Interface
ASIP	Application Specific Instruction Set Processor
BAA	Branch to Auxiliary Architecture
CFS	Completely Fair Scheduler
CGRA	Coarse-Grained Reconfigurable Architecture
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HLL	High-Level Language
HLS	High-Level Synthesis
ICAP	Internal Configuration Access Port
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
MPSoC	Multiprocessor System-on-Chip
PA	Primary Architecture
PLD	Programmable Logic Device
RISC	Reduced Instruction Set Computer
RISP	Reconfigurable Instruction Set Processor
RPA	Return to Primary Architecture
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
VLIW	Very Long Instruction Word

Acknowledgements

I would like to thank my thesis co-supervisor Dr. Hayden So whose deep knowledge of the field has guided me in exploring the ideas leading to this work. This thesis draws heavily on the insights that I have gained through our many inspiring discussions over the past few years. I deeply value the time that I have been fortunate enough to have spent working with him in Hong Kong.

I am thankful to my thesis supervisor Prof. Michael Inggs for the continuous support throughout my time at UCT. He has always provided the sound advice and guidance required to keep me on track with this project. He has been instrumental in exposing me to the many interesting people in this field of research, and enabling me to have the opportunity to travel numerous times in support of my work.

Furthermore, I am grateful to Dr. Alan Langman for the constant encouragement. He is a friend that is always willing to advise and motivate.

I owe many thanks to the numerous developers in the open source community. The availability and free access to open source software has been critical in the development of this work, particularly that of the Linux kernel.

I am extremely thankful to my parents who laid the foundations and gave me the freedom and support that enabled me to successfully pursue this endeavour, away from home, for many years.

Finally, I owe a special thanks to my wife Alexandra, who has been a patient and consistent source of moral support throughout this period. Beyond this, she has offered invaluable help with improving the legibility of this work.

This work was partly funded by SKA South Africa.

For Alexandra

Chapter 1

Introduction

Traditional computing, built upon homogeneous architecture machines, has increasingly focussed on the use of multiple cores and advances in techniques such as instruction-level parallelism in an effort to scale to the demands of modern applications. However, due to the difficulty of parallelising applications and operating systems, the performance gain achieved through these approaches has not scaled proportionally to the number of cores [Wal91]. Additionally, in pushing the boundaries of high-performance computing, the energy requirements are quickly becoming the dominant limiting factor. The laws of thermodynamics and quantum mechanics limit the computational power increases achievable in silicon devices. This has led to the need for large portions of the transistors to be turned off at any point in time - so called *dark silicon* - as chips are unable to effectively further dissipate the heat produced. This effect limits the multi-core scaling that takes advantage of Moore's law [EBSA+12]. As the domain for high-performance computation has grown to encompass mobile applications, the concomitant strict energy requirements of these devices has also driven the research into more energy-efficient architectures. It has been recognised that radically different architectures from regular CPUs will be required in order to meet the future need for high performance and energy efficient computing [Sin11].

1.1 Heterogeneous Computing

In contrast to homogeneous computing, where only one mode of parallelism can be employed within a machine (e.g SIMD or vector processing), heterogeneous computing takes advantage of diverse high-performance architectures. Such heterogeneous machines enable the performance and energy advantages of the multiple distinct architectures to be efficiently exploited, which are capable of delivering order of magnitude performance increases by matching the applications based on certain classes of algorithms to the application-specific suited architectures [KPSW93]. These machines offer a desirable

target for many application domains that have more than one type of embedded parallelism. For example, heterogeneous machines containing CPUs coupled with graphics processing units (GPUs) have enabled some large performance enhancements for various application domains, especially science-related applications. However, while GPU devices typically provide a very high performance-per-price ratio, they come with the cost of high energy requirements [DAF11].

Heterogeneous machines, most commonly composed of general purpose processing (GPP) units combined with specialized compute hardware accelerators (HA) or co-processors, are often very different from regular CPU-centric devices. A wide range of specialized architecture devices have been effectively employed in heterogeneous machines, including GPUs, Digital Signal Processors (DSPs), Very-Long-Instruction-Word (VLIW) processors, heterogeneous System-on-Chips (SoCs) like the Cell Broadband Engine, and programmable logic devices (PLDs). Each of these architectures provide different performance-price-energy profiles that may be suited toward certain application domains.

Reconfigurable (or adaptive) computers have proven to be greatly beneficial to a wide range of applications [HD07, EGEAH+08]. These machines contain devices composed of reconfigurable programmable logic fabrics and include both fine-grained architectures such as field programmable gate arrays (FPGAs), and coarse-grained architectures that contain configurable special-purpose processing blocks such as MorphoSys [LSLB00], ADRES [BBKG07] and many others [Vas07]. For computational purposes, these devices can be used to implement full heterogeneous systems that support both the adaptation of the number and type of processing elements as well as the communication infrastructure between them, customised to application requirements at both design and runtime. These programmable logic devices are commonly connected as additional accelerators or co-processors controlled by a host CPU, capable of being tightly-coupled on the same silicon as the GPP (e.g the Xilinx Zynq platform) to achieve low-latency communication. In these platforms, the GPP is typically tasked with the control and input-output (I/O) operations while the performance-critical computational operations are offloaded to the dedicated application-specific architectures realized in the reconfigurable fabric. In contrast to the fixed-architecture heterogeneous machine, a reconfigurable computer allows the application to *dynamically* change the architecture upon which it may execute during runtime, allowing a potentially unlimited variety of application-specific architectures that can be tailored to optimally match the application requirements. Due to the lower energy requirements of reconfigurable-fabric devices, these reconfigurable systems are able to provide a high performance-per-watt ratio.

Application-specific systems are favourable for solving particular classes of problems [Sin11], indicating that there is a performance benefit for using multiple types of processing elements within a machine rather than a single fixed-accelerator architecture. By allowing for distinct execution models and performance characteristics, applications running on

these hybrid platforms are able to utilise the particular computational advantages of each type of device as needed (e.g. a GPU can be utilised for data-intensive floating point operations, whereas dedicated cryptographic algorithms can be realized on an FPGA).

1.1.1 Heterogeneous application design

Optimal performance is achieved through an efficient mapping of the algorithm to the underlying architectures available via the multiple processing elements in a heterogeneous system. This mapping will vary amongst different problem domains [ABC+06], each requiring architectural traits that are suited for their efficient computation. Thus a key challenge in the efficient use of heterogeneous machines is presented in the complicated programming models that are required to take maximum advantage of the available architectures. This mapping process is often very difficult and complex [FB89], with numerous heuristics employed to aid with achieving an acceptably performing design [BSB+01].

Accelerator device hardware vendors provide libraries and frameworks that allow the explicit handling of the interface between the accelerator and CPU as well as aiding in the execution of specified operations on the accelerator. The most popular of these are the GPU targeted libraries, CUDA and OpenCL, which have become commonplace due to the high market penetration of GPU devices. By using such frameworks, the application developer is able to create a heterogeneous computing application using a familiar software systems programming language that has been extended with accelerator specific functionality which the suitable compiler will map into an application capable of running on the targeted system.

Automatic translation tools for reconfigurable devices are capable of converting traditional sequential programming algorithm code into programs making use of different execution models. So called C-to-Gates tools, such as Xilinx Vivado High-Level Synthesis (HLS) and Altera SDK for OpenCL, allow the programmer to specify algorithms in the familiar sequential programming paradigm, and then perform automatic analysis and synthesis to the target hardware architectures supporting the desired computational models. These tools are an active topic of research and commercial development as they provide traditional software developers a lower barrier to entry to hardware design and development by allowing them to use familiar paradigms. Additionally, these tools provide the benefit of specifying the full system in a single programming language, which aids in reducing the complexity of the hardware/software interface and makes the program easier to understand [BRS13].

1.2 Problems statement

There are numerous difficulties that developer and system designers must overcome when creating and working with heterogeneous systems, particularly:

Vendor specific implementations As heterogeneous computing devices and components such as CPUs, GPUs and FPGAs are traditionally developed by different vendors with varied design objectives, for the most part they share very few similarities. For instance, while GPU designs rely on a software-centric design flow and proprietary runtime libraries for execution, FPGA designs remain tied closely to the traditional hardware-centric design methodology with little to no operating system runtime support. The lack of vendor agreement on standard interfacing models leads to each system being mostly developed from scratch with little design re-usability.

Complexity of design Heterogeneous systems have been typically employed to solve the problem of *scaling to volume*, capable of high performance processing large datasets using fairly static programs and architectures. A more difficult problem of *scaling to complexity* is becoming increasingly important. Due to the heterogeneous nature of the underlying architectures, there is often no commonality in design approach and programming model, leading to complicated designs that end up being composed of a combination of multiple paradigms with a large amount of interface logic. Additionally, different applications usually demand distinct types of computational components for optimal performance improvements. Users of reconfigurable devices are regularly required to perform difficult custom application mapping solutions to unfamiliar platforms. There is a high degree of complexity in the boundaries between the distinct execution models that often leads to ad-hoc approaches to parallel programming. Being able to exploit the potential of the these machine requires not only experience in application and software design, but computer architecture and hardware-description languages. Often the algorithms will need to be rewritten in a completely different language, using a completely distinct computation model to that of the original.

As more processing elements are added to a system, further issues arise as to the best approach at effectively maximising the performance of applications running on a heterogeneous machine. Common techniques of parallelising traditional algorithms to suit the system using multiple threads has many drawbacks, although they do still prove useful [HM08]. Increased non-determinism often leads to race-conditions in applications that are difficult to diagnose and become increasingly complex to understand and maintain. Practical drawbacks include the performance penalties due to the overhead introduced by thread synchronisation mechanisms.

Furthermore, once systems have been developed and deployed in the field, ongoing maintenance and support costs become burdensome due to this high degree of complexity. For example, due to the problems mentioned above it is often the case that the operational and maintenance skill-set required is often not readily available as the original system developers are no longer accessible.

Lack of an overall system view of both hardware and software Component devices in heterogeneous machines feature significantly varied architectures and functionality from general purpose CPU cores, often leading to application designs that are device-centric and modelled around the execution model of the individual hardware components. This has led to a lack of an overall system view encompassing both hardware and software components that is required for an effective approach to the design, implementation and execution of computation on heterogeneous platforms. This becomes even more difficult in reconfigurable systems, where the underlying hardware architecture is no longer fixed, but often being altered dynamically throughout the application life-cycle. In these system it is not only the components themselves, but rather the composition of the components that is critical to the performance and efficiency of the system. As such, for sustainable development in heterogeneous computing it is essential to develop a unified view of disparate processing elements within the same system.

Lack of portability Portability of systems and application code is routinely not feasible as systems are often composed of multiple diverse computing devices not only from different vendors, but could even contain large variations within the generations and versions of each component. Programs written for these systems include large portions of device specific initialisation and interfacing logic. Each component in the system typically requires it's own programming model, libraries and interfaces, thereby requiring considerable effort to coordinate compatibility. Changing and often incompatible versions of the software frameworks, device drivers and runtime ecosystems limit the ability to use the same programs across diverse machines.

Lack of re-usability Heterogeneous system applications are mostly designed in a manner that is very specific and suited to the particular problem being solved, leading to very little commonality between designs. With the current tools and methods available it is fairly difficult to achieve design reuse between heterogeneous applications.

Difficulty of collaboration and community involvement Heterogeneous and reconfigurable systems and tools are often prohibitively expensive for non-professional users, particularly in the case of devices and components that have not yet achieved economy of scale via mass market adoption. Combined with the numerous limitations described above, these problems impose serious impediments to fostering a community

of heterogeneous computing users and facilitating collaboration that would produce the many benefits that have been seen in open source communities [Web04].

1.3 Contributions of this work

There is a clear need for a consistent machine model that describes the interaction between the multiple processing architectures within an heterogeneous system. In an attempt to tackle the problems highlighted above, the main contribution of this work comprises the *Multiple Runtime-reconfigurable Architecture Computer (MURAC)* model. This unified machine model captures the hardware/software interface within a single-context process, eliminating the implicit forking of a user process when a processing element such as an accelerator within a heterogeneous machine is utilised. The complex intra-process synchronisation mechanisms are shifted out of the role of a programming model notion and into the role of *implementation* of a programming model notion as system-level details are hidden below a consistent abstraction. The new notion is more re-usable and portable by virtue of being uniform and it significantly reduces the amount of complex coordination and communication normally required within an application. Using this model, application code becomes independent of the interfacing and interaction between the underlying architectures. Instead, the system designer is tasked with implementing an optimised micro-architecture to deal with these system-specific details.

Heterogeneous processing elements within a MURAC system are treated as alternative execution architectures from the point of view of the application, and not as accelerators controlled by a host CPU. This removes the burden of a manually managed and coordinated master-slave relationship typically required in heterogeneous applications. The concept of a *mixed-architecture application* enabled by this abstraction is employed, whereby a single instruction stream is able to dynamically morph the underlying architecture to carry out computation in any desired computation model supported by the machine during runtime. Thus the MURAC model supports the efficient application of computational granularity. To maximise performance, the best balance between computational load and communication overhead needs to be found, and can be achieved by allowing the system to change during runtime between very fine granularity of an FPGA to larger data-path width CPUS, GPUs or data-stream-driven reconfigurable data-path arrays (rDPA). Automatic compilation techniques that provide useful advantages in translating algorithms between diverse architectural models benefit from such a standardised consistent model that is provided by this simplified abstraction layer.

The unified model will enable new paradigms in heterogeneous computing where multiple heterogeneous accelerators may be employed to accelerate the same application. From the perspective of the operating system, current heterogeneous systems tend to dedicate accelerators or specialized-architectures to a single application, blocking other users from

accessing them concurrently. Sharing of the accelerators is therefore limited to a per-application basis. In these systems, with the accelerators treated simply as I/O devices to the system, the operating system process scheduler usually does not take into account the computing time an application spent on them, resulting in upsetting the fairness and responsiveness of the system as a whole [FC08, RFC09]. The careful coordination of the user processes within the operating system is required to provide true multi-user support. This is only achievable if the operating system treats the reconfigurable resources as first-class computing resources of the system. This principle is particularly important in tightly coupled FPGA-CPU systems in which multiple mixed-architecture user processes are going to be executing concurrently, with each process possibly spending some of their execution time on non-CPU computing resources. The design and implementation of an *operating system process scheduler* that is fully aware of the mixed-architectural nature of the user processes is presented in Chapter 6.

In the MURAC model, the hardware/software interface implementation is abstracted to enable a *simplified programming model*. This feature allows the software designer to focus on the core application while still being able to leverage the efficient underlying system implementations to achieve desired performance efficiency. In combination with the single-context execution model, this abstraction enables and encourages a system that embodies the UNIX philosophy [KP84], whereby support for modularity and re-usability enables *composability* as opposed to monolithic design. A common machine model such as MURAC allows applications to be more readily *portable* across different systems, which is an essential incentive for developing reusable libraries and infrastructure for this class of heterogeneous computing machine. This portability is facilitated by maintaining an abstraction layer that hides the system-level implementation details from the programming model. Such a common machine model also enables system researchers to *compare and collaborate* on designing future heterogeneous machines with multiple types of integrated architectures as applications can be run across multiple diverse machine configurations.

The ultimate goal of the model is to increase the user's productivity when working with heterogeneous systems. This productivity gain is an important factor in determining the value of any high-performance computing system [ZBA+05, BTL10]. One of the key motivations driving this work is the focus on enabling a simple and familiar environment to both software and hardware designers.

1.4 Research Hypothesis

A unified abstraction model for heterogeneous computers will provide important usability benefits to system and application designers. This abstraction enables a consistent

view of the multiple different architectures to the operating system and user applications. This allows them to focus on achieving their performance and efficiency goals by gaining the benefits of different execution models during runtime without the complex implementation details of the system-level synchronisation and coordination.

1.4.1 Objectives

1. To develop a high-level design methodology that allows easy integration of heterogeneous computing components in a common environment that separates the programming model from the system implementation.
2. To study the theoretical and practical trade-offs of the proposed unified machine model for systems with multiple heterogeneous computing architectures.
3. To demonstrate a real-world system design based on the proposed machine model and implementation using commercial off-the-shelf hardware.
4. To demonstrate real-world application design using the proposed programming model and show that the abstraction provided to the application remains consistent with respect to different underlying system implementations.
5. To study the interaction between the operating system and the underlying machine to efficiently support multi-user computation using this abstraction model.
6. To implement a simple operating system scheduler for a heterogeneous machine that demonstrates a multitasking environment for running mixed-architecture applications.

1.4.2 Thesis Structure

The structure of this work expands on the research objectives that support the hypothesis as described above:

Chapter 1 has introduced the problem domain that drives the research, highlighting several important limitations in the current approaches to heterogeneous system adoption. The core contributions of this work have been highlighted in relation to the attempt at address these limitations. The research hypothesis is stated as a driving factor in the approach presented in this work, along with a breakdown of the specific objectives that have been established to validate this hypothesis.

Chapter 2 presents an overview of the key concepts defining the research topic through a survey of modern heterogeneous system architectures along with the state of the art. This is presented as context to the large body of prior work aimed at addressing aspects of the problem space outlined in Chapter 1. These related works are roughly categorised in terms of either system architecture or language-driven approaches.

Chapter 3 presents the proposed MURAC abstraction model for heterogeneous computing system design. The key concepts of a *unified machine model*, a *unified instruction stream* and a *unified memory space* are presented along with the requirements and role that they play within a MURAC heterogeneous system. Under each of these concepts, the practical factors affecting the efficiency and performance of such systems are highlighted along with approaches to address these issues. Following on from this, the simplified programming model that is enabled by this unified abstraction is presented, with a particular focus on the hardware/software interface as well as the role of the design and synthesis tools.

Chapter 4 presents an implementation of an instruction-accurate functional simulator of a heterogeneous multicore system designed in accordance with the MURAC model. This simulator allows for the execution of mixed-architecture applications containing any number of alternative computational models. Through the design of this simulator, the system-level implementation techniques that are required to support the MURAC abstraction are explored, particularly with regard to the consistent hardware/software interface presented to the user application. The programming model exposed by the MURAC abstraction is used in example applications to demonstrate the portability and re-usability that is gained through this approach.

Chapter 5 demonstrates the feasibility of realizing the idealized MURAC model in an implementation of a soft-processor based System-on-Chip. A walk-through of the design considerations made in mapping the system to the MURAC model is provided. This leads to the implementation of an embedded application specific instruction processor that integrates a fixed accelerator core. The application design shows that the MURAC abstraction is able to be consistently applied across different systems while hiding these low-level system differences from the user. This system is implemented on a commercial off-the-shelf FPGA device which is used to determine the overhead that is required in supporting the MURAC model at the system level. The results show a minimal overhead in hardware resources is required to gain the benefits provided by the abstraction to the programmer.

Chapter 6 investigates the role of the MURAC model in a multitasking operating system on a tightly-coupled CPU+FPGA system. The software based implementation

of the MURAC system-level support demonstrates the emulation of the unified machine model without the need for hardware modifications. The system supports arbitrary execution architectures embedded within the application that are executed on the programmable logic fabric during runtime. The operating system scheduler is extended to become aware of the mixed-architecture nature of the system processes, which is used in an investigation of different scheduling strategies and parameters.

Chapter 7 concludes the thesis, including an overview on the contributions and value of the work. The hypothesis is revisited and explored through a discussion of the objectives set out in this chapter and how they have been satisfied through the theories, designs and implementations carried out in this project.

Chapter 2

Background and Related Work

In this chapter a review of the relevant concepts relating to heterogeneous computing systems will be discussed, presenting the common hardware configuration and design approaches. As an active hot topic of research, there are various promising concepts and systems presented in the literature that address aspects of the the problems encountered with the design and usage of these systems as highlighted in Section 1.2.

These works may be broadly categorised as

- **System-driven** approaches that are focussed on the physical construction of the system to support multiple heterogeneous processing elements.
- **Language/Library-driven** approaches that provide a machine-independent framework for access to the heterogeneous processing elements, with a particular focus on user-experience.

These related and prior works will be explored within these categories, providing a background and context to compare, contrast and inform the approach proposed by the MURAC model presented in this work. Furthermore, existing works related to the runtime support of reconfigurable heterogeneous systems will be summarised to extract the principles and compare to the MURAC-enabled operating system support explored in Chapter 6.

2.1 Heterogeneous computing

Design and implementation of heterogeneous systems requires a combination of general-purpose computing, high-performance computing and embedded computing techniques. While general purpose architectures trade off performance, power and area efficiency for generality and ease of programmability (Figure 2.1), specialized hardware co-processors

and accelerators are designed to reduce power and improve performance for specific classes of applications. These accelerators are suited to match the properties of algorithms within the target domain to provide optimized performance. With the addition of reconfigurable and adaptable architectures into a heterogeneous system, a blend of high flexibility and energy efficiency is achieved. However, a guiding design principle of such a system must be that the performance increase achieved is not negated by the overhead of using these multiple heterogeneous components within a single machine.

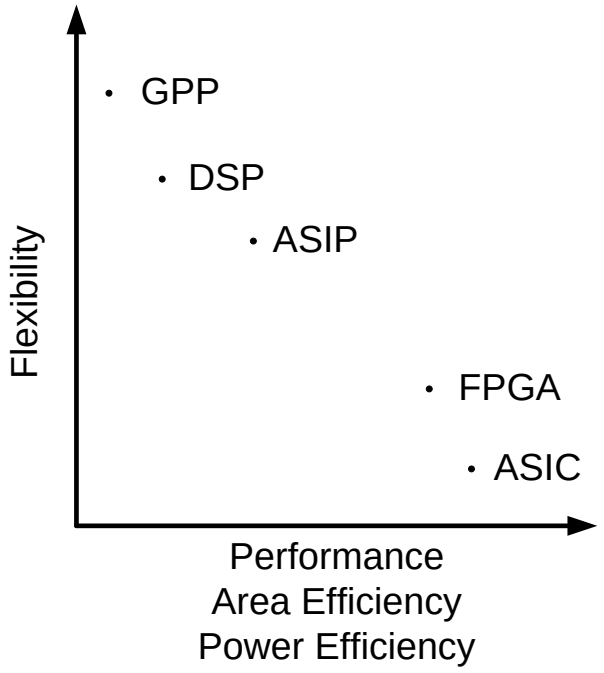

FIGURE 2.1: Trade-off between flexibility and efficiency

2.2 System Architectures

A computer processing element is configured and controlled through its instruction set architecture (ISA). This interface exposes the computational model of the architecture and defines the communication language that is available to the programmer to instruct the machine to carry out operations.

A large number of diverse computing architectures have been proposed and implemented, each with its own strengths and weaknesses. This spectrum varies from being completely general purpose to entirely application specific. The combination of multiple *different* processing architectures within a single machine aims to combine the strengths of each while attempting to minimize the weaknesses. The optimal combinations of architectures is often suited to a class of algorithms and driven by the needs of the application domain. Architectural design choices and techniques are mostly driven by the need for making

systems faster (performance), making systems use less energy (power), and making the hardware smaller (space).

Complex Instruction Set Computer (CISC) architectures are designed to support and execute a large series of operations often varying in runtime cost. An important advantage of this architecture is that the level of the ISA is much closer to that of high level languages, making the job of the compiler much easier. This also leads to shorter programs with reduced space requirements for instruction storage in memory. There is an emphasis on implementing complex tasks directly within the processor hardware.

Reduced Instruction Set Computer (RISC) architectures take the approach of supporting a small subset of fundamental instructions that have predictable short execution times. The advantage of this architecture is that it greatly reduces the hardware complexity and has a much smaller physical footprint. As these instructions are all executed within a single cycle, the use of pipe-lining allows for an increased throughput.

Hybrid approaches that implement micro-operations (μops) on RISC-style processing cores below a CISC instruction set architecture are able to extract the performance benefits of the RISC model while still maintaining the advantages of the CISC design [IJJ09]. In such an architecture it is left to the CISC implementation to efficiently translate from its ISA into the appropriate micro-operations. Processor architecture design may also be optimized for certain classes of application, for example the widely adopted Digital Signal Processors (DSPs) that offer irregular instruction set architectures optimized for the digital signal processing domain [Hea14].

Processor architectures are also able to obtain large performance gains through the exploitation of instruction level parallelism (ILP) [JW89] using heavy pipelining [RL77], out-of-order (OOO) execution and speculative precomputation [CWT$^+$01]. By overlapping instruction execution with the servicing of outstanding cache misses, these techniques are able to hide the memory latency overhead associated with the latter. The system-level approach of superscalar architectures allows for higher single processor throughput by simultaneously executing parallel instruction pipelines. By adding hardware multithreading support to superscalar processors, Simultaneous Multithreading (SMT) [TEL95] architectures allow for multiple independent threads of execution within a single processor core. Alternatively, Very Long Instruction Word (VLIW) processors allow for the specification of simultaneously executing instructions at the ISA level, with the instruction dependency management performed in software by the compiler that is completely aware of the target processor architecture [FFY05, WvAB08].

The current state-of-the-art in processor architecture makes use of the tightly-coupled integration of multiple independent cores on the same silicon die [Vaj11]. This ranges from the popular multicore processors (e.g. Intel i7 and the AMD Phenom II) containing just a few cores, to large scale many-core systems featuring a high numbers of cores (e.g. Intel Larrabee [SCS$^+$08] and Tilera Tile CPU).

Further performance gains have been achieved by the close interaction between the hardware and software layers in application specific instruction set processors (ASIPs) and reconfigurable instruction set processors (RISPs). The ASIP is composed of a hybrid of programmable processor and customized logic as extensions of the instruction set. The applications for such a machine would implement the computationally light parts in the standard instruction set of the processor and the computationally critical parts in a specialized instruction set. These specialized instructions may map to additional functional units within the processor customised for the application. In this way, the processor is able to achieve higher performance than a general purpose processor while reducing power consumption as well as the application code size. Taking this concept further, the class of reconfigurable instruction set processors (RISP) are composed of a processor core that has been extended with reconfigurable logic. These processors are able to provide greater flexibility by allowing for dynamic decoding logic in the control path, supporting hardware specialization for computationally intensive tasks. A detailed design-space exploration and classification of RISPs has been presented by Barat et al. [BLD02], and a comprehensive survey of reconfigurable processors has been undertaken by Chattopadhyay [Cha13]. The design of an ASIP (described in Chapter 5) as well as that of a RISP (described in Chapter 6) using the MURAC model is presented in this work.

Göhringer et al. [GPHB09] have elaborated on the well known Flynn's Taxomony [Fly72] of processing architectures to develop a classification that includes modern reconfigurable multiprocessor Systems-on-Chip (MPSoCs), shown in Figure 2.2. In addition to the categorization of the instruction- and data-streams in terms of parallelism, the dynamic reconfigurability of the instruction (control path) and data path processing elements are also considered in this classification of architectures.

2.2.1 Granularity

Driven by need to support parallelism and concurrency for performance gains, the use of application-specific processors, multi-core and many-core homogeneous and heterogeneous architectures has become increasingly common. The granularity of the parallelism available in these architectures is an important factor to consider in their design and usage. A variety of architectures are available that support different levels of granularity, each suited to specific use-cases. Field-Programmable Gate Arrays (FPGAs) are composed of single-bit wide configurable logic blocks (CLBs) while devices with wide data-paths include well known 32-bit and 64-bit CPUs, Coarse-Grained Reconfigurable Architecture (CGRAs) and the Data-Path Units (DPUs) of reconfigurable DataPath Arrays (rDPA).

Matching the granularity of *reconfigurable* architectures [Vas07] to a target application is an important technique used to obtain an optimal architecture suited to its efficiency

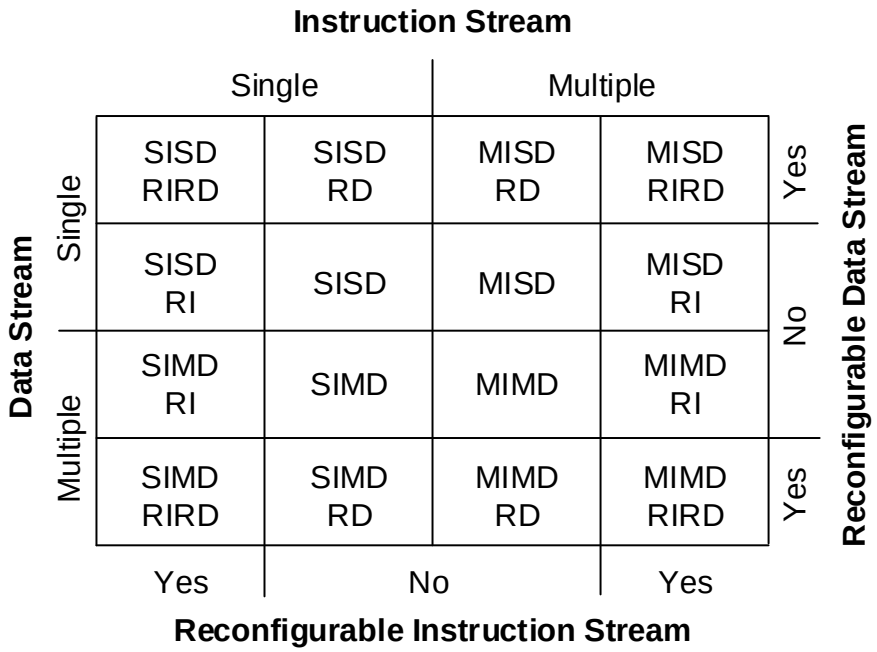

FIGURE 2.2: A Taxonomy of Reconfigurable Single-/Multiprocessor Systems-on-Chip [GPHB09]

requirements. **Coarse-grained architectures** enable high performance with a lower energy consumption. The favourable configuration time due to the small size of their configuration words can lead to improved performance compared to fine-grained, general-purposed programmable logic devices. However, the use of very coarse granularity may also lead to inefficiencies and load imbalance. Coarse-Grained Reconfigurable Arrays (CGRAs) exemplify this architecture, consisting of an array of a large number of function units interconnected by a mesh style network.

In contrast, Field Programmable Gate Arrays (FPGAs) offer a **fine-grained architecture** built from lookup tables (LUTs) and flip-flops, although these devices often contain additional coarse-grained components such as DSP blocks, on-chip memories and even embedded general purpose processors. This provides a very flexible hardware architecture as they may be reconfigured as often as necessary based on the needs of the application. Dynamic and partial reconfiguration [LBM+06] of FPGA devices allows the manipulation of much smaller regions of configuration frames at runtime while the other regions of the logic remain unaffected and can continue to operate. This feature allows the subsets of the hardware resources of the FPGA to be used in a time-multiplexed manner throughout the runtime of an application, but it is critical to ensure logic level compatibility between the static design and the dynamically changing partial regions. By giving up the deep instruction memories found in general purpose processors and DSPs, FPGA devices can achieve a fine-grained controllability and a high computational density advantage [DeH00]. However, the greater potential for performance speed-ups obtained through this finer granularity is opposed by the increased synchronisation and communications overheads.

2.2.2 Reconfigurability

Adaptive systems can expose either dynamic or static reconfigurability to the application allowing for either full or partial device customization. Reconfigurable processors extend this flexibility to traditional processor pipelines, offering customization of either the instruction set architecture or the data-path.

Instruction set architecture (ISA) based approaches aim at targeting the hardware/software interface at the level of the traditional processor instruction processing pipeline. The on-demand dynamic modification of a processor instruction set architecture during runtime provides a key advantage in a number of systems. The Dynamic Instruction Set computer (DISC) [WH95] is composed of a tightly-coupled static reconfigurable GPP employing a CISC instruction set architecture. The use of partial reconfiguration enables DISC to provision instruction modules on demand by using resources only as required, as well as providing the ability to physically relocate these modules within the FPGA. Augmenting the core processor's functionality with new operations enables the Processor Reconfiguration through Instruction-Set Metamorphosis (PRISM) [AS93] general-purpose architecture to speed up computationally intensive tasks. Similarly, FITS [CTM04] provides an instruction synthesis paradigm that replaces fixed instruction decoding units in the processor pipeline with programmable decoders.

These ideas are taken further by incorporating reconfigurable logic into the instruction pipeline. The Adaptive Processor Architecture [HGT+12] includes the internal configuration access port (ICAP) of an FPGA as part of the execution phase in the processor pipeline, which enables dynamic reconfiguration of the reconfigurable logic directly from the instruction stream. OneChip [CC01] provides tightly-coupled reconfigurable units directly within a superscalar RISC processor pipeline. The EXOCHI [WCC+07] system provides an interface for abstracting reconfigurable devices, presenting FPGA accelerators as ISA-based MIMD computational resources within the system. Runtime support through the operating system is used to allow the application to control the hardware accelerator through an extension to OpenMP instrinsic functions in the programming model. The MOLEN polymorphic processor [WGB+04, KG04] features a tightly coupled reconfigurable processing element that supports micro-code primitives at the ISA level, exposing the hardware to the application and allowing combined code and hardware descriptions within the same programming model.

The Convey HC-1 system combines tightly-coupled FPGAs with CPU in a shared memory environment [Bre10] that supports dynamically loadable instruction sets. This system also features hardware implemented reloadable instruction sets, called personalities, which are suited to particular classes of application. Bauer et al. [BSH08] have proposed a modular approach that allows multiple implementation variations for each instruction in the processor. The runtime system is capable of selecting the appropriate variation

of the instruction set architecture, triggered by instructions embedded at application compile-time.

As an alternative to the customization of the instruction decode logic, variable data-path systems shift the focus to customising the processor data-path. Without support for any predefined instructions, the No Instruction Set Computer (NISC) [RGG05] rather maps the entire application directly onto the hardware data-path by generating control signals in the compiler instead of in a hardware decoder. This approach is similar to VLIW architectures that emit wide instruction words every cycle, and enables the exploitation of the horizontal and vertical parallelism in the application. Extending this idea, FlexCore [TSB+09] exposes a dynamically configurable data-path that is created by the compiler and scheduled via micro-code at runtime. The Dynamically Specializing Execution Resources (DySER) [GHN+12] architecture is a coarse-grained architecture integrated into a processor pipeline that enables energy efficient computing by dynamically creating specialized frequently executing regions and applying parallelism.

With the Many-core Approach to Reconfigurable Computing (MARC) [LCD+10], reconfigurable processors are implemented via a template as fully parametrized specialized cores dedicated to either control or algorithmic processing. The MARC system supports both coarse-grain multithreading and dataflow style fine-grain threading while allowing bit-level resource control.

An overview of the challenges and survey of solutions to the design and development of reconfigurable multiprocessor Systems-on-Chip (MPSoCs) has been provided by Göhringer [Göh14].

2.2.3 Platform configuration

The design space for heterogeneous systems is large and involves complex trade-offs that make it important to consider not only the mix of processing elements, but also the memory hierarchy, coherence protocol, and on-chip network [MS11].

A typical configuration of heterogeneous systems is that composed of the combination of a general purpose processor (GPP) with one or more specialized hardware accelerators (HA). These systems are well suited to the combination of control and I/O portions of applications with computationally intensive kernels.

In these GPP-centric machines the interconnection of additional heterogeneous processing elements (HAs) can be realized in multiple ways (Figure 2.3):

(a) **tightly-coupled** with the RISP or ASIP embedded directly within the same die as the GPP core (e.g. Xilinx Zynq),

(b) **loosely-coupled** coprocessors attached to the font-side bus (e.g. Floating Point processors, Nallatech's in-socket FPGA FSB platform [SG10], GARP [CHW00]),

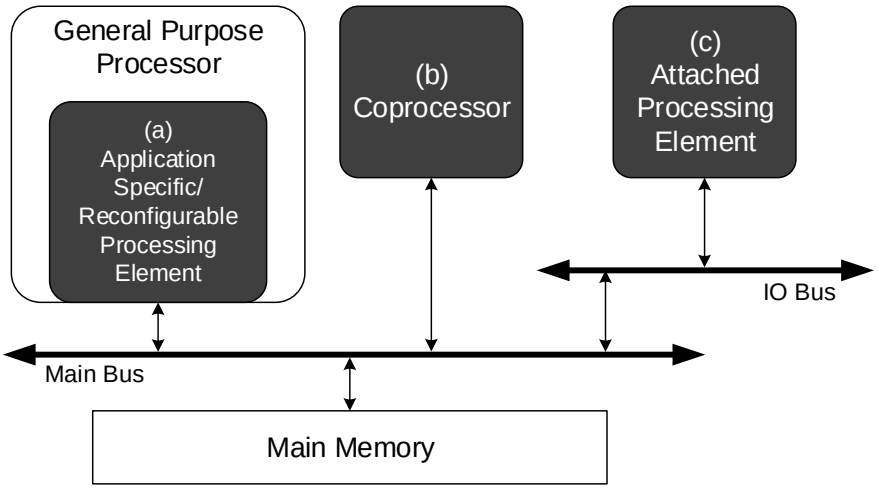

FIGURE 2.3: Heterogeneous processing element coupling

(c) **loosely-coupled** attached processing elements connected via an I/O bus such as PCI (e.g. GPUs, PRISM [AS93]).

In order to address the higher overheads associated with loosely-coupled systems, system designers have increasingly turned towards tightly-coupled architectures that aim at minimising communication latency between the processing elements. For example, while the most widely deployed commercial heterogeneous system currently consists of loosely-coupled CPU with attached GPU, the fusion of GPU and CPU on the same die has been shown to provide much better data transfer performance [DAF11].

2.2.4 Communication

To achieve effective communication, most heterogeneous systems implement the idea of a shared memory region that is accessible from all processing elements. Supporting an efficient, consistent and coherent memory access involves careful system-wide co-ordination and synchronisation.

The communication in heterogeneous systems between the different processing elements consists of both the data transfers that require high-performance high-bandwidth access to main memory, as well as the control information transfers for which low latency is critical but can tolerate a much lower bandwidth. Multiprocessor systems with a large number of processors will tend to have complex interconnection networks for shared memory access, adding latency to requests passing through these networks.

The actual performance impact of the data access contention varies with the application and is caused by exceeding the bandwidth of the shared level of the memory hierarchy. This leads to a reduction in the ability of applications to take advantage of multicore processing computation capabilities in domains like scientific computing. Techniques like

FastLane+ [LK07] attempt to solve this by means of bypassing the normal bus transactions and directly connecting the memory intensive hardware accelerator cores to the main memory controller. Analytical models of these data access bottlenecks in heterogeneous multicore processors have been developed to help developers tweak application performance [BSH09]. It is critical in reconfigurable systems that the arbiter that manages the access to shared memory resources has the ability to handle the dynamically changing processing elements. An example of such a system using a Network-on-Chip is presented by Göhringer et al. [GMHB11].

The problem becomes worse in multi-level hierarchical memory systems, containing combinations of shared and private caches leading to non-uniform memory-access scenarios. In addition to the global shared memory regions, processing elements may also have access to local memories that are either attached (e.g SRAM chips) or integrated (e.g. FPGA distributed and block RAM) into the architecture. Effective memory management becomes more complicated as the overhead in transferring the data between the multiple levels of memory increases and issues of coherence and consistency arise due to several copies of shared data existing within the memory hierarchy. These non-uniform memory access problems often need to be dealt with in the application itself, usually imposing limitations on the performance gains of highly optimized and parallelised algorithms. The scratchpad memory hierarchies proposed in LEAP [AFP$^+$11] provide an abstraction of the the multi-level memory model for FPGA devices, providing a consistent interface to both on-chip and global memory. Techniques such as Transactional Coherence and Consistency (TCC) provide promising results in enabling cache coherency in shared-memory systems by using programmer defined transactions as the fundamental unit of parallel work [Olu05].

Access to a shared memory region in heterogeneous system may be configured in either a master/slave or master/master mode. While the master/slave configuration does not provide the same level of performance as the latter, it avoids the complicated coherence and consistency issues that need to be addressed when each processing element requires master mode access to the shared memory region.

A memory access request from an attached processing element in a master/slave configuration (Figure 2.4 a) will need to (1) signal an interrupt on the general purpose processor, requiring it to (2) interact with the main memory to service the request, and then (3) transfer this resulting data back to the originator. In contrast to this, systems configured with master mode access to memory elements from processing elements (Figure 2.4 b) are able to directly access shared memory, thereby avoiding the overhead in latency and contention. This direct access to a shared memory is often achieved via the use of DMA engines that enable I/O attached devices direct access to memory after initial set up by the general purpose processor.

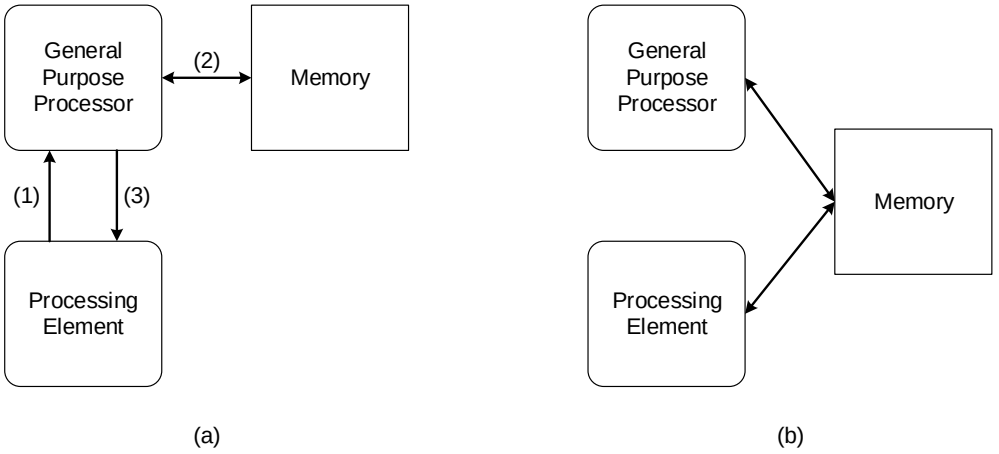

FIGURE 2.4: (a) Master/Slave and (b) Master/Master mode access to a shared memory region

With the virtual memory window [VPI06] mechanism, a hardware accelerator is able to access any region in the main system memory through a shared memory window managed via a virtual memory window manager in the operating system layer. Direct access to a virtual memory space from a hardware accelerator without the need for the involvement of a general purpose process has been demonstrated in the FPGA-centric approach of Ng et al. [NCS13].

As a proposed abstraction for memory access, the Memory Architecture for Reconfigurable Computers [LK00] is a scalable, device-independent memory interface supporting both irregular and regular memory accesses from hardware accelerators. Additionally, this system creates a target environment for automatic hardware compilation by hiding the specific physical memory characteristics and focussing on the semantics of memory accesses.

A variety of interesting mechanisms have been proposed to overcome the issues posed by shared memory regions, such as the type-specific memory coherence of Munin [BCZ90] for distributed shared memory multiprocessors, and the task-centric memory model of Kelm et al. [KJL+10] targeting single-chip multiprocessor with private caches.

As an alternative to the use of shared memory mechanisms for enabling the interaction between processing elements, message passing protocols provide another approach to enabling this communication. These protocols are usually implemented as libraries in the software layer rather than needing hardware implementation and support. TMP-MPI [SPM+08] and SoC-MPI [MLIB08] are two such examples of effective message passing systems targeted at multiprocessor SoCs.

2.3 Languages, frameworks and tools

The hardware/software interface presents one of the biggest challenges in the use of modern heterogeneous computing systems. A large barrier to the adoption of these systems is posed by a lack of widely applicable programming models and techniques that effectively reduce the complexity of programming them. For example, the programming models for both the ASIP and RISP style architectures are tightly tied to the underlying hardware platform. While some systems treat the processing elements as I/O devices that are decoupled from a host GPP, others may treat accelerators as tightly coupled co-processors armed with cache-coherent shared memory access. A majority of the techniques developed to address this challenge are focussed on the creation of programming abstractions that allow developers to easily model the parallelism and communication relationships between the different hardware and software components of an application. In the approaches that aim at supporting full parallelism and concurrency, careful attention is required by the programmer to avoid the common pitfalls such as data-races and synchronization overhead together with inability to easily reason about the flow of the application due to the non-determinism. Additionally, some of these approaches may become burdensome to the class of smaller applications that do not have clear hardware/software boundaries. Although these approaches differ in the scope of their objectives, they all share the common goal of raising the level of abstraction required to design and integrate hardware and software components.

A high-level approach to the development of a bridging model for multicore computing [Val11] is aimed at capturing the most basic resource parameters of multicore architectures. Including the mechanisms for both synchronization as well as computation and communication, a multi-level model is proposed that has explicit parameters for processor numbers, memory/cache sizes, communication costs, and synchronization costs. This approach is guided by the noteworthy goal of focussing efforts on the design of portable algorithms that target the bridging model rather than the considerable efforts required in designing efficient algorithms for individual architectures and systems. In a similar manner, the MURAC model aims to influence the future design of both software and hardware heterogeneous systems.

System-level modelling languages like Ptolemy [BHLM94] and Rosetta [AK01] enable the specifications of capabilities in heterogeneous systems that are then able to drive simulation, software compilation and hardware synthesis. Chattopadhyay et al. [CLMA08] introduce a language-driven exploration of ASIPs that models the partially reconfigurable processors via a high-level Architecture Description Language (ADL). Traditional techniques for instruction level parallelism have been successfully extended to reconfigurable machines [CW98]. Burgio et. al [BMCB14] have introduced an integrated high-level synthesis-based toolflow for design space exploration of heterogeneous many-core

embedded platforms, providing an abstraction to exploit accelerators from within an OpenMP application.

Numerous projects focus on lifting the level of abstraction for reconfigurable logic programming from that of hardware descriptions of gate-level parallelism to that of a modified and augmented C syntax - so called High Level Synthesis (HLS). Along with the current state-of-the-art tools provided by the major FPGA vendors - Xilinx Vivado High-Level Synthesis (HLS) and Altera SDK for OpenCL - numerous attempts at adapting high-level software programming languages (typically C or C++) to hardware design have provided tools and frameworks with varying success. The quality of the hardware generated by these tools strongly depends on the structure of the high-level language code. It is indispensable for the designer to have a good understanding of the target architecture to achieve favourable performance and hardware resource usage. Numerous Ansi-C to hardware description language compilers are available such as SPARK [GDGN03], ROCCC [VPNH10], DWARV [VYB07] and LegUp[CCA+13]. The ASH (Application-Specific Hardware) implementation of Spatial Computation architecture [BVCG04] includes a compiler that transforms an input C program into a hardware dataflow machine equivalent in Verilog. Handel-C [Agi09] is an extension to the C programming language that provides explicit parallelism, hardware data types and inter-thread communication channels based on the model of Communicating Sequential Processes (CSP). Based on the idea of automatic compilation such as those shown in GarpCC [CHW00] and Nimble [Mac01], Comrade [GK07] compiles imperative C code into combined hardware/software applications targeted for adaptive computers. Warp [VSL08] supports the automatic and transparent transformation of executing CPU binaries into FPGA circuits.

Domain specific languages provide custom syntax and semantics for expressing applications for heterogeneous architectures. Lime [ABCR10] is a Java-Compatible and synthesizable Language for Heterogeneous Architectures. An approach to providing a framework for creating DSLs that are able to map efficiently to target devices can be found in the Delite [SBL+14]. The Variable Instruction Set Communication (VISC) Architecture [LCKR03] allows a compiler to select from a dictionary of instruction sets that best suit the program. This is achieved by optimizing for an abstract representation of the code and enumerating instruction scheduling to determine an optimal ISA for the target architecture.

Cell superscalar (CellSs) [BPBL06] provides a programming model for the Cell Broadband Engine architecture, a system that combines a general purpose processor core with streamlined co-processing elements [GHF+06]. The automatic exploitation of the parallelism in a sequential program at the functional level through the mapping to the different co-processing elements is supported. This is achieved through the simple annotation of the source code, enabling the runtime to build a task dependency graph

that is used for locality-aware task scheduling and data handling between the processing elements.

Other approaches aim at creating a standard interface for heterogeneous computing systems. RIFFA [JFK12, JK13] aims to create a reusable integration framework for FPGA accelerators for traditional software environments by providing communication and synchronization for FPGA accelerated software using a standard interface. The Open Component Portability Infrastructure (OpenCPI) [SI09] aims to provide an open source set of tools by providing a real-time embedded middle-ware framework aiming to simplify programming of heterogeneous processing applications. The Reconfigurable data-stream hardware software architecture (Redsharc) [KSS+12] provides an abstract API for the development of simultaneously executing hardware and software kernels with a seamless communication interface supported by on-chip network. The Latency-insensitive Environment for Application Programming (LEAP) [FYAE14] FPGA operating system is designed to provide a set of portable latency-insensitive abstraction layers. It provides an extensible interface for management of heterogeneous computing resources at compile-time.

A number of tools provide an automatic hardware/software interface specification and generation. The Balboa [DSGO02] hardware/software co-design framework abstracts the IP interfaces into domain specific languages that provide automatic data type matching and interface generation. The Parametric C Interface For IP Cores (PACiFIC) [LK04] allows for the automatic embedding of complex IP cores in a high-level language by presenting imperative function interfaces for hardware cores to the software programmer, hiding the formal hardware descriptions and platform behaviours.

With a particular focus on the communication between processing elements, TMP-MPI [SPM+08] extends the standardized and portable Message Passing Interface (MPI) to multiprocessor Systems-on-Chip implemented in FPGAs with the aim to provide a standard API with a common programming model for high performance reconfigurable computers. The need for a simple and portable communication and synchronization interface has inspired the proposal of the Simple Interface for Reconfigurable Computing (SIRC) [Egu10], which offers an extensible reconfigurable computing communication API. The main benefit of such an API is that it enables the hardware and software interfaces to remain consistent across different platforms and versions of the machine. A system based on the MURAC model will lead to a trivial implementation of such an API by taking advantage of the unified abstraction.

Programming models that are coupled with corresponding customised compiler support and runtime frameworks provide powerful ecosystems for more complete heterogeneous design paradigms. Lange and Koch [LK10] extend the Comrade system to present an execution model and efficient realization of a system that orchestrates the fine-grained interaction of a general purpose processor and a reconfigurable accelerator that has

full master-mode access to memory. Brandon et al. [BSG10] have presented a generic, platform-independent approach for exploiting reconfigurable acceleration in a general purpose machine. This system requires an FPGA device driver, reconfigurable wrapper in the FPGA and an extension to a standard compiler that directly inserts system calls to the code for the control of the reconfigurable accelerator. This technique obviates the need to extend the ISA of the target system. The Merge Framework [LCWM08] is a general purpose programming model for heterogeneous multicore systems that takes a library-based approach. A modified compiler is used to provide a language with map-reduce semantics by dynamically selecting the best available function implementations for a given input and machine configuration. Tools like Harmony [DY08] provide automatic identification and off-loading of compute kernels into hardware accelerators and include sophisticated runtime support for management and scheduling. In this paradigm programs are specified as a sequence of kernels and control decisions expressed in a control-flow graph form.

2.4 Runtime support

2.4.1 Multitasking

Hybrid multithreading techniques aim at extending the well-known POSIX threading model for concurrent programming into reconfigurable and heterogeneous systems. This is achieved by providing the runtime management of the threads across the underlying processing elements, along with a programming model and libraries necessary for integration into heterogeneous applications. HybridThreads [APA+06, PAA+06] extends these techniques by providing synchronization primitives and run-time scheduling services with the ability to perform thread migration for both hardware and software threads into hardware. ReMAP [WA10] implements the multithreading approach within a shared reconfigurable architecture that features re-use of reconfigurable fabrics among threads and the integration of custom computation within the communication. These features result in large performance gains over systems with dedicated hardware communication. Task Superscalar [ECR+10] demonstrates a system that can run multiple tasks on different accelerators at the same time using out-of-order execution techniques.

The MOLEN [WGB+04] processor and tools have been adapted to support multi application, multitasking scenarios [SB10]. The model processor controls a reconfigurable accelerator via an ISA extension, and achieves communication of function parameters and results between the processor and a reconfigurable accelerator through a set of exchange registers.

Various techniques have been proposed to handle context switching and hardware preemption on programmable logic devices. Rupnow et. al. demonstrate three execution

policies for hardware multitasking [RFC09], *block*, *drop* and *rollback*, applied at each software thread context switch interval. The system requires software implementations of hardware kernels to be used to avoid stalling in the case that hardware resources are not available when they are needed. This system also requires applications to register hardware kernels with the operating system at application load time, with a scheduler periodically allocating reconfigurable resources. A reconfiguration-based hardware context-switching method for dynamic partially reconfigurable systems using a database in memory to save and restore the significant frame and register data is presented by Lee et al. [LHLT10]. Koch et al. [KHT07] extend the concept of software checkpointing to the hardware in order to save the state of the hardware task at defined intervals in a multitasking environment, in contrast to a readback-based approach. A comprehensive comparison of pre-emption schemes for partially reconfigurable FPGAs has been presented by Jozwik et al. [JTE+12]

From the perspective of the operating system, not much work has been devoted to the study of scheduling processes that execute with a mixture of computing architectures during runtime, such as those on a tightly-coupled CPU and FPGA system. Bower [BSC08] has shown that power efficiency and performance are compromised unless operating system schedulers consider dynamic heterogeneity on dynamically heterogeneous multicore processors. Pham et. al [PJC+13] propose the use of a microkernel-based hypervisor for virtualized execution and management of software and hardware tasks running on a commercial hybrid computing platform. Dittmann et al. [DF07] described a pre-emptive scheduling approach based on a deadline monotonic algorithm for a single processor and several runtime reconfigurable accelerators. The Performance Aware Task Scheduler (PATS) [BGSH12] for runtime reconfigurable processors considers the state of the reconfigurable processor and the task efficiency in determining task scheduling during runtime.

In contrast to these approaches, the MURAC model rather views the full reconfigurable hardware region as a possible execution architecture for running applications, thereby allowing the software scheduler to manage the resource utilization. The single-context process view of the MURAC model allows for a simple extension of a CPU scheduler to support heterogeneous applications within consistent operating system framework, described further in Chapter 6.

2.4.2 Operating Systems

A heterogeneous computing system must consider both *hardware* and *software* architectures, wherein all the different processing elements should be integrated efficiently within the operating system to allow for sharing of resources between all the programs running on the machine.

HybridOS [KL08] provides simple interfaces for both hardware and software design for reconfigurable accelerators. It uses a library-call approach with an accelerator framework integrated into a linux kernel to address application integration, data movement and communication overhead. Application specific accelerators are implemented in the FPGA and allocated to user application in the operating system.

The ReconOS [LP07, AHK+14] operating system for reconfigurable computing supports the runtime execution of both pure software and reconfigurable hardware threads in a unified environment using a fixed-priority scheduling algorithm. These hardware threads are provided with standard concurrency primitives. This system has been further extended to support hardware thread relocation by using a module placer to compute feasible placement locations during runtime [WAT14].

The Berkeley Operating system for ReProgrammable Hardware (BORPH) [STB06] is a general purpose, multi-user operating system for FPGA-based reconfigurable computers. BORPH provides a portable abstraction to underlying reconfigurable hardware at the operating system level that is machine and language independent. The familiar UNIX semantics of a process are extended to the execution of gateware on a reconfigurable device, with data transfer abstracted in a file-system interface. BORPH provides a useful abstraction of the underlying reconfigurable architecture as additional computational resources to the linux kernel. Building on top of the same guiding principles as the BORPH operating system, MURAC has a similar goal of eliminating the boundary between software and hardware components in a heterogeneous machine.

2.5 Chapter Summary

This chapter has presented the relevant background and prior approaches to heterogeneous system design, with a survey of the numerous techniques and mechanisms available for addressing the challenges of efficient design and implementation of these systems. The study of these works has informed the design choices and exploration leading to the proposal of a system-level approach for heterogeneous machine design in the MURAC model that aims for simplicity by providing a unified computing model with consistent view of the machine to the users.

Chapter 3

The MURAC model

The Multiple Runtime-Reconfigurable Architecture Computer (MURAC) is presented as a system design and implementation model that allows for the easy integration of heterogeneous computing components in a common environment. Through the separation of the programming model from the system implementation, MURAC enables applications to leverage support for multiple distinct runtime computing architectures co-existing within the same machine. A primary goal of the MURAC abstraction is to provide a consistent interface for migrating between these diverse architectures as needed at runtime. The three key concepts that make up the MURAC model are a *unified machine model*, a *unified instruction stream* and a *unified memory space*.

3.1 Unified Machine Model

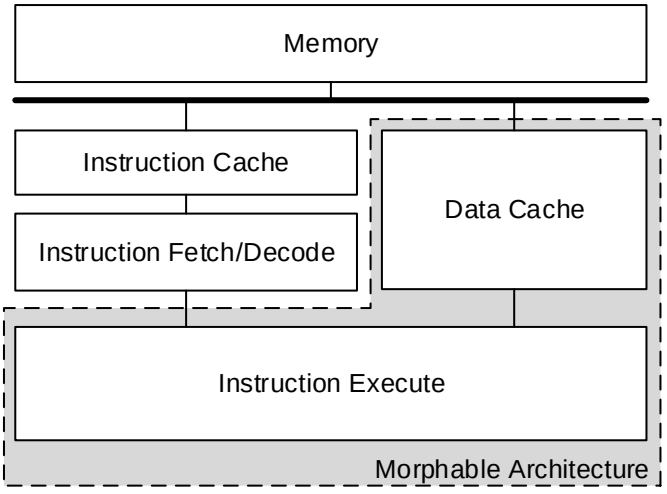

FIGURE 3.1: Ideal MURAC processor with morphable architecture

MURAC models systems composed of multiple heterogeneous processing elements as a single idealized machine with a morphable compute architecture as illustrated in Figure 3.1. This machine resembles the idea of a variable data-path system, where the instruction execution hardware is able to support alternative computing models dynamically mapped to an appropriate underlying hardware architecture. The abstraction maintains a view of a heterogeneous machine as a time-multiplexed homogeneous machine - at any point in time the machine presents a single compute architecture to the running application. Furthermore, the definition of a machine instruction is generalised and thus a single unified instruction stream controls the main flow of a user program regardless of the underlying compute architecture.

To support this abstraction, MURAC specifies a simple generic extension at the instruction set architecture (ISA) level that allows the machine to morph during runtime, enabling an application to choose to execute portions or all of its program instructions in any one of the architectures at a time. By accommodating a wide variety of heterogeneous architectures at this system-level, any application running on a MURAC machine may take advantage of such features even in the absence of any operating system or runtime library support. The exact mechanisms used to morph the architecture are isolated from the user through this abstraction model and are left as optimisation goals for low-level system engineers. This allows users - application developers, the compiler, or the operating system - to focus exclusively on the core benefits of diverse modes of computation without being overburdened with system and vendor specific details and complicated orchestration and synchronisation mechanisms.

Within this model the specialized application-specific processing elements found in heterogeneous machines are no longer treated merely as accelerators or coprocessors. They are now equally modelled as alternative computational architectures that a user program may choose to morph the machine into during execution. These alternative architectures are then able to be invoked passively as subroutines within a program, or may be used for independent portions of dataflow computations requiring only input and output queues.

3.1.1 Computational architectures

The default architecture that the machine boots into is termed the **Primary Architecture** (PA). Any other computing architecture that the heterogeneous machine also supports is termed an **Auxiliary Architecture** (AA). There may be multiple different types of auxiliary architectures in the system that are characterised by their *mode of computation* rather than by the number of alternative processing elements in the system. At a single point in time, the machine may exist in exactly one type of processing architecture, either as the primary or one of the auxiliary architectures. Figure 3.2 illustrates an example platform configuration of a MURAC system containing a multicore CPU

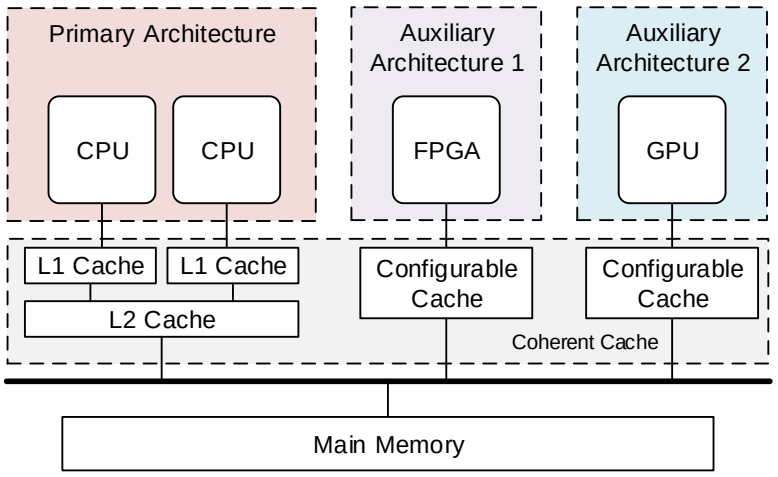

FIGURE 3.2: Example heterogeneous MURAC system

acting as the primary architecture, with alternate auxiliary architectures provided by the GPU and FPGA.

To support morphing between distinct architectures - an architecture branch operation - MURAC defines two mechanisms on the level of the Instruction Set Architecture (ISA):

- a *Branch-to-Auxiliary-Architecture* (`baa`) operation that instructs the machine to change the mode of computation by morphing into a particular *auxiliary architecture*.
- a corresponding *Return-to-Primary-Architecture* (`rpa`) operation that returns to the original mode of computation of the *primary architecture* from the active *auxiliary architecture*.

The low-level implementation of these operations is a system specific task left to the system designer. Typically, the `baa` and `rpa` operations would be implemented in the ISA of the primary and auxiliary architectures respectively. As an example, an ASIP design would benefit from the MURAC model by mapping the base ISA as the primary architecture and the custom instruction set as auxiliary architectures. Similarly, RISP systems would view the dynamically changing custom instructions as auxiliary architectures. Implementing the MURAC operations for performing an architectural branch at the hardware level within these processors would enable the application and users to take advantage of the benefits provided by the programming model described in Section 3.4.

When mapping the model to a fixed-architecture ASIP that physically contains more than one type of auxiliary architecture *at any single point in time*, the programming model and compilation tools will require an awareness of the physical machine implementation to correctly generate the appropriate instructions for morphing the machine into the desired architecture. In addition to this, system-level support will be required

for recognising and provisioning the corresponding underlying processing element at run-time. As an example, this capability could be achieved through the implementation of a specialized functional unit that activates the required processing element based on the instructions in the processor pipeline. In contrast, an adaptable system containing reconfigurable logic will be able to be reconfigured into any desired auxiliary architecture without such requirements because the full information for any possible target auxiliary architecture may be embedded directly within the instruction stream itself.

3.1.2 Practical implementation considerations

The idea of a perfectly morphing processor must be mapped onto the implementation platform while still maintaining the required abstraction layer that enables the MURAC programming model. This morphing behaviour of a processing element must be emulated, for instance, either by provisioning architectures that are available in a fixed-architecture ASIP system, or through the implementation of reconfigurable instruction computing systems. As a design optimization, time-multiplexing of the different underlying architectures leads to efficient resource usage of the system as multiple applications can utilise different processing elements simultaneously, assuming that the overheads do not negate the gains.

In a true MURAC style embedded machine only one architecture would be active at any point in time, reducing the dynamic power consumption of the system as a whole. Any processing element that is not actively performing program execution should no longer be running. This can be achieved by suspending the processor into idle/wait state and then resuming upon the triggering of the architecture branch operations. In practice, the implementation of this idea may be limited by requirements for the concurrent execution of more than one processing element. For example, in a general purpose processor-based accelerator system the operating system may continue to run on the CPU while the application is executing on the attached accelerator processing element acting as an auxiliary architecture. The consistent view of the underlying machine to the running application is an important capability must be supported in this scenario.

Challenges to the system performance are found in the long configuration time for the processing elements like FPGAs and the GPU. Particularly in reconfigurable systems, the long reconfiguration times need to be amortised. As devices grow larger to accommodate more resources, this time only increases. Making use of partial reconfigurability can allow for much faster configuration times as the level of granularity is reduced from the FPGA-fabric as a whole to groups of individual frames. In the MURAC model, the configuration process is modelled as simple instruction loading. As a result, the execution of the code on the FPGA or GPU may not commence until the configuration is completed. A coordinated approach between the compiler and operating system may

choose to perform instruction pre-fetching before the configuration is needed, conceptually similar to speculative pre-loading [YERJ99]. Alternatively, if the host processor is indeed available in a multi-tasking system, the operating system may schedule another available process for execution on the CPU fabric while waiting for such long instruction load. This is similar in principle to the simultaneous multithreading (SMT) approaches used in modern multicore processors. The system designer may also choose to incorporate configuration pre-fetching techniques [Hau98, LCH00, LH02]

3.2 Unified Instruction Stream

A guiding principle behind the design of a MURAC system is the efficient application of computational granularity. Applications are able to easily combine multiple modes of computation into a single program through an abstraction that hides the implementation details.

The MURAC model adopts the broad definition of an instruction to represent any entity that configures the machine to carry out proper computation. A CPU machine instruction, for example, configures components such as the register file and the arithmetic & logic unit (ALU) to carry out an operation. Using this definition, an FPGA configuration bitfile can be treated as an ultra-long instruction word, similar to that proposed by DeHon [DeH96]. Similarly, within a GPU-based auxiliary architecture a binary GPU kernel configuration would be mapped to a single instruction, for instance the CUDA [CUD07] compiled binary kernel *cubin* file used with Nvidia GPUs.

Each instruction in a MURAC processor represents an atomic unit in program execution that maintains a consistent observable machine state. This definition is essential for establishing a program order for an application that utilises various heterogeneous architectures. The definition of the program order, in turn, is invaluable in defining the memory consistency model when the MURAC model is implemented into a multiprocessing context.

With this extended definition of an instruction, the MURAC model essentially extends a computer to behave as if it were a complex instruction set computing (CISC) machine. For example, the execution of part of an application on a reconfigurable fabric can be viewed as the execution of a complex instruction in a CISC machine, except that this instruction may be significantly more complex in a MURAC processor. From this perspective a machine based on the MURAC model may be considered as belonging to a class of very complex and varying instruction set computing (VISC) machines.

With this broad definition, the execution of a program is controlled by a single stream of instructions regardless of the underlying computing architecture. Treating an FPGA or GPU configuration as instructions moves this configuration data into the instruction

memory space rather than the data memory space. The task of fetching, caching, and decoding these instructions can thus be handled consistently by the instruction fetching and dispatching unit of a processor.

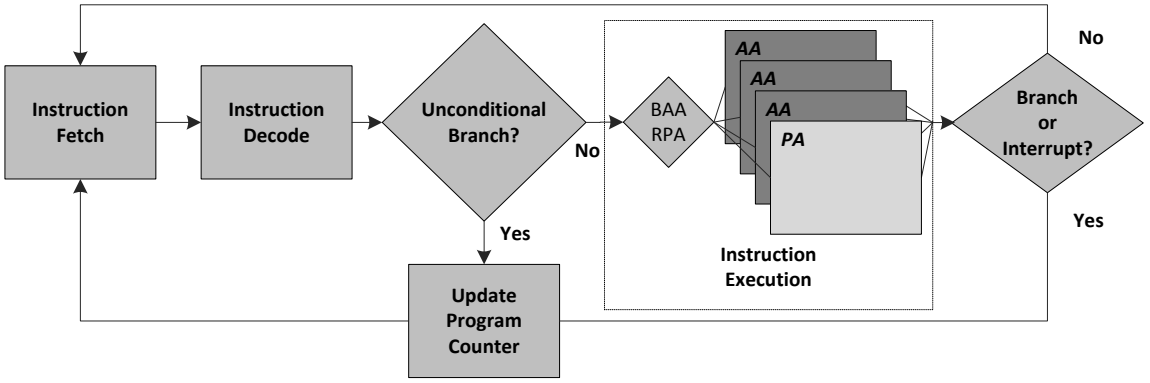

FIGURE 3.3: Conceptual instruction pipeline with support for morphing executing stages.

Figure 3.3 depicts a conceptual instruction pipeline that consists of a morphable instruction execution stage, corresponding to the idealized MURAC processor in Figure 3.1. As part of the mechanism responsible for morphing the architecture into different auxiliary architectures, the instruction fetch unit is controlled by the *Instruction Fetch Frequency (IF)* and *Instruction Fetch Width (IW)*. A simple 32-bit RISC processor, for example, may fetch a new instruction of a fixed width ($IW = 32$) on each cycle ($IF = 1$). The instruction pipeline characteristics of a FPGA will have an extremely small instruction fetch frequency ($IF << 1$) specific to the application design, along with a large width ($IW >$ 1Mb) equal to size of the FPGA configuration bitfile. The lengthy FPGA configuration process may now be easily modelled as an instruction pre-fetch process that can be triggered either by the user via operating system calls, or by a hardware speculative loading unit. Additionally, management of the FPGA bitstream may now be handled cleanly by the instruction cache subsystem instead of by the data cache.

Runtime scheduling and process management within an operating system environment becomes significantly easier to support by supporting a single unified mixed-architecture binary that represents the full application.

3.2.1 Practical implementation considerations

The functional units of the target processor pipeline that are involved in the instruction fetch and dispatch stages will need to be augmented to support the concept of a unified instruction stream. This will lead to a variable-width instruction set architecture capability that can be achieved in a variety of ways. The most obvious of these is the direct modification of existing soft-processor core. The use of freely available open

source processor cores allow for an easier direct modification of the pipeline. An alternative approach to the direct modification of the pipeline is to integrate a tightly-coupled specialized instruction fetch/dispatch unit for handling the variable length instruction fetch, dispatch and provision of the auxiliary architectures directly from the application instruction stream. By combining such a unit with a processor core in a System-on-Chip (SoC), practical support for MURAC can be achieved at the system-level.

Without the ability to augment the physical processor with the required logic to support this unified instruction stream, a runtime emulation layer implemented in software will be capable of supporting the necessary mechanisms. This emulation layer would need to sit at the level of the operating system kernel, with direct access to the underlying hardware. An operating system kernel driver is a natural choice as a means of realizing this emulation layer that meets these requirements. Chapter 6 describes a system that has implemented this strategy by hooking into the interrupt trap for an illegal instruction in the operating system. In the case that a full operating system is not required, the emulation layer could alternatively be implemented as a light runtime layer for bare-metal programs on an embedded system. These emulation techniques would incur a performance penalty due to the extra work required on the CPU in carrying out the MURAC operations in software rather than hardware.

The decoupled programming model allows for a simplified application partitioning between hardware and software. It is at these boundaries that particular care needs to be taken by the system implementer to adhere to the consistency model [Lam79]. For example, when executing the architectural branching operations, the system would need to ensure that any outstanding memory operations are performed before activation of the program on the target architecture. These mechanism can be enforced on the hardware level, or by the compiler or operating system on the software level through the use of synchronisation mechanisms like memory barriers.

The impact of the architectural branch operations on the functionality of the processor pipeline also needs to be considered. Modern processors use a range of techniques such as branch-prediction, speculative pre-loading and out-of-order execution to achieve high throughput by hiding the memory latency. The operation of these mechanisms would be disrupted by an ignorant implementation of the MURAC model, possibly leading to critical delays and poor performance. Additionally, as the instruction stream is now used to hold the possibly large configuration data for the auxiliary architectures, the impact on the instruction cache may be significant if not properly considered in the design of the system. However, techniques such a instruction prefetching and cache-prefilling can be adapted to mitigate some of these issues.

3.3 Unified Memory Space

Logically, a MURAC processor contains only a single unified virtual memory address space, regardless of the number of auxiliary architectures defined. This is a natural consequence arising from modelling the alternative computing architectures as a transformation of the primary computer architecture. In practice, the decision on whether the memory address space is mapped to the same physical memory is a trade-off that system engineers may exploit for sake of power or performance optimisation.

An implication of such memory model on the actual implementation is that every auxiliary architecture implementation must have autonomous access to the main memory system. For instance, if an FPGA is used within the system to implement an auxiliary architecture of a MURAC machine, then this FPGA must be able to retrieve and write data in the shared memory system autonomously. Furthermore, in a true multitasking environment, this FPGA must also provide full virtual memory support to the executing application gateware.

The significance of this unified memory address space is that it enables computer architects to define concrete semantics for the role of memory in alternative processing elements. In particular, it forces system designers to differentiate between accesses to memory locations that are entirely *local* to the processing element, such as caches and embedded memory, and accesses to memory locations in the overall main memory address space. The handling of memory access in the first case is relatively straightforward. They are identical to many conventional accelerator systems where external memory modules are connected to an accelerator and are used exclusively for the computation on this architecture. Access to on-chip memory within an FPGA is an example of such local memory access as content is never required beyond the FPGA itself in other processing elements. However, memory accesses that belong to the second case must be handled carefully. In particular, the underlying hardware implementation must ensure that the shared memory content is consistent according to program order, especially for memory accesses that occur across the architectural boundaries of the `baa` and `rpa` operations.

3.3.1 Practical implementation considerations

A unified virtual memory space presented to the application is required to support the MURAC model. It is essential for the implementation platform to support the consistent view and independent master-mode access to this shared memory from all processing elements. The widespread use of direct memory access (DMA) engines enables the autonomous master access to the shared memory, avoiding the performance costs incurred by involving the host processor. Another advantage to supporting master-mode access comes in the performance gained through utilizing burst-mode access to memory.

The efficient integration of a hardware accelerator that is capable of autonomous master-mode access to main memory within an operating system environment supporting virtual memory remains a challenge. A memory management unit (MMU) is typically used to translate the virtual addresses into physical bus addresses for the host processor. This provides transparent address translation and handling of page faults to the software running on this processor. Most heterogeneous systems, particularly those that are loosely-coupled, do not support access to this MMU from the attached accelerators. Although they may have fast memory access via DMA engines, this lack of address translation and page fault handling on the accelerator leads to numerous difficulties in the operation of the shared memory environment, particularly within an application that needs to share data between the hardware and software partitions.

Shared resources, such as the memory hierarchy within a system, become a limiting factor to performance due to contention and limited bandwidth. To achieve optimal performance, data needed by a processor should be as close as possible to the processor in the memory hierarchy. Stalling to wait for any required data leads to degraded performance, although modern processors are somewhat effective in hiding this latency by context switching to another process. Multiple levels of cache memories exist in modern systems to bridge the gap between speed of processing elements and memory. These caches take advantage of the locality of memory reference over time and space, which allows it to handle a large percentage of memory requests. Extending these principles to provide efficient sharing of data in a heterogeneous system-on-chip consisting of multiple distinct computational processors is a challenging undertaking. In particular, ensuring the consistency of the shared data that is stored in multiple local caches is of primary concern.

In order to address these problems, the introduction of cache-coherence protocols [PP84] is required. Modern tightly-coupled reconfigurable devices such as the Xilinx Zynq Platform provide hardware cache-coherence on a high-speed bus through the so-called Accelerator Coherency Port (ACP), enabling master-mode access to on-chip memory from the programmable logic with automatic cache-invalidation of the processor subsystem caches.

3.4 Programming model

A simplified and portable programming model is made possible due to the consistent interface that hides the low-level system specific details of the underlying platform. This programming model provides implementation-independent access to auxiliary architectures at runtime. The configuration and co-ordination of control of a slave accelerator device that would normally be required in a heterogeneous system is removed from the

programming model, leaving a simple explicit application control flow path that is easier to reason about. The application is written as consecutive sequences of compute kernels each having a different computing model that can include a custom ISA and supporting architecture. This design encourages the efficient exploitation of parallelism through a single-threaded concurrency model. Rather than providing a mechanism for specifying parallelism or concurrency, the model exposes all the implicit parallelism - the parallelism inherent in the computational model in which each part of the program is expressed - to be exploited by mutating the execution architecture.

Figure 3.4 illustrates the benefit of this approach through a comparison with the programming and execution model of traditional accelerator-based heterogeneous systems. Figure 3.4(a) depicts a typical programming model that utilises accelerators for computation. To make use of the attached FPGA accelerator, the application must first open the device driver associated with the FPGA, read the corresponding configuration file from the data memory and program the FPGA accordingly. Once the FPGA is configured, the main software thread must remain active to monitor the execution of the FPGA, perform tasks such as transferring additional data to the FPGA and reading the output data, or simply determine if it should terminate the use of FPGA. As such, the main program control is implicitly split at the point when the FPGA is utilised, creating complex synchronisation problem between the FPGA and the main controlling software on top of the fine-grain parallel execution within the FPGA.

In contrast to this approach, Figure 3.4(b) demonstrates the programming model of MURAC for the same system configuration. Regardless of the underlying architecture in use at any point in time, there is always a single main flow of program execution. The use of the FPGA is simply modelled as a mutation of the processor into an FPGA-like fabric. The main thread of control continues to execute on this new architecture, under the control of the same top-level instruction stream. The configuration of the FPGA is handled by the underlying machine while the synchronisation between the software and the FPGA for data transport is eliminated as the FPGA takes exclusive control of the machine and data is accessed directly from the shared memory region.

The MURAC model makes the control flow of heterogeneous processing element utilisation explicit to the application. For instance, applications must explicitly spawn a separate thread of control to take advantage of computing concurrently on *both* the host CPU and the FPGA. The operating system or the underlying hardware abstraction layer then becomes responsible for scheduling the spawned thread for concurrent execution.

The programming model is program language and operating system agnostic due to the abstraction at the ISA level. While a system programming language like Ansi-C is a popular choice for the implementation of heterogeneous programs, the MURAC programming model is not limited to this paradigm and may be supported by any language, compiler or framework that is capable of emitting the `baa` or `rpa` instructions.

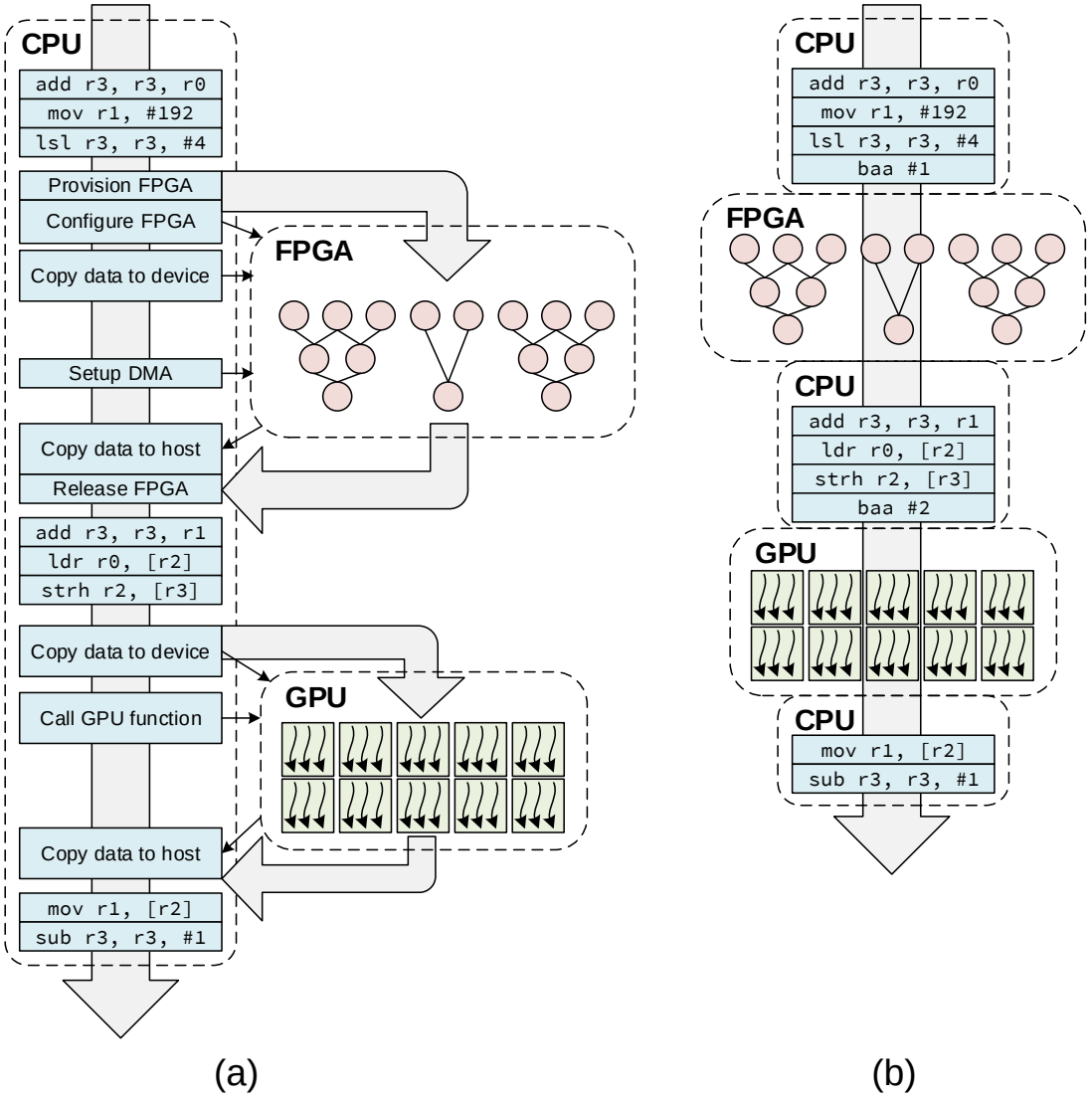

FIGURE 3.4: Application execution under the traditional accelerator model (a) forks implicitly when an accelerator is used. It remains a single stream of control in the MURAC model (b).

Applications created for systems designed using the MURAC model are able to leverage the optimised system-level micro-architecture supporting the unified abstraction, gaining performance benefits that would normally require significant, and often non-reusable, development efforts.

The MURAC programming model is suited for the common heterogeneous application profile in which the control logic is implemented on a general purpose processor acting as the primary architecture, with computational kernels offloaded to auxiliary architectures throughout the life-cycle of the application. The task of hardware/software partitioning in the designation of portions of an application to the primary and auxiliary architectures remains the domain of the programmer. However, the underlying co-ordination

and communication is now standardised and consistent. One benefit of the unified machine model is that it simplifies many traditional hardware/software co-design problems by removing the master-slave relationship that is normally required between the host processor and attached accelerators.

3.4.1 Hardware/Software Interface

From the point of view of the software, the underlying machine should always exist in a single consistent state presenting an homogeneous computing architecture. As the machine is presented as a morphable architecture, the hardware/software interface is captured in the mutation between the available architectures in the system. This provides a course-grained approach that eliminates the complexities of concurrent programming in a heterogeneous environment. The single-context thread of control exposes the application flow between multiple diverse computational modules explicitly to the application developer.

By decoupling the hardware/software interface from the programming model, the individual partitions of the application are able to be designed and optimised independently. The hardware/software partitioning task will thus consist of separating the program into these computation kernels, differentiated by the computational model that each of them supports. These partitions of the application are then temporally linked together via the `baa` and `rpa` operations. As the control flow between computational models is explicit, an imperative approach is suited to support the deterministic sequencing of execution between architecture boundaries. However, within any single partition exhibiting a consistent model of computation, other paradigms such as functional, event-driven or asynchronous designs etc. may be used and possibly preferred.

From the system perspective the view of processing elements communicating over space has now been transformed into one in which these processing elements are communicating over time. This communication is achieved through the unified memory space.

Within the application, a consistent calling convention is required between the partitions of the application. In particular, the interface needs to encapsulate:

(a) The location of parameters and return values in the shared memory space.

(b) The order in which actual arguments for formal parameters are passed.

(c) How the return value is delivered from the auxiliary architecture back to the primary architecture.

A stronger form of this calling convention may be implemented by including support for the enforcement of the expectations between the caller and the callee, similar to the

concept of design-by-contract [Mey92, Bol04]. This would take the form of three main checks: pre-condition (the right parameters are passed), post-condition (the results are returned correctly) and consistency checks (all data remains in a consistent state).

On the system-level, the required interface between two processing elements can be generalized to that shown in Figure 3.5. In addition to the execution control signals, a mechanism for transferring the live variables is needed. This additional mechanism may take the form of dedicated physical signals, memory mapped registers, or merely the use of a fixed dedicated address range in the shared memory. The shared virtual address space is crucial to this model as only a pointer to the shared memory region of the execution program needs to be passed to the auxiliary architecture during an architectural branch operation.

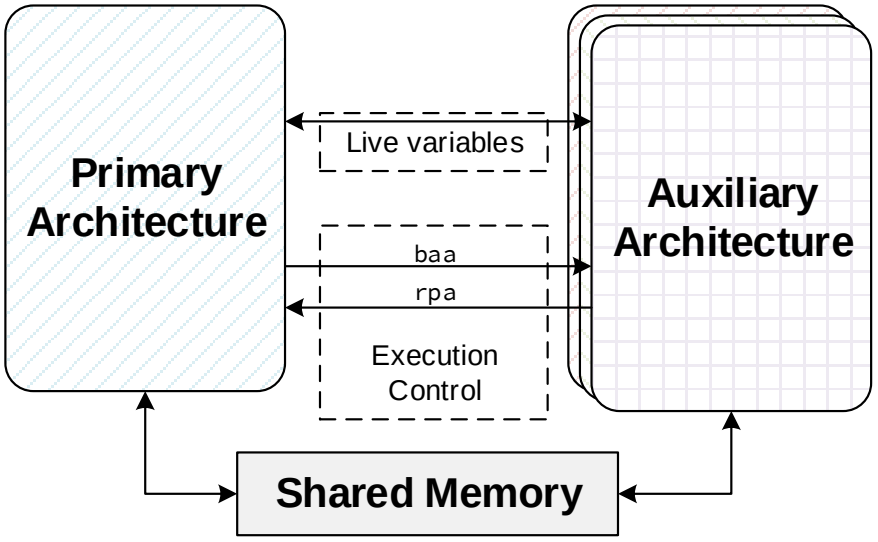

FIGURE 3.5: Generalized Hardware/Software interface

3.4.2 Compilation and Tools

Following the application partitioning phase of the hardware/software co-design, each portion of the application will need to be compiled. The existing familiar compilation tools available for each supported architecture are used to compile the relevant portions of the application. For general purpose processors, this would typically be the a standard programming language compiler. In the case of reconfigurable logic, the standard vendor-specific synthesis tools would be used.

Figure 3.6 shows the high level flow of an example mixed-architecture application compilation toolchain. There are two methods that may be used in combining the different architectural components to generate a single unified instruction stream binary:

(a) **Embedding** the compiled binary of the auxiliary architectures directly within the source code of the primary architecture program. This simpler method does not

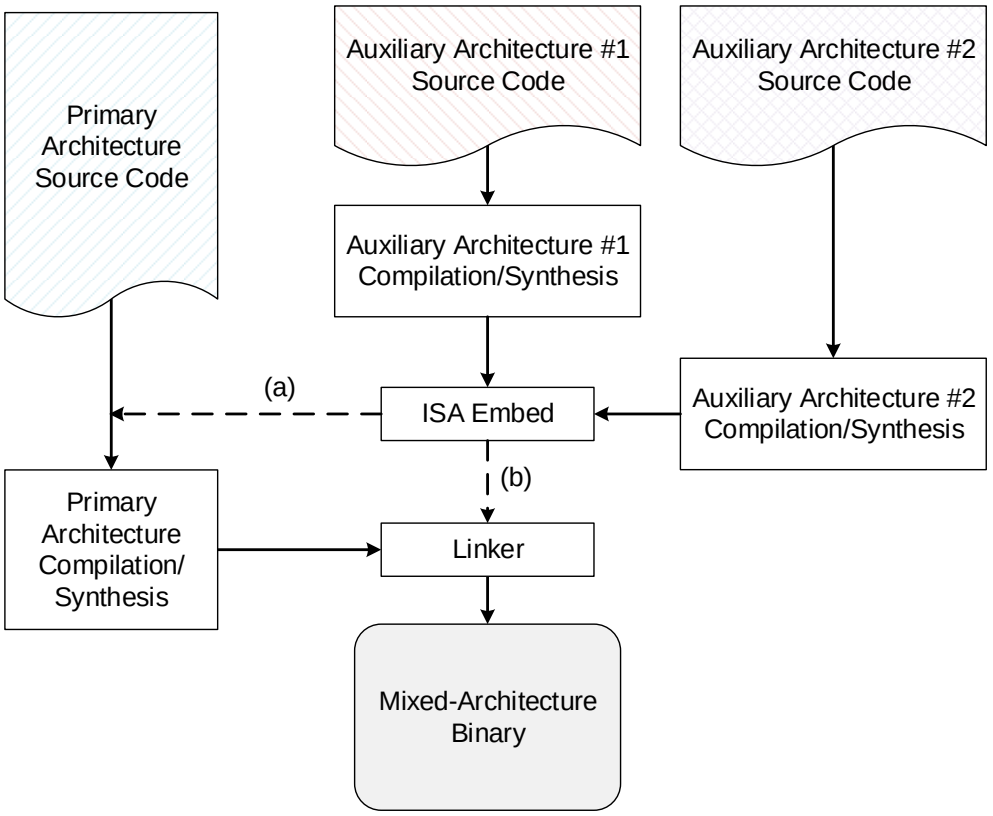

FIGURE 3.6: Mixed-Architecture Binary toolchain

require any special compiler or linker modifications. An ISA embedding tool wraps the compiled binary data from the auxiliary architectures into the assembly language directives of a high-level language. The output of this tool will be a software header file containing the embedded data that is wrapped into a function call or pre-processor macro, including the injection of the `baa` and `rpa` instructions as necessary. In this way, unmodified high-level language compilers like GCC or LLVM are able to directly produce a mixed-architecture binary application. This method is used in the MURAC systems described in Chapter 5 and Chapter 6.

(b) **Linking** the compiled auxiliary architecture directly with the compiled primary architecture object files. This may be achieved in an ISA embedding tool by wrapping the binary data from the output of the auxiliary architecture compiler into a linkable ELF object file within a function wrapper defining the exported interface. Another benefit of this method is that it enables the compiled auxiliary architecture application partitions to be created as shared libraries that are linked to the main application at runtime. Corresponding header files defining the interfaces to these shared libraries of auxiliary architectures would be required to be included in the primary architecture source code to make use of them. This method is demonstrated in the MURAC system simulator described in Chapter 4.

3.5 Chapter Summary

The Multiple Runtime-Reconfigurable Architecture Computer (MURAC) machine model introduces *system-level mechanisms* for creating a consistent abstraction of a heterogeneous computer. The key concepts enabling this model are the *unified machine model*, the *unified instruction stream* and a *unified memory space*. The unified machine model designates processing elements as either the primary architecture or additional auxiliary architectures, distinguished by their model of computation. A simple *programming model* is built upon this abstraction that provides a consistent interface for interacting with the underlying machine to the user application. The abstraction of the hardware/-software boundary within an application allows it to branch between different execution models through the ISA level operations. This programming model allows applications that utilise various heterogeneous computing resources to maintain a single stream of program execution. The communication interface between the partitions of the application is generalized through the use of the unified memory space. The system-level and software requirements for supporting the abstraction model are also discussed in this chapter.

Chapter 4

Instruction accurate system simulator

An instruction accurate system simulator for a prototypical system provides an opportunity to validate the abstraction of the MURAC model. This chapter demonstrates a *system-level* design and implementation of the model that supports the execution of hybrid mixed-architecture applications within an event-driven simulation kernel.

The system is implemented using the SystemC [IEE05, LMSG02] event-driven simulation toolkit that allows for system-level modelling of concurrent systems. The system employs Transaction-level modelling (TLM) [CG03] to model the communication between components. This provides a high-level approach that encapsulates the low-level details of communication mechanisms. These communication mechanisms, such as the interconnect bus between processing elements, are modelled as channels and the transaction requests are initiated by calling interface functions of these channel models.

4.1 MURAC system-level design

The high level components of the MURAC simulator system are illustrated in Figure 4.1. The simulator models a heterogeneous platform composed of a fixed general-purpose processor element, an auxiliary architecture interface unit and global shared memory regions.

The pipeline of an ARM Cortex-A8 processor core has been modified to support the MURAC `baa` mechanism as part of the ISA. Additional hardware interrupts have been added into the processor for signalling the transfer of the execution to the auxiliary architectural components and back. These changes effectively implement the MURAC

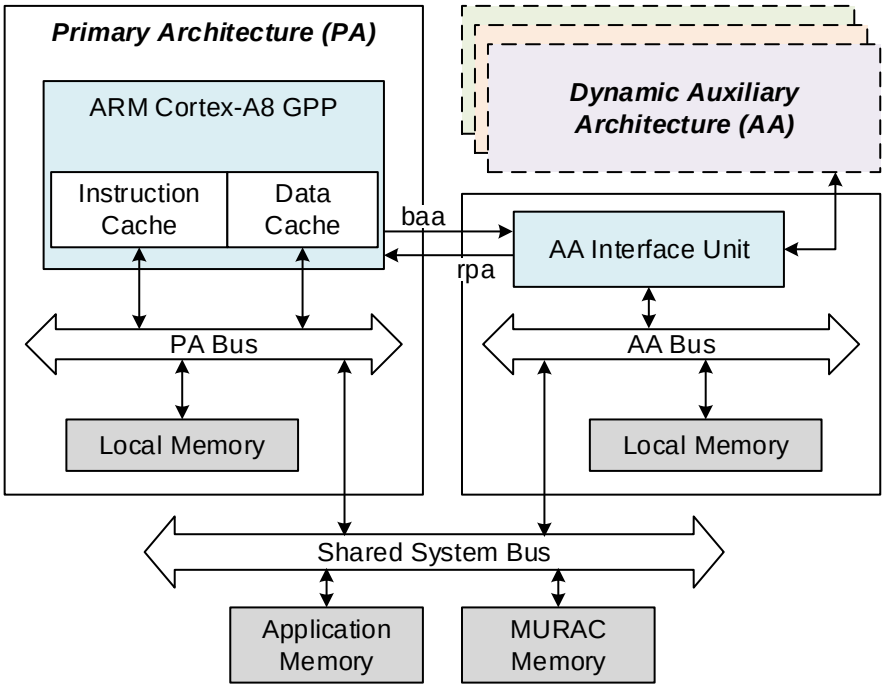

FIGURE 4.1: MURAC functional simulator

data-path morphing mechanism at the hardware level within the core. This primary architecture processor is implemented using the Open Virtual Platform (OVP)[1] processor simulator embedded within a SystemC TLM loosely-timed model to enable it to run within the simulation kernel.

Modelled as a peripheral device within the system, an auxiliary architecture interface unit is attached as a fixed element to the main system bus. This unit is responsible for loading and managing the dynamic auxiliary architecture cores which are provisioned directly from the running application's instruction stream. Arbitrary auxiliary architecture cores that are composed of SystemC models are fully embedded within the application and can be injected during runtime via the `baa` operation. To support the unified instruction stream concept, the auxiliary architectures are compiled into shared object files that are dynamically loaded, linked and executed during runtime by the auxiliary architecture interface unit.

To simulate the single-context view of the unified machine model, the primary architecture processor is put into a halted state for the duration of execution on the auxiliary architecture. Once the auxiliary architecture execution has completed, the system returns the execution to the primary architecture by signalling using an interrupt to the halted processor. Due to this design choice, this particular implementation of a MURAC machine does not provide any runtime support for a multi-threaded concurrency model in the application.

[1] http://www.ovpworld.org

Four separate memory areas are defined in the system:

- A local primary architecture memory area connected to the PA bus is used to provide a local scratchpad memory region for private variables and data.
- A local auxiliary architecture memory area connected to the AA bus is used to provide a local scratchpad memory region for private variables and data.
- A global application shared memory area that is connected to the shared system bus is used for storing the application data as well as the application instruction stream. This memory area plays the role of the unified memory space required by the MURAC model. This region is used for the application stack which includes all the live variables, such as parameters and return values, that are used in the application between the primary and auxiliary architectures. The primary and all auxiliary architectures are able to autonomously read and write to this memory area.
- A reserved MURAC system shared memory area connected to the shared system bus is used to transfer runtime information between the primary architecture processor and auxiliary architecture interface unit. This memory area is used by the system to share the information such as the program counter and application stack location for facilitating the `baa` and `rpa` operations.

4.2 Programming model

The application is provided with a standard interface for communicating between application primary and auxiliary architecture code, shown in Listing 4.1. This interface defines 3 macros that are used by the application to perform an architecture branch between the primary architecture and auxiliary architectures.

The `MURAC_AA_INIT` and `MURAC_AA_EXECUTE` operations are implemented in the application auxiliary architecture code partition. These operations are triggered by the simulator when the application branches to the auxiliary architecture, and are used to perform any initialisation if necessary, and then perform the computations respectively. The implementation of the application auxiliary architecture code will be defined within this `MURAC_AA_EXECUTE` operation.

The primary architecture application partition is provided with one generic operation, `MURAC_SET_PTR`, which is used to indicate the memory address of the stack containing the live variables of the application. All live variables that are to be shared across the processing element boundaries must be explicitly created in a region in the global application shared memory area. In addition to this, the application is able to perform the `baa` operation by using the respective `MURAC_AA_EXECUTE` implementation macro that

is defined in the generated header file (c.f. Section 4.2.1) for each auxiliary architecture partition.

The detailed runtime execution behaviour and interaction of these operations is further described in Section 4.3.

```
// ./framework/murac.h
#ifndef MURAC_H
#define MURAC_H

// Setup Auxiliary Architecture
#define MURAC_AA_INIT(NAME) extern "C" int murac_init(BusInterface* bus)

// Execute Auxiliary Architecture
#define MURAC_AA_EXECUTE(NAME) extern "C" int murac_execute(unsigned long int stack)

// Store location of live variables
#define MURAC_SET_PTR(ADDR) asm volatile("mov r0,%[value]" : : [value]"r"(ADDR) : "r0");

class BusInterface {
public:
    virtual int read(unsigned long int addr,
                     unsigned char*data,
                       unsigned int len) = 0;

    virtual int write(unsigned long int addr,
                      unsigned char*data,
                        unsigned int len) = 0;
};
#endif
```

Listing 4.1: MURAC auxiliary architecture interface

As an illustration of the programming model, applications have been designed containing algorithms that benefit from using alternate execution architectures in addition to sequential general purpose processing. These applications were designed to use the primary architecture to handle the reading of input and parameters into memory, and then branch to the auxiliary architecture to perform the core computation before returning back to the primary architecture to output the results. The sequential code targeted for the primary architecture was developed in C, while the parallel portion of the application was written in a SystemC wrapped program that allows for any arbitrary computational model.

The applications are executed directly in the absence of an operating system on the simulator. Although, these applications illustrate the use of only a single auxiliary architecture, it is trivial to extend this to include any number of additional auxiliary architectures. This can be achieved by including the respective header files in the primary architecture application and calling the respective `MURAC_AA_EXECUTE` macros at each point in the program where the auxiliary architecture computation is desired.

The source code for an Advanced Encryption Standard (AES) encryption application is shown in Listing 4.2. The AES encryption core is implemented as a parallel SystemC model[2] to be run as an auxiliary architecture of the simulator.

[2]Based on SystemC AES available at http://opencores.org/project,systemcaes

Similarly, Listing 4.3 shows the source code snippet for a genetic sequence alignment application featuring an auxiliary architecture core that implements the Smith-Waterman algorithm [SW81] for local sequence alignment[3].

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../aa/embed/aes128.h"       // embedded auxiliary architecture core
#include "../../../framework/murac.h" // shared library interface

int main(void) {

    printf("[PA] MURAC AES-128 example...\n");

    char *key   = (char *) malloc(16);
    sprintf(key, "mysimpletestkey!");
    char *input = (char *) malloc(16);
    sprintf(input, "random inputdata");
    char *encrypt_output = (char *) malloc(16);
    char *decrypt_output = (char *) malloc(16);
    memset(decrypt_output, 0, 16);

    unsigned int *stack = (unsigned int *) malloc(4*sizeof(unsigned int));
    stack[0] = (unsigned int) key;
    stack[1] = (unsigned int) input;
    stack[2] = (unsigned int) encrypt_output;
    stack[3] = (unsigned int) decrypt_output;

    MURAC_SET_PTR(stack)          // Set live variables

    EXECUTE_AES128                // Perform computation on AA

    printf("[PA] Encryption output: ");
    for (int i = 0; i < 16; i++) {
        printf(" 0x%x", encrypt_output[i]);
    }
    printf("\n[PA] Example finished...\n");

    free(key);
    free(input);
    free(encrypt_output);
    free(decrypt_output);
    free(stack);

    return 0;
}
```

Listing 4.2: Cryptographic application - Primary architecture source listing

4.2.1 Compilation

This mixed-architecture application compilation process is illustrated in Figure 4.2. The system has been designed to allow for the use of the standard compilation tools (GCC for ARM) without modification. An ISA embedding tool is used to directly inject the `baa` instruction as well as the binary data of the compiled auxiliary architecture modules. This tool generates a header file that exposes macro calls that will directly insert this information in the ISA of the primary architecture. This binary contains both standard

[3]Based on DNA Sequence Alignment Accelerator available at http://opencores.org/project,seqalign

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../aa/embed/seqalign.h"      // embedded auxiliary architecture core
#include "../../../framework/murac.h"  // shared library interface

int main(void) {

    printf("[PA] Sequence Aligment example...\n");

    unsigned char *input = (unsigned char*) malloc(4);
    unsigned char *output = (unsigned char*) malloc(1);

    .... // Write data from command line to input variable

    unsigned int *stack = (unsigned int *) malloc(2*sizeof(unsigned int));
    stack[0] = (unsigned int) input;
    stack[1] = (unsigned int) output;

    MURAC_SET_PTR(stack)          // Set live variables

    EXECUTE_SEQALIGN              // Perform computation on AA

    printf("[PA] Sequence Alignment result: ");
    printf(*output > 0 ? "MATCH\n" : "NO MATCH\n");
    printf("[PA] Example finished...\n");

    free(input);
    free(output);
    free(stack);

    return 0;
}
```

Listing 4.3: Sequence alignment application - Primary Architecture source listing

ARM ISA instructions for execution on the simulator primary architecture, as well as compiled SystemC binary code for execution on the simulator auxiliary architecture.

4.3 Execution model

To illustrate the system-level implementation of the MURAC model during the architecture branch operations, the following sequence of events occurs in the runtime of a simulated application kernel:

1. The application execution begins on the primary architecture. The initial partition of the application runs until the application wishes to morph the underlying architecture by branching to an auxiliary architecture.

2. The application saves the location of the stack containing the live variables into the predefined `r0` register in the primary architecture by calling the `MURAC_SET_PTR` macro.

3. The `baa` instruction is issued on the PA which will be handled by:

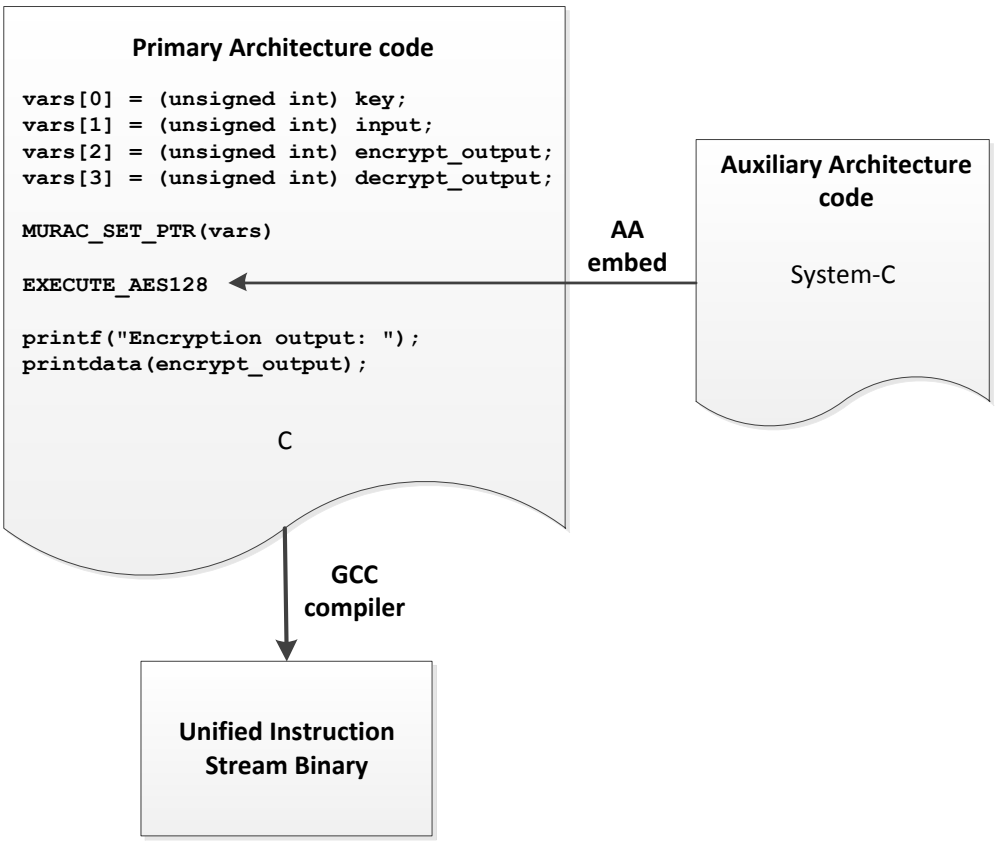

FIGURE 4.2: Simulator application compilation

(a) Writing the current program counter to a predefined location in the reserved MURAC shared memory area.

(b) Writing the size of the auxiliary architecture instruction to a predefined location in the reserved MURAC shared memory area.

(c) Writing the stack location (currently in the `r0` register) to a predefined location in the reserved MURAC shared memory area.

(d) Halting the primary architecture core

(e) Updating the primary architecture core program counter to the value of the next primary architecture instruction after the auxiliary architecture instruction in the unified instruction stream memory.

(f) Triggering the `baa` interrupt on the AA Interface Unit

4. The AA Interface Unit will begin execution and:

(a) Read the current program counter, size of the auxiliary architecture instruction and the live variables location from the predefined location in the reserved MURAC shared memory area.

(b) Extract the auxiliary architecture instruction from the unified instruction stream from the global application shared memory area.

(c) Load the auxiliary architecture instruction as a shared simulation library into the simulation kernel and call the `MURAC_AA_INIT` function of this library. The bus interface is passed to this function so that the auxiliary architecture is dynamically connected to the shared memory address space.

(d) Run the loaded auxiliary architecture by executing the `MURAC_AA_EXECUTE` function of this library in a separate SystemC simulation thread. The stack location is passed to this function so that the auxiliary architecture has access to the live variables of the application.

(e) Wait until the auxiliary architecture simulation terminates.

(f) Trigger the `rpa` interrupt on the Primary Architecture and halt execution of the AA Interface unit.

5. The primary architecture will resume execution at the next instruction of the application after the auxiliary architecture.

6. The application will be able to retrieve any results or data by reading from the live variables that were passed as a stack pointer to the auxiliary architecture.

4.4 Chapter Summary

This chapter has demonstrated a simulator for systems designed using the MURAC model presented in Chapter 3. Using this simulator, a practical realization of a MURAC system that implements the core abstraction on the system-level has been designed and created. The system has implemented a RISP style architecture that extends a CPU core with arbitrary application specific capabilities. The practical mechanisms involved in supporting the unified abstraction to the application programming model have been shown through a detailed explanation of the execution of the MURAC architecture branch operations.

Demonstrating the programming model, two applications are presented that are representative of programs that are usually targeted for implementation on heterogeneous system due to their computationally heavy algorithms. Through these examples the benefit of the explicit specification of the architectural boundary call is illustrated, demonstrating that the application has become more *portable* as the lower-level device-specific details that are normally required in such programs are no longer present. These examples have also illustrated the loosely coupled nature of the different computational kernels within an application, illustrating how the individual components of the application can be more readily *re-usable* as the hardware/software boundary and communication has been made explicit.

Chapter 5

A MURAC System-on-Chip

Following on from the simulated design of a MURAC system, this chapter will apply the model to the design and implementation of a practical embedded heterogeneous System-on-Chip (SoC). This system-level implementation of the MURAC model is realized in the form of an Application Specific Instruction Set Processor (ASIP) composed of an embedded general purpose processing (GPP) system extended with fixed application-specific accelerators. An illustration of the design process for this system shows the considerations made in mapping the MURAC model to the underlying platform.

The SoC is synthesised and implemented on a Xilinx Spartan-6 FPGA platform. This implementation is used to measure the overhead of the MURAC model support by comparing the resource utilization with an equivalent base system. This system implementation demonstrates the applicability of the MURAC model to embedded systems, where the abstraction is able to provide a consistent interface to bare-metal applications in the absence of an operating system. In this design only single-threaded processes are supported, as is the case in real-world embedded systems without runtime support.

The programming model demonstrating the use of the fixed application-specific accelerators is used to show that the abstraction provided to the application remains consistent with respect to different underlying system implementations. Even though the system implementation varies greatly with that of the simulator described in Chapter 4, the application source code closely resembles that of the simulator application described in Section 4.2.

5.1 System design

The system is based on the Gaisler LEON3[1] soft-processor core. This GPP contains a 7-stage pipeline with a SPARC V8 [2] compatible instruction set. Using this platform as a target for the SoC, it is therefore natural to define the primary architecture of the SoC as the SPARC ISA. On the other hand, several options exist in terms of how the auxiliary architecture may be defined.

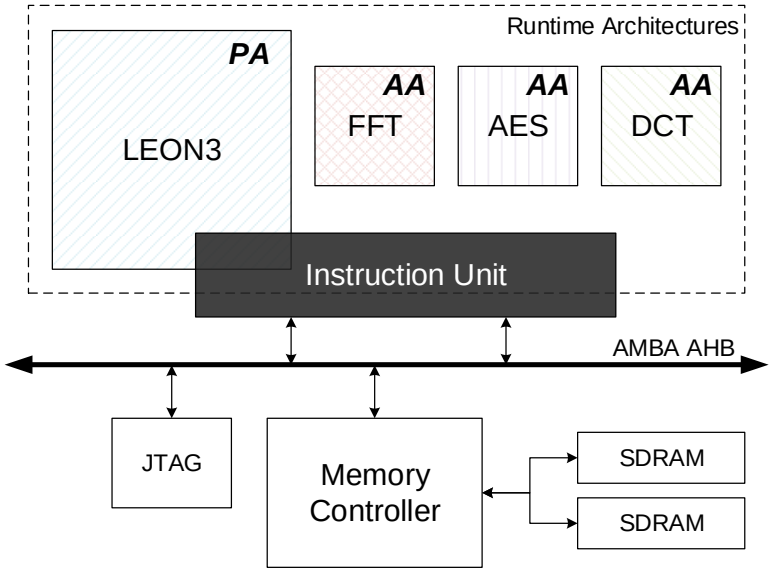

FIGURE 5.1: High-level block diagram of a MURAC SoC implementation

The system contains several fixed-function application accelerators implemented as custom hardware (Figure 5.1). Assuming that three accelerators targeted in the system are a fast Fourier transform module (FFT), a discrete cosine transform module (DCT) and an advanced encryption standard (AES) cryptography module, the unified machine model may be realized in two alternative strategies:

Option 1: FPGA fabric as auxiliary architecture One option is to model the low-level FPGA fabric itself as an auxiliary computing architecture. In particular, a partial reconfiguration region of the FPGA can be designated to load any one of the three accelerators above during runtime. With this definition, the partial reconfiguration bitstream of each accelerator essentially serves as a very long instruction word of the user application.

Once the MURAC machine is morphed into this auxiliary architecture via the `baa` instruction, the instruction fetch (IF) unit would take over the task of fetching and loading the partial bitstream to the partial reconfiguration region. Under the MURAC

[1] http://www.gaisler.com/index.php/products/processors/leon3
[2] http://sparc.org/technical-documents/specifications/

model, such a partial bitstream has become part of the instruction stream, embedded in the instruction memory directly after the `baa` instruction in the user program. Compared to a normal processor, this auxiliary architecture simply has a much wider and varying instruction width. Once loaded, the acceleration module will subsequently carry out its computation in the form of the instruction execution (IE) stage of the idealized MURAC processor pipeline. Additional auxiliary architecture instructions may be loaded and executed subsequently until an `rpa` instruction is encountered, in which case the machine will return to load and execute the following instructions with a SPARC ISA in the primary architecture pipeline.

The use of partial reconfiguration allows for physically changing the runtime architecture of the MURAC machine according to the user application. The key benefit of defining an auxiliary architecture in this way is that it allows user-defined application accelerators to execute on an FPGA as a run-time gateware. It forms a general purpose architecture that allows for the execution of different gateware applications not known at compile time. The drawback of this option, however, is the very long instruction fetch and loading time due to the size of the partial bitstreams. To hide such long loading time complex pre-fetching and caching schemes must be employed.

A system designed in this way would greatly benefit from the runtime support provided by an operating system, and this design choice has been explored further to implement the system in Chapter 6. For the sake of simplicity, the design of the embedded SoC assumes that the accelerators are known at compile time, which allows it to avoid the long configuration times by defining its *auxiliary architecture* as a CISC architecture.

Option 2: Custom CISC ISA as the auxiliary architecture As an alternative option, which the embedded SoC design has adopted, the auxiliary architecture is defined as a custom complex instruction set computer (CISC) architecture. In the system, this custom ISA supports exactly 4 complex instructions: one instruction for each accelerator implemented and the `rpa` instruction. The instruction unit of this architecture is responsible for fetching and decoding complex instructions from the user program. At the same time, the three accelerators act as the execution unit of the CISC machine and perform all necessary computation in multiple cycles. Upon completing the action of a particular accelerator instruction, the CISC machine continues by fetching and executing the next instruction from memory until the `rpa` instruction is encountered that will cause the program execution to resume on the primary architecture.

By defining the auxiliary architecture in this way the instruction width of the ISA is constant and significantly shorter than that of Option 1 above. However, such a definition is possible only because the types of accelerator functions are fixed when the system is implemented. This design choice becomes a trade-off of flexibility for performance in a simple embedded design.

By adopting the definition of an auxiliary architecture as described in Option 2 above, a custom non-pipelined CISC machine is implemented and integrated with the modified LEON3 processor to form the SoC. The instruction unit of the LEON3 processor has been modified to support additional logic implementing the `baa` operation, as well as exposing the relevant data signals, such as the current program counter (PC) and stack pointer (SP) registers, to allow for integration with the auxiliary architecture. Uniform access to a shared memory region is available via a memory controller which is connected to the high speed AMBA AHB system bus. Through the specialized instruction unit the fixed auxiliary architectures support direct access to the main memory. However, for the sake of simplicity and in the absence of runtime support, the system lacks full virtual memory support.

5.2 Execution model

The custom CISC ISA acting as an auxiliary architecture is implemented as a simple state machine that communicates with the modified LEON3 processor. In effect, it extends the LEON3 instruction pipeline and is able to fetch and decode instructions from the main memory autonomously. As there are only 3 instructions defined in this CISC architecture (excluding the `rpa` operation), its main task in the instruction execution stage is to enable the computation of the corresponding accelerator module. Additional accelerator support may be incorporated by simply extending this custom ISA as needed.

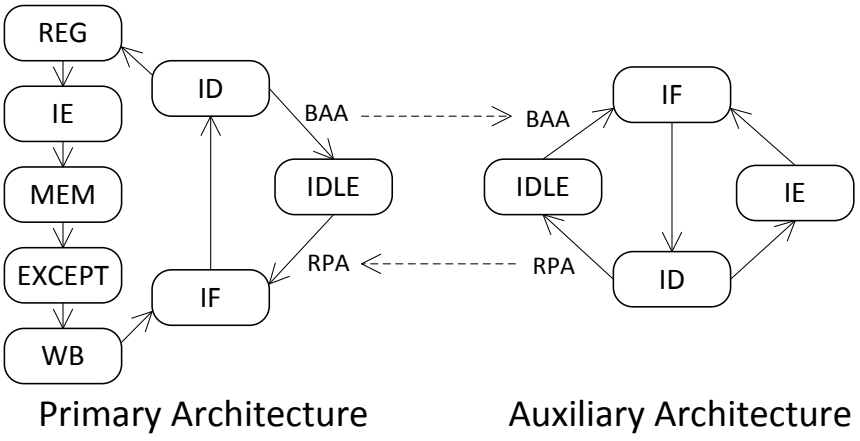

FIGURE 5.2: Instruction unit operation

The operation of the instruction unit is illustrated by Figure 5.2. The operation of branching between architectures, which effectively changes the system execution model during application runtime, is achieved at the ISA level by issuing a `baa` instruction in an application running on the LEON3 primary architecture, or a `rpa` instruction on the auxiliary architecture. The operation of the system upon execution of this `baa` instruction proceeds as follows:

1. The program counter and stack pointer registers of the primary architecture are exposed to the auxiliary architecture.

2. A `baa` operation is signalled to the auxiliary architecture, causing it in turn to enter the instruction fetch *IF* state (previously in the *IDLE* state).

3. The primary architecture will annul the next instruction (which would correspond to the first auxiliary architecture instruction, already loaded into the pipeline), and stall the pipeline.

4. The auxiliary architecture issues a memory read instruction during the *IF* state, and waits for the instruction to become available on the bus.

5. The auxiliary architecture enters the instruction decode (*ID*) state, decoding the instruction and continuing to the relevant state based on the instruction opcode:

   - If the `rpa` instruction is encountered: the auxiliary architecture will output the current Program Counter (PC), signal the `rpa` operation to the primary architecture, and enter the *IDLE* state.
   - If an unknown instruction occurs, this will be signalled to the *primary architecture* where it will be dealt with as an unknown operation exception. The auxiliary architecture will enter the *IDLE* state once again.
   - If an available CISC instruction is encountered, the auxiliary architecture will execute the microcode, which includes fetching any data needed from memory and invoking the included instruction execution logic. Once completed, the auxiliary architecture will re-enter the *IF* state and repeat the process from step 4.

6. The primary architecture will continue pipelined execution, where it will read the next instruction from the location in the instruction stream pointed to by the update program counter received from the auxiliary architecture.

5.3 Programming model

Listing 5.1 includes a code snippet from an application designed and run on the MURAC SoC. This code highlights the usage of the MURAC abstraction in executing a portion of the code to perform AES encryption using a fixed hardware cryptographic core that is implemented as an auxiliary architecture.

For simplicity in the embedded system, the `baa` and `rpa` instructions have been expressed as in-line assembly code through *C macros*, allowing the application to be compiled with the tool-chain of the primary architecture (BCC for LEON3). Alternatively, these operations could be automatically generated by tools or a compiler that has been modified

```
#define BAA asm volatile(".word 0x20400001");
#define RPA asm volatile(".word 0x80000000");

int main() {
    ...
    BAA
    aes_encode(key_data, data_in, data_out);
    RPA
    print(data_out);
    ...
}
```

Listing 5.1: MURAC embedded SoC application

for producing a unified instruction stream application based on the target auxiliary architecture. This illustrates the simplified programming model for switching execution between alternate computational architectures with the use of a single instruction, as well as the explicit control flow of accelerator utilization from the point of view of the application similar to that in Figure 3.4(b).

The portion of code targeted to run on the auxiliary architecture generates the relevant architecture specific instructions based on the target auxiliary architecture ISA. In the case of the system design implemented as described in Section 5.1 using a custom CISC ISA, a single in-line assembly `aes` instruction will be embedded directly into the unified instruction stream. The compiler would play the role in determining the actual instructions generated for the unified instruction stream.

The above embedded application is run as bare-metal in the absence of an operating system. Even in the case where the application would be running under an operating system, the low level device interaction and control details are hidden from the application developer (c.f. Section 6.2). The unified memory address space is available to all processes within the embedded system without needing to keep track of the consistent mappings of virtual addresses between them. This enables the auxiliary architecture to directly access the relevant data required by addressing the relevant memory locations, with consistency ensured across architecture boundaries upon execution of the `baa` and `rpa` operations. the locations of application data stored on the heap or application stack can be communicated between architectures in the system via the shared register value of the stack pointer as part of the `baa` or `rpa` operations. In this way, each architecture is able to execute autonomously, supporting the control flow abstraction provided to the application.

5.4 Implementation and Analysis

The MURAC model delegates most of the complex interfacing logic between the host CPU and the accelerators as additional micro-architectural features with a goal to provide a unified ISA. Due to this it is important that the additional platform hardware that enables MURAC features be kept at a minimum when compared to a typical implementation that doesn't follow such a model.

A summary of the device resource utilization of the SoC on the Xilinx Spartan 6 XC6SLX45T FPGA-based platform is shown in Table 5.1, which lists the resource utilization of three implementations:

(a) an off-the-shelf LEON3 System-on-Chip,

(b) a MURAC enabled SoC without any fixed application-specific accelerators.

(c) a MURAC enabled SoC with an `aes` CISC instruction supported by a fixed application-specific accelerator.

Implementation (c) describes the full SoC processor that supports execution of the sample code in Listing 5.1. With only a single application-specific accelerator included, it represents the worst case scenario of the hardware overhead required to support the MURAC abstraction in this system. This overhead is determined via a comparison with an identical SoC implementation without the additional hardware elements required.

The additional resources that consist of the MURAC model implementation are:

- the extended instruction unit, including the signals interfacing the LEON3 core with the CISC core and handling the `baa` and `rpa` operations,
- the unpipelined CISC processor core and its connection to the memory bus.

Table 5.1: Resource utilization of the SoC implemented on a Xilinx Spartan 6 XC6SLX45T FPGA.

FPGA Resources	(a) LEON3	(b) Base SoC	(c) SoC + AES
Slice Registers (54,576)	6,813	6,912	8,121
Slice LUTs (27,288)	11,666	11,705	14,913
Occupied Slices (6,822)	4,013	4,333	5,166
LUT Flip Flop pairs	12,781	13,057	16,335
Block RAM (348)	30	30	30

Comparing the resource utilizations from the three columns in Table 5.1 to the anticipated typical implementation above, a hardware overhead of 0.26% of the FPGA Slice LUTS was observed as shown in Figure 5.3.

Fundraiser Results by Salesperson

AES core	3 208
MURAC logic	39
Base Leon3	11 666

ıtational archi-
gic required to
ıis small hard-
the underlying
n. Such efforts
ved portability

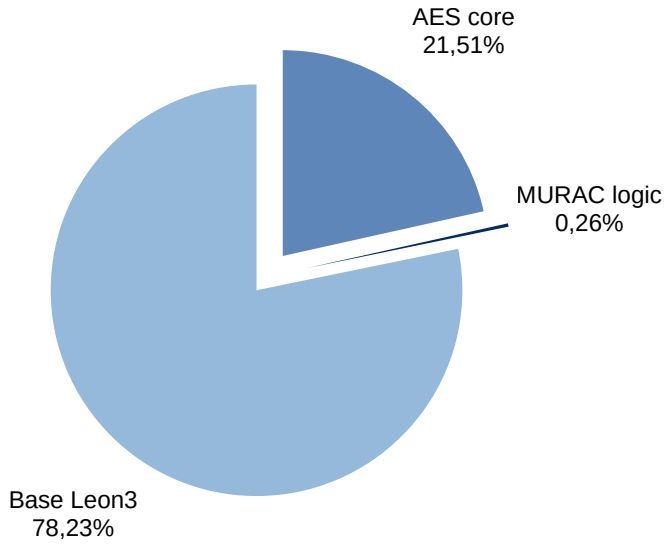

FIGURE 5.3: Composition of resource utilization (FPGA LUTs) for a SoC with AES core

1

5.5 Chapter Summary

An embedded System-on-Chip has been designed using the MURAC model, consisting of an application specific instruction processor (ASIP) with fixed-instruction accelerator cores. The process of mapping the MURAC model to the target platform has been illustrated, leading to the design choice of using a custom CISC ISA core as an auxiliary architecture. The auxiliary architecture implementation is discussed along with the instruction unit modifications necessary to support the MURAC architecture branch operations. A detailed explanation of the execution of these operations illustrates the system-level detail that is abstracted by the MURAC model. Through a comparison of the implementation of the system on the target platform, it is shown that the overhead required in terms of hardware resources for the MURAC system-level logic is minimal.

This system-level design demonstrates the benefits of the abstraction and the programming language agnostic features of the MURAC model. The application programming model is able to be applied to bare-metal programs without the need for runtime support from an operating system.

Chapter 6

Runtime environment

This chapter explores the interaction between the operating system and the underlying machine to support efficient multi-user computation in heterogeneous systems. A tightly coupled CPU+FPGA platform is used for the implementation of a simple operating system scheduler for this heterogeneous machine to demonstrate a multitasking environment capable of running mixed-architecture applications. The MURAC model enables the development of a multi-user runtime environment where processing elements are fairly time-shared among users in the same framework as a conventional homogeneous CPU-based machine. The model extends the idea of elevating hardware processes into the UNIX process model [STB06] by moving the hardware-software boundary into the process itself. This single-context process abstraction provides an advantage to scheduling in a multitasking environment, and is supported by the consistent view of underlying heterogeneous processors provided by a system-level implementation of the model.

6.1 System design

An experimental system is implemented as a reconfigurable instruction set processor (RISP) using partial configuration to dynamically provide customized instructions as arbitrary auxiliary architectures on a programmable logic fabric. This system is modelled after the design choices explored in Option 1 of Section 5.1, where a partial reconfiguration region of the programmable logic is used to provision application accelerator cores at runtime. In this approach, the partial reconfiguration bitstream of each auxiliary architecture essentially serves as a very long instruction word of the user application. The Xilinx Zynq XC7Z020 based Zedboard (Figure 6.1) is used as an implementation platform, consisting of a tightly coupled dual Cortex-A9 Processing System (PS) and Programmable Logic (PL) region, shown in Figure 6.2. The processing system, which naturally assumes the role of the primary architecture, hosts the linux operating system

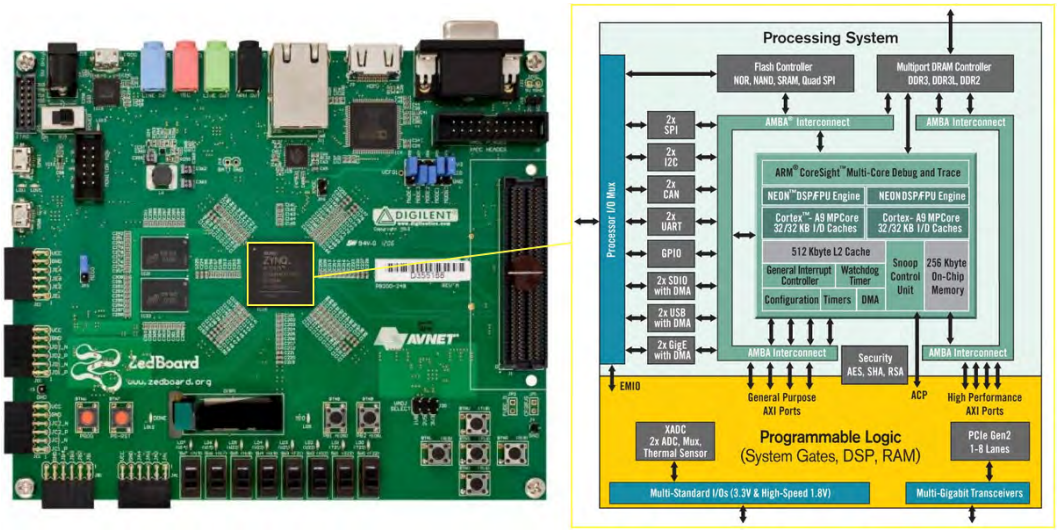

FIGURE 6.1: Xilinx Zynq XC7Z020 based Zedboard platform

kernel. The programmable logic region is used to support dynamically reconfigurable auxiliary architectures which are configured directly from an application instruction stream during runtime.

A kernel module has been developed to implement the system-level requirements of the MURAC model in a software layer without the need for hardware modification. This kernel module is also used to implement the extended programmable logic scheduler, further described in Section 6.4.2.

An AXI bus interface is connected to the *Accelerator Coherency Port* (ACP) from the programmable logic region to support the unified virtual memory space required by the MURAC model. This ensures that any read operations to the shared memory from the programmable logic are looked up in the processor caches, and write operations to the shared memory invalidate the appropriate cache lines in the processor system. Using this mechanism, the application running on the programmable logic is able to consistently access program data (either in memory or on the heap), ensuring coherency in addition to high performance. In order to support virtual memory within the operating system, the kernel module is tasked with translating the virtual memory addresses of the application stack in the processing system into a physical address that is passed to the application running in the programmable logic.

The Zynq system includes a dedicated device configuration (*DevCFG*) unit that provides a mechanism for the dynamic reconfiguration of the programmable logic via a Processor Configuration Access Port (PCAP). To take advantage of the partial reconfiguration to support the MURAC programming model, the *DevCFG* unit is used to configure the programmable region with a base hardware image implementing an AXI connected *Internal Configuration Access Port* (ICAP) module that acts as the interface to the programmable logic. This base image is automatically configured by the kernel module

as it is loaded into the operating system. Thereafter, during a context switch performed by the programmable logic scheduler, the ICAP core running on the programmable logic is used to configure and read-back the partial configuration data relevant to the current executing task. In the case of this partially reconfigurable implementation, the hardware portion of the user application design must be coordinated with that of the initial configuration performed by the kernel module to ensure that it is compatible with the running gateware.

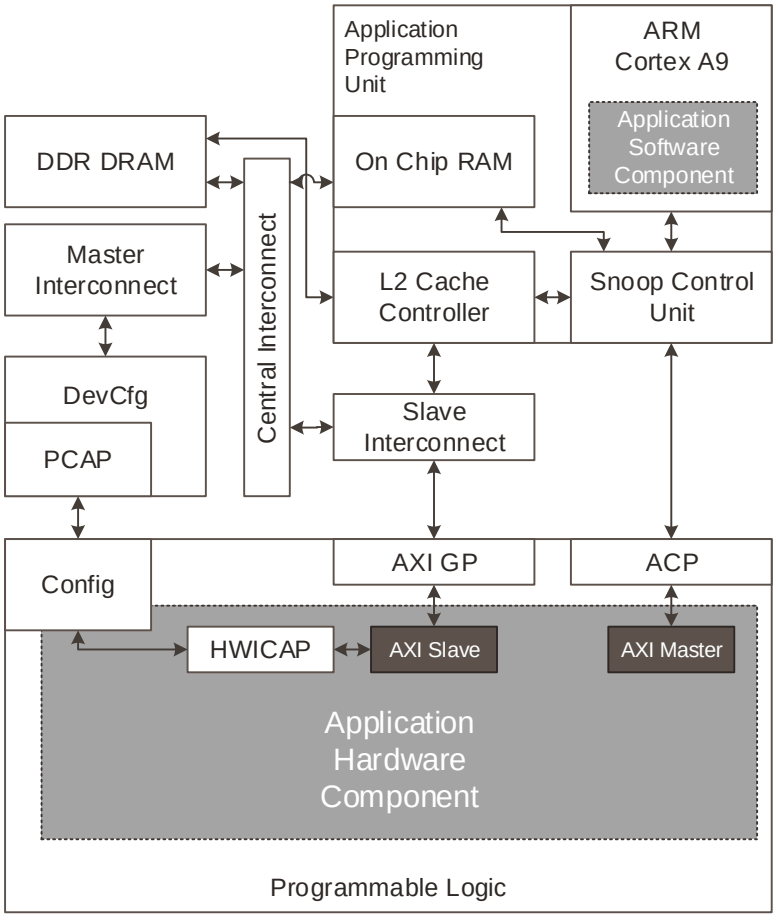

FIGURE 6.2: System overview

6.2 Programming Model

An explicit calling convention between the different components of the application has been used to simplify the programming model for the application. The operating system kernel module will automatically pass the stack pointer of the application through to the auxiliary architecture portion of the application as part of the `baa` operation. This simplifies the implementation as there is no need for any additional mechanism to explicitly pass data between the processing system and the programmable logic. However, the order and size of parameters must be carefully coordinated to ensure that they are

consistent between the different components of the application across the architectural boundary.

The user design must support a mechanism to signal completion of the hardware task to the processing subsystem, indicating the `rpa` instruction to the waiting kernel module in order to continue task execution on the primary architecture.

As a test case for evaluating the scheduler, a synthetic workload application has been designed that implements a parametrized programmable logic region as auxiliary architecture code (Figure 6.3). This part of the application implements a simple countdown timer register that triggers the `rpa` operation when finished. This mechanism allows the application to control the execution time of the auxiliary architecture portion via this register.

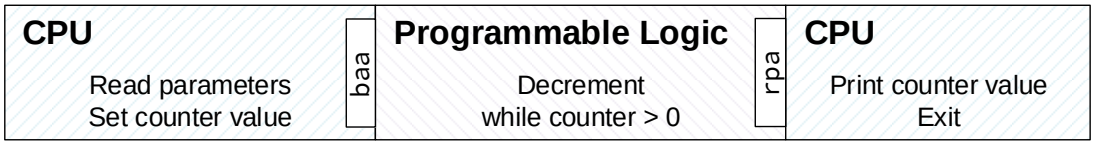

FIGURE 6.3: Mixed-architecture application

```
#include "include/config_impl_partial.h"

#define HW_ADDR 0x60A00000

int main(int argc, char *argv[]) {

    unsigned int reg;

    // Input counter start value from command line parameters
    reg = (argc > 1) ? atoi(argv[1]) : 0xAAAAAAAA;

    // Branch to FPGA
    BAA_CONFIG_IMPL_PARTIAL(HW_ADDR)

    printf("Output: 0x%x\n", reg);

    return 0;
}
```

LISTING 6.1: Example mixed-architecture application source listing

```
#ifndef CONFIG_IMPL_PARTIAL_H
#define CONFIG_IMPL_PARTIAL_H

#define BAA_CONFIG_IMPL_PARTIAL(PERIPHERAL_BASE) \
  asm volatile("mov r1,%[value]" : : [value]"r"(PERIPHERAL_BASE) : "r1"); \
  asm volatile("mrc 1,0,R15,c1,c2"); // BAA instruction \
  asm volatile( // Embedded Programmable Logic configuration data \
  ".word 0x52554dff\n\t" \
  ".word 0x1004341\n\t" \
  ".word 0x30600300\n\t" \
  ".word 0xffffffff\n\t" \

  ...
```

LISTING 6.2: Partial embedded bitstream header file

The mixed-architecture application compilation toolchain is demonstrated in Figure 6.4. The hardware portions of the application - the auxiliary architectures - that are targeted for the reconfigurable logic in the system are compiled using a standard FPGA toolchain. In this case, Vivado HLS synthesis tools were used to generate the HDL code that is synthesised for the Zynq Programmable Logic.

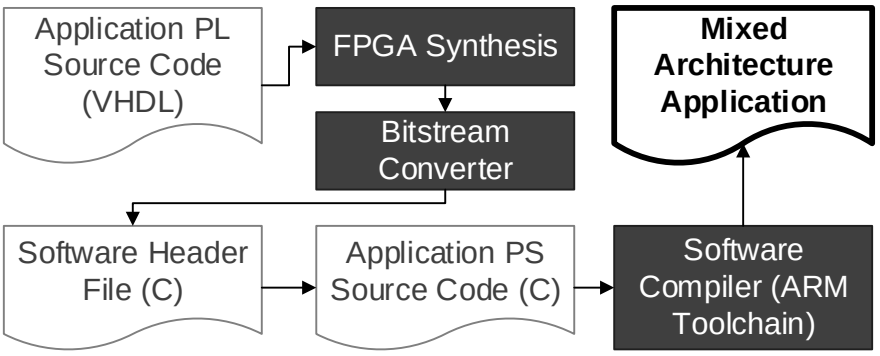

FIGURE 6.4: Mixed-architecture application compilation

The resulting configuration bitfiles are then embedded directly into the software application source code at the desired point of execution within the program flow. This is achieved at the ISA level as an architectural branch instruction followed by the configuration data (an example of this is illustrated in Listing 6.2). This mechanism enables the entire mixed-architecture application code to be compiled by the unmodified toolchain targeted for the primary architecture CPU, in this case an ARM compiler for the Zynq processor system, producing a single executable binary application file.

Listing 6.1 shows a code snippet for our example synthetic workload application, showing the usage of the MURAC abstraction to execute portions of the code using an alternate compute model on the programmable logic fabric. This illustrates the simplified programming model for switching execution between alternate compute architectures with the use of a single instruction, as well as the explicit control flow of accelerator utilization from the point of view of the application.

6.3 Execution Model

An operating system kernel module enables the execution of mixed-architecture binary applications within a standard linux kernel by implementing the programmable logic specific scheduler extension. This module is responsible for the configuration of the programmable logic region as well as co-ordination and management of task scheduling which is transparent to the user process. Upon being loaded into the operating system kernel, the module hooks into the kernel instruction handler to enable device configuration and scheduling upon execution of an *architectural branch* (`baa`) instruction in a user process. The management of the programmable logic is performed by this module

that extends the operating system and enables runtime support for the execution and scheduling of mixed-architecture applications.

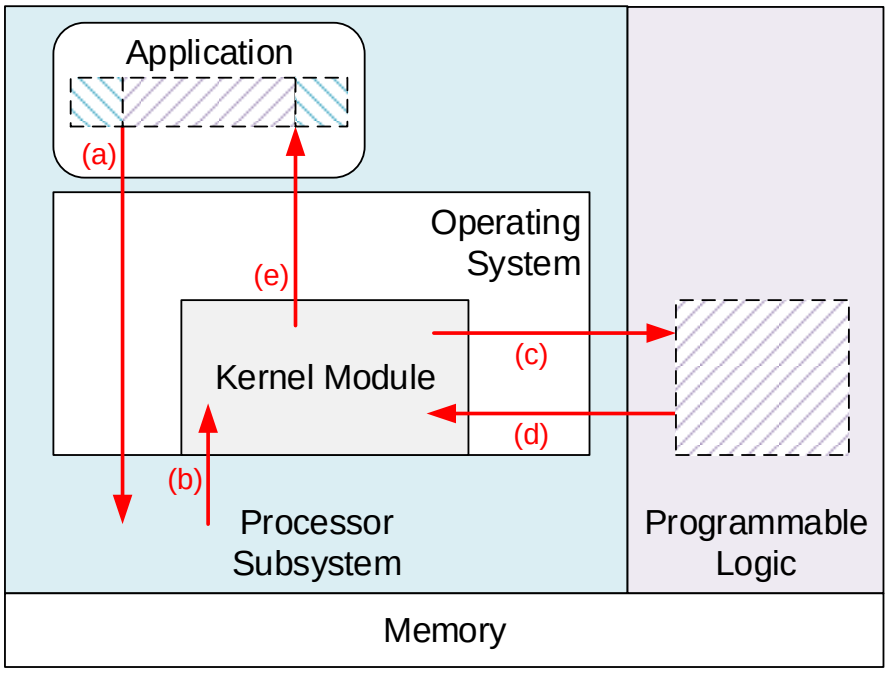

FIGURE 6.5: Architectural branch operation emulation in the Operating System

The details of the MURAC architecture branch operation implementation are illustrated in Figure 6.5. This sequence of operations is initiated when the application instruction stream currently running on the processor subsystem executes the `baa` instruction.

(a) The processor pipeline encounters the `baa` instruction in the application instruction stream. At this point the bus address of the template programmable logic interface containing the ICAP core is available in a processor register.

(b) The processor issues an interrupt to the operating system layer, which is serviced by the kernel module.

(c) The kernel module extracts the programmable logic configuration data available at the current program counter in the instruction stream and provisions the partial region of programmable logic via the ICAP primitive. The kernel module will block the application on the processor until the programmable logic issues a `rpa` instruction. During this period, the kernel module may execute any programmable logic context switches as directed by the scheduler.

(d) The `rpa` operation is signalled via a shared register implemented in the programmable logic.

(e) The program counter of the processor is updated to advance the instruction stream to the first instruction after the embedded configuration data, and the normal processing pipeline continues execution.

As the system contains a unified virtual address space, application data is accessible directly from the programmable logic region, allowing the software to fully suspend on the primary architecture while the auxiliary architecture is in use. The running programmable logic design does not need to be controlled by software as is typical in the master-slave accelerator model. The information about the current process memory address space (stack) is passed to the programmable logic via a register exposed over the AXI bus. This mechanism thus enables the explicit single-context control flow that is visible to the application.

6.4 Process scheduling and resource allocation

In modern operating systems, multitasking enables the simultaneous execution of multiple processes by interleaving execution on available processing elements. Processor cycles are allocated among various processes and threads according to policies and process priorities. A system timer periodically triggers a scheduler to select an appropriate task for execution. Taking into account the fact that tasks may exhibit different behaviours and have varying characteristics, these processes are typically grouped into categories according to their assigned priority such as real-time, batch or interactive.

The Completely Fair Scheduler (CFS) [Mol07, Pab09], the current state-of-the-art scheduler implemented in the linux kernel, is modelled after a perfectly fair CPU: if two programs are running simultaneously, they both run at 50% of the CPU power at the same time. This scheduling algorithm is based on the principle of weighted fair queuing (WFQ). A time-ordered red-black tree [Meh84] - a self-balancing binary tree with $\mathcal{O}(\log n)$ search, insert and delete performance - is used to maintain a list of runnable tasks on a given processor (the *run-queue*). Figure 6.6 shows an example of such a runqueue for a number of active tasks, indexed by their *virtual runtime* - the weighted time that task has spent on the processor.

The dynamic time quanta (timeslice) is calculated for the selected task before it is scheduled for execution as $timeslice = period \times \frac{task\_load}{cfs\_runqueue\_load}$ [KS09]. This value is dependent on the number and priority of the existing tasks in the runqueue ($cfs\_runqueue\_load$, the weight of the fair queue) as well as the previous load of the task ($task\_load$).

Under normal operation, process scheduling will occur under the following conditions:

(a) a process switches to the waiting state from the running state (e.g. by making an I/O request).

(b) a process terminates.

(c) a process switches to the ready state from the running state (e.g. in response to an interrupt).

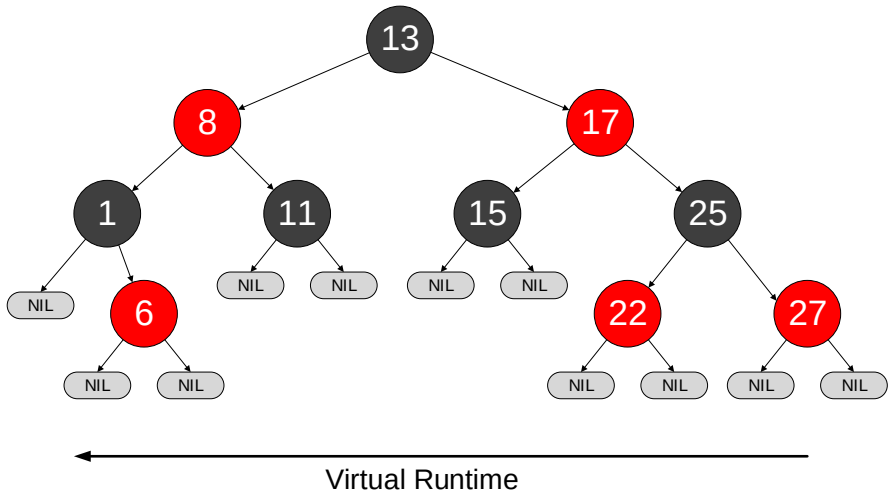

FIGURE 6.6: Tasks indexed by their Virtual Runtime in a CFS run-queue

(d) a process switches to the ready state from the waiting state (e.g. completion of I/O).

Conditions (a) and (b) will always result in the activation of a new process for execution resulting in a context switch, whereas conditions (c) and (d) may only cause a context switch in a *preemptive* system.

This task will continue to execute until one of the following conditions occur on a scheduler tick:

- the allocated timeslice to the task has been used
- a task with a lower virtual runtime in the run-queue
- a new task is created (which would be assigned the minimum current virtual runtime)

The CFS scheduler will either select the task that has just become runnable, or that with the smallest value of the virtual runtime (the left-most node of the red-black tree). This operation can be executed in constant time due to the use of the red-black tree data structure. When the scheduler performs a context switch, it will update the virtual runtime according to the actual time the task executed on the processor and the current system load. This task is then inserted back into the run-queue. In this way, the scheduler is able to divide the computational resources fairly due to the selection process that is based on the normalized virtual runtime of the tasks, resulting in every task being executed once per epoch.

The CFS scheduler behaviour is controlled through the `sched_granularity_ns` parameter which affects the minimum period between scheduler ticks. This allows it to be

tuned from low-latency (interactive) to batch workloads. Unlike most other scheduling algorithms, CFS remains independent of the speed of the underlying processor.

6.4.1 Mixed-Architecture Process Scheduling

A simplified scheduling model is enabled by the MURAC model as the operating system scheduler does not need to be aware of the underlying execution architecture. Instead, it is able to just view the whole application as a single context task. The hardware/software boundary is captured entirely within a single process due to the abstraction that permits the user application to execute on multiple system architectures within a single process context.

In this mixed-architecture system the main CPU kernel scheduler is used to control the overall system scheduling. It is aided by a programmable logic specific scheduler extension running in the kernel module that has been designed to handle mixed-architecture processes containing programmable logic. Figure 6.7 shows the high level operation of the mixed-architecture scheduler during a scheduler tick.

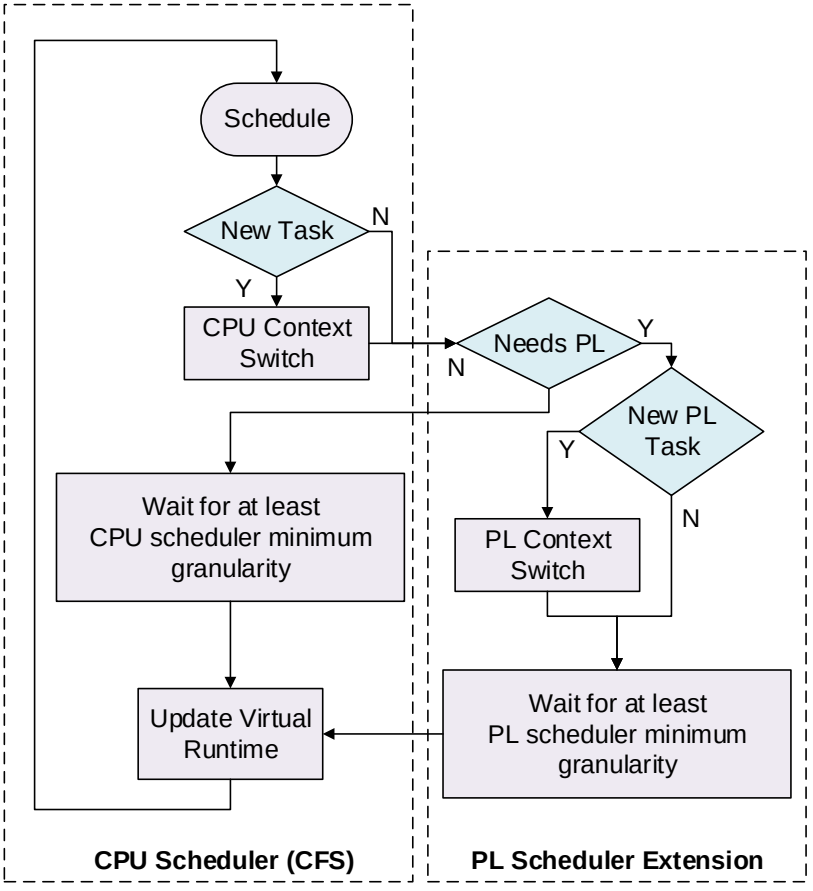

FIGURE 6.7: Mixed-Architecture Process Scheduler

In practice, due to the physical implementation that uses a separate reconfigurable region from the processor, the resource allocation of the system is optimized to ensure

better performance. For example, the operating system may not remove an actively running task on the programmable logic region if the next task does not require the programmable logic fabric during it's respective scheduled runtime.

The two strategies that are considered when allocating the programmable logic to actively executing tasks are:

- **Blocking**, whereby a process is blocked until the programmable logic region becomes available. While waiting, the process yields it's timeslice, making it more likely to be scheduled subsequently and receive a comparable share of the processor and programmable logic when it eventually needs it (so called *sleeper fairness*).
- **Preemption**, whereby a process will actively preempt an exiting running programmable logic region task, forcing a programmable logic context switch to take place.

The differences between these two strategies is illustrated through the example of two simultaneously executing processes in Figure 6.8.

Due to the much higher dispatch latency of the context switch operation for an application running in the programmable logic region, the concept of a *minimum PL schedule granularity* (`pl_sched_granularity_ms`) is used in combination with the *minimum CPU scheduler granularity* (`sched_granularity_ms`). This parameter is needed to avoid thrashing as it ensures that a programmable logic task will be allowed to run for a minimum period of time before it can be preempted. This parameter will also have an effect on the subsequent scheduling of the task as this running time will be accounted for in the virtual runtime of the task. In this way, execution on the programmable logic region is effectively modelled as a CPU-bound process from the perspective of the operating system.

Using a preemptive scheduling strategy requires the ability to suspend the execution of an ongoing task and to restore a previously interrupted task. These save and restore operations are performed when a software (process/thread) context switch occurs in the operating system scheduler. They are then carried out by the programmable logic specific scheduler extension that is implemented in the kernel module. The execution state of an application currently running on the programmable logic region is suspended and saved for later continuation before being removed to allow for the state of the next task to be configured in the programmable logic. Alternatively, the need for this saving of the task state is eliminated when the scheduler uses a blocking strategy because the next task will only be scheduled upon completion of the current task.

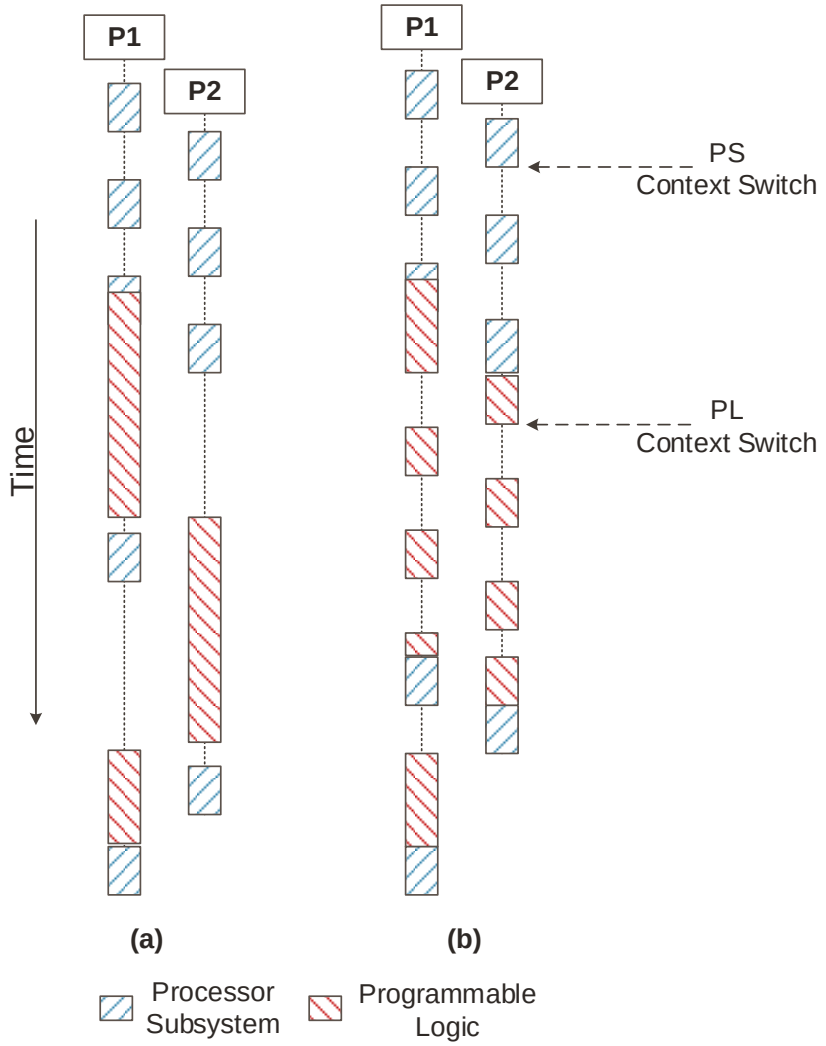

FIGURE 6.8: Comparison of (a) blocking and (b) preemption scheduling strategies for two concurrently executing mixed-architecture applications

6.4.2 Scheduler implementation

The context switch save operation (that occurs within the 'PL Context Switch' block in Figure 6.7) is implemented as a programmable logic read-back operation carried out by the kernel module. This is achieved writing commands to the configuration access port on the Zynq programmable logic region, namely `capture` (IOBs and CLB contents) and `read-back`. Once this operation has been performed, the data that has been saved during the read-back command represents the current state of the suspended task and contains the programmable logic region configuration frames and the values of all the CLB and IOBs. The activation of a previously suspended programmable logic task is carried out by using this saved configuration data to reconfigure the programmable logic region and restore the task state.

Table 6.1 illustrates the programmable logic context switch latency for various sized

partial configuration regions in the implementation via the Internal Configuration Access Port (ICAP) on the Zynq processor. With this in mind, we have run tests using a fairly small partial bitstream (219776 bytes), which leads to a context switch latency of 45ms within the system. For each test, 10 mixed-architecture processes were executed concurrently using the synthetic workload application described previously.

Partial Region Size	PL Utilization	Context Switch Latency
58176 bytes	1,46%	14 ms
219776 bytes	5,47%	45 ms
1056864 bytes	26,23%	197 ms

TABLE 6.1: Context Switch Latency

6.4.3 Scheduler strategies

The synthetic workload application described in Section 6.2 has been implemented to simulate a variable length runtime mixed-architecture process. This application was run under the various conditions described in Table 6.2. The `sched_granularity_ms` and `pl_sched_granularity_ms` are varied to illustrate the effect on scheduler performance based on the workload by controlling the minimum CPU task scheduler runtime and minimum programmable logic task scheduler runtime respectively. In these tests, the processes are all run with the `SCHED_NORMAL` process priority class.

As a reference, the standard linux kernel CFS scheduler is tuned for a better interactive workload performance by using a default value of 1.5ms for the `sched_granularity_ms`.

Scenario	`sched_granularity_ms`	`pl_sched_granularity_ms`
BLOCKING	2	-
PREEMPT(Short PS, Short PL)	2	2
PREEMPT(Short PS, Medium PL)	2	50
PREEMPT(Short PS, Large PL)	2	120
PREEMPT(Medium PS, Medium PS)	50	50
PREEMPT(Medium PS, Large PL)	50	120

TABLE 6.2: Programmable Logic Allocation parameters

For each of these strategies, a different workload was selected to compare the various performance profiles:

- **Equal granularity long-running processes** whereby all processes were tuned to execute a long running PL task. These results are illustrated in Figure 6.9, showing the total runtime to complete all processes combined, as well as an indication of the combined scheduler overhead experienced. Figure 6.10 illustrates the minimum, maximum and average runtime of these processes. A comparison of the number of context switches performed by the scheduler can be seen in Figure 6.15

- **Equal granularity short-running processes** whereby all processes were tuned to execute a very short running PL task. These results are illustrated in Figure 6.11, showing the total runtime to complete all processes combined, as well as an indication of the combined scheduler overhead experienced. Figure 6.12 illustrates the minimum, maximum and average runtime of these processes. A comparison of the number of context switches performed by the scheduler can be seen in Figure 6.16
- **Mixed granularity processes** whereby half of the processes were tuned to execute a long running PL task, and the other half were tuned to execute a very short running PL task. These results are illustrated in Figure 6.13, showing the total runtime to complete all processes combined, as well as an indication of the combined scheduler overhead experienced. Figure 6.14 illustrates the minimum, maximum and average runtime of these processes. A comparison of the number of context switches performed by the scheduler can be seen in Figure 6.17.

Based on the results presented in Section 6.A, it can be seen that the **blocking** strategy provided better overall performance for tasks of equal granularity, as it minimizes the turnaround time. This indicates that blocking allocation is better for batch processing workloads that are computationally intensive as it minimizes the expensive scheduler latency which provides better throughput. However, when running a mixed granularity workload, the **preemption** strategy outperforms the blocking strategy, as it ensures more fairness for the shorter running tasks. It can be observed that the scheduler performs best when the `pl_sched_granularity_ms` is higher than the dispatch latency of the context switch for these workloads.

As programmable logic devices become larger and applications grow to make use these much larger areas, the high dispatch latency of this simple scheduler will have an adverse effect on the fairness of the system as a whole. The dispatcher needs to be as fast as possible, as it is run on every context switch operation. More comprehensive techniques that hide this latency, such as speculative preloading and task relocation, will be required to mitigate this issue. These techniques have not been explored further in this work as the scheduler implementation has only served to demonstrate the role that the MURAC model plays in enabling a multitasking runtime environment on a heterogeneous machine.

6.5 Chapter Summary

The benefits of the MURAC model towards enabling a runtime environment in a multitasking heterogeneous system have been explored in this chapter. This has lead to the design of a reconfigurable instruction set processor (RISP) using the programmable logic

fabric of the tightly-coupled platform. The system-level implementation of the MURAC model in an operating system kernel module is presented. This demonstates that the unified machine model abstraction can be emulated in within the software layer of an operating system without the need for any hardware modifications.

The consistent unified machine model has enabled a simple extension of the operating system scheduler to support mixed-architecture programs. This implementation includes an extension to the CPU scheduler in the kernel that allows the hardware/software interface to be abstracted into a single process model. Under this unified runtime environment, *blocking* and *preemption* scheduling strategies have been implemented and tested using a mixed-architecture application to simulate variable length computational workloads within a single process context.

Due to the relatively long configuration time of the programmable logic, the fairness of the system has shown to be sensitive to the scheduler parameters that determine the minimum interval between scheduling decisions on the CPU and programmable logic. The results of this experiment indicate that the scheduler should implement a minimum runtime granularity that is much higher than this dispatch latency in order to effectively perform in a multitasking environment.

6.A Scheduler Test Results

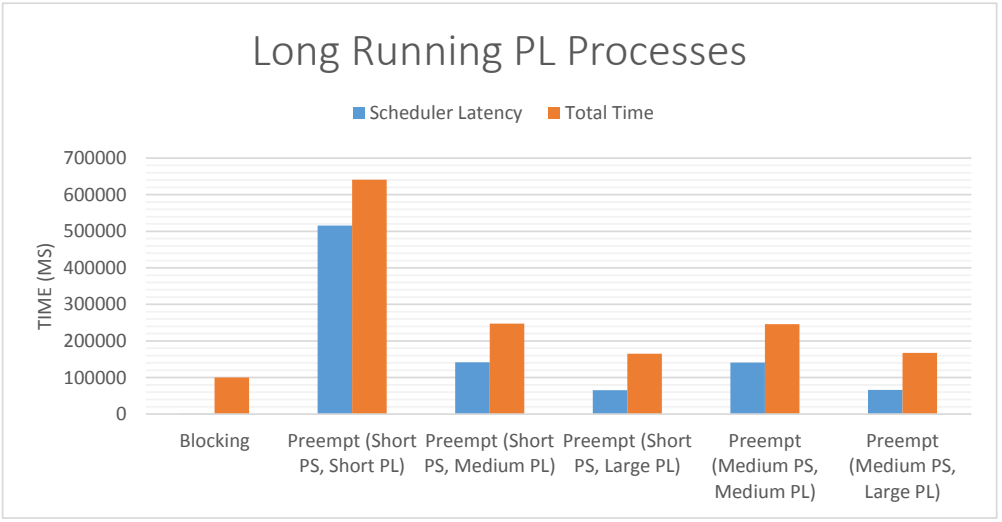

FIGURE 6.9: Simultaneous equal granularity long-running processes

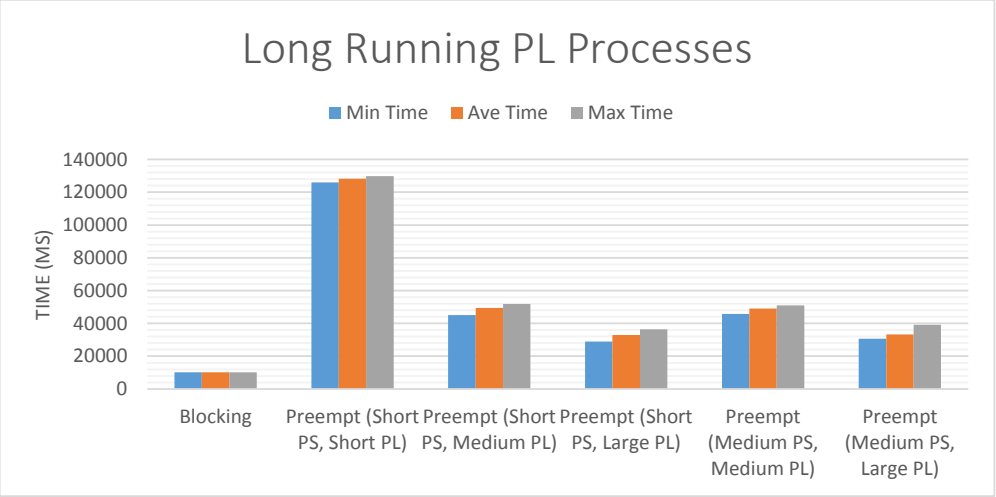

FIGURE 6.10: Simultaneous equal granularity long-running processes

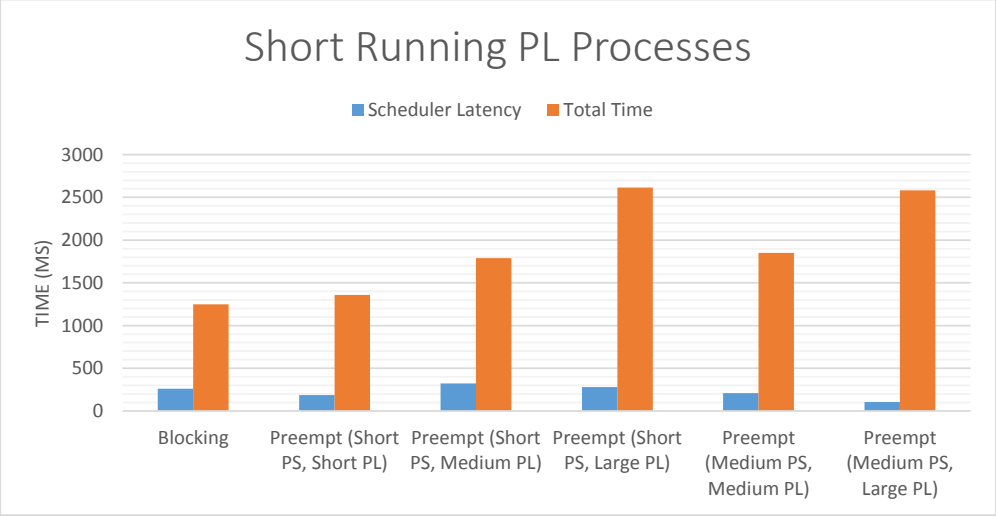

FIGURE 6.11: Simultaneous equal granularity short-running processes

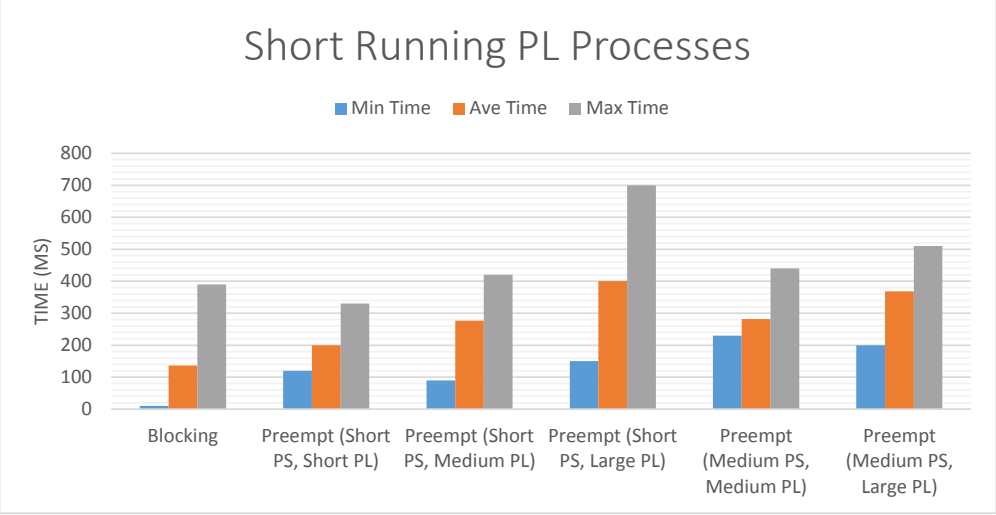

FIGURE 6.12: Simultaneous equal granularity short-running processes

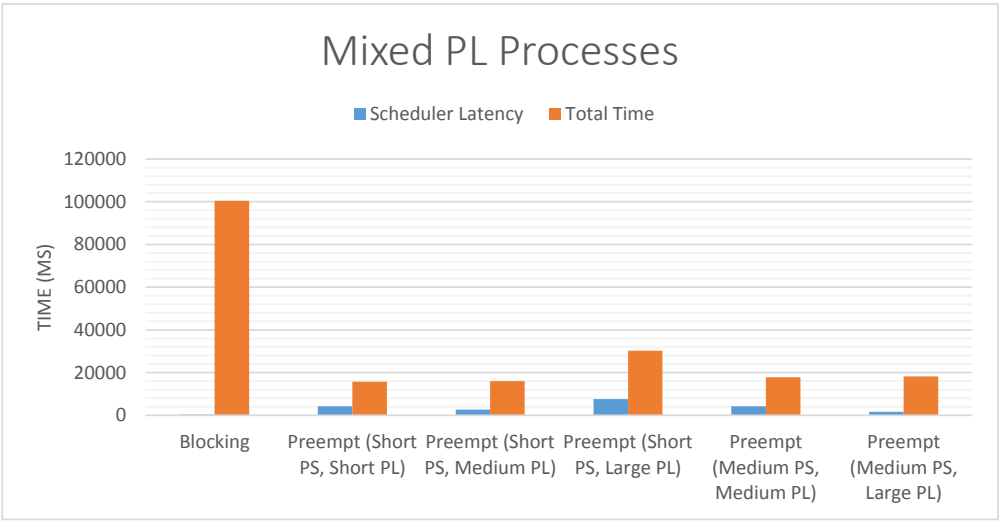

FIGURE 6.13: Simultaneous mixed granularity processes

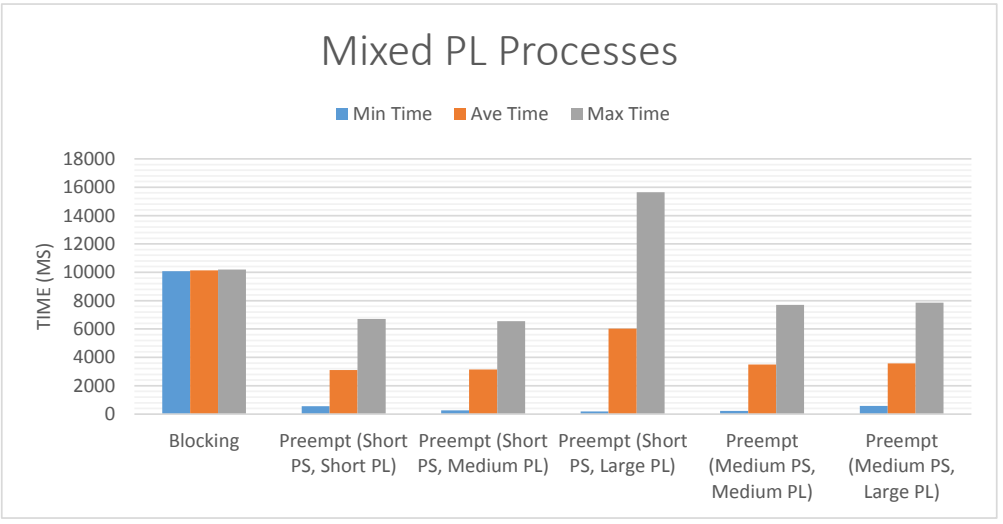

FIGURE 6.14: Simultaneous mixed granularity processes

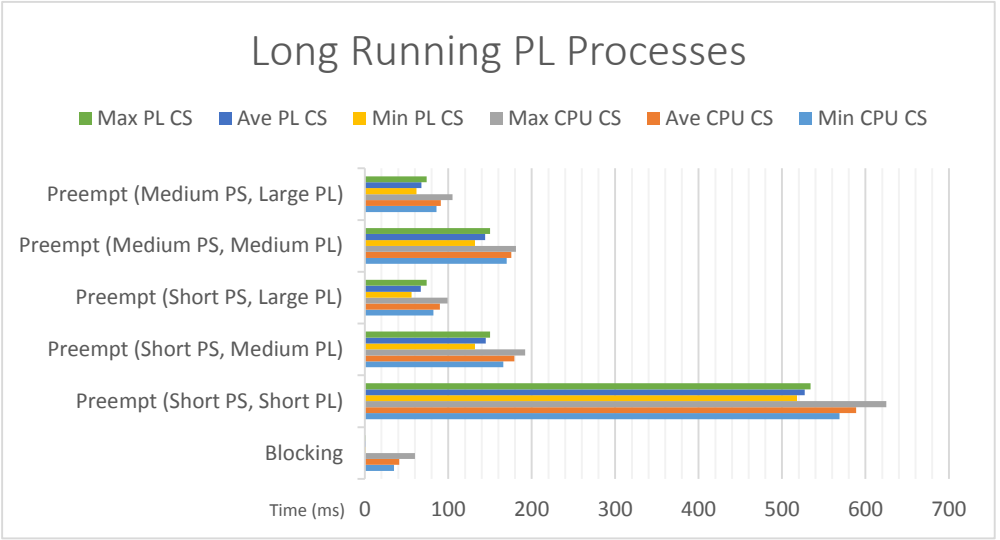

FIGURE 6.15: Context-switch latency for equal granularity long-running processes

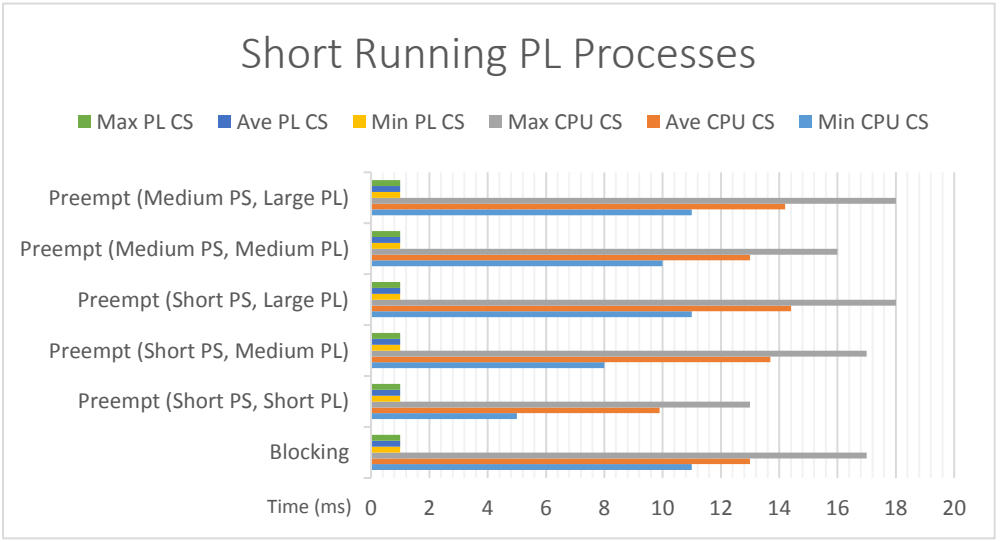

FIGURE 6.16: Context-switch latency for equal granularity short-running processes

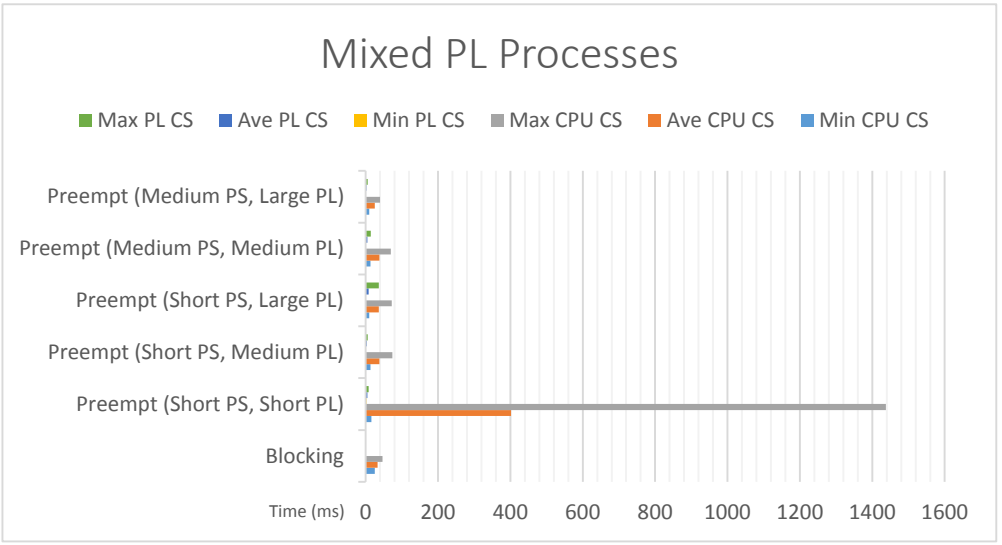

FIGURE 6.17: Context-switch latency for mixed granularity processes

Chapter 7

Conclusions

The Multiple Runtime-reconfigurable Architecture Computer (MURAC) model has been proposed to tackle the problems commonly found in the design and usage of heterogeneous machines. This model presents a system-level approach that allows the programming model to become agnostic of both the operating system and the programming language. By creating a clear separation of concerns between the system implementer and the application developer, MURAC allows each to focus on their area of expertise, which greatly reduces the *complexity of design*. The system implementer will be most familiar with the hardware and low-level design intricacies while application designers can focus implementing the computational algorithms. This concept affords even a relatively new developer the ability to leverage a well-implemented system and achieve substantial performance gains that would otherwise require an enormous effort on their part.

The concepts of the *unified machine model*, *unified instruction stream* and a *unified memory space* are the key components of the MURAC model. Within the unified machine model a heterogeneous computer is viewed as a collection of processing architectures distinguished by their model of computation. This model defines operations at the instruction set architecture (ISA) level for allowing applications to migrate between these architectures as needed during runtime. Through the unified instruction stream, a single application is able to dynamically morph the underlying architecture to carry out computation in any desired computation model supported by the machine during runtime. This leads to the concept of a mixed-architecture application that encapsulates multiple models of computation within a single process context. The abstraction of the hardware/software interface in the single-context process model provides an *overall system view of both hardware and software*. By pushing the implementation details below the abstraction layer, the complex interfacing mechanisms between multiple heterogeneous processing elements are no longer exposed to the application. This removes the reliance on the non-standard *vendor specific implementations*.

A simple programming model built upon these abstractions provides a consistent interface for interacting with the underlying machine to the user application. This programming model simplifies application partitioning between hardware and software and allows the easy integration of different execution models within the single control flow of the application. By enabling this consistent interface between the partitions of the application, the programming model encourages composability. This also promotes design *re-usability* as the loosely-coupled application partitions become independent of the system specific communication and synchronisation details between the underlying architectures. These applications are now capable of running across multiple diverse machine configurations making them *portable*. These features also encourage *collaboration* and a greater possibility of *community involvement*. By implementing the MURAC model in a wide range of systems, they can gain the benefit of conforming to a standard abstraction model similar to that found in the software development community. This would further benefit the community by increasing accessibility to developers with a wider range of skills and removing the barriers to entry created by high development costs.

7.1 Research objectives

A unified abstraction model for heterogeneous computers will provide important usability benefits to system and application designers. This abstraction enables a consistent view of the multiple different architectures to the operating system and user applications. This allows them to focus on achieving performance and efficiency goals by gaining the benefits of different execution models during runtime without the complex implementation details of the system-level synchronisation and coordination.

The achievements of the work presented in this thesis are evaluated through an examination of the objectives set to validate the hypothesis that provides the guiding principle behind this project.

High-level design methodology By focussing on the easy integration of heterogeneous computing components, MURAC has created an abstraction to the hardware/software interface complexities that are dealt with by designers and users of heterogeneous systems. The MURAC model has separated the programming model from the system implementation, illustrating a high-level methodology that provides multiple benefits to system and application designers by enabling a simple and familiar environment to explore heterogeneous system design. The system implementer is able to provide an optimized system design by focussing on providing the capabilities of the MURAC model. This system-level design has been illustrated through the implementations of the simulator, System-on-Chip and RISP systems presented in this work. The application designer

is able to compose multiple computational models together by using the simple high-level programming model that avoids any system-specific details. With a particular emphasis on ease-of-use and portability, the programming model has freed the developer from the large effort required in coordinating multiple diverse computational architectures within a single application. As these hybrid systems are increasingly adopted, a consistent abstraction will play an important role in supporting the users in being able to efficiently take advantage of the benefits provided by these systems.

Theoretical and practical trade-offs Performance and power efficiency are typically the key factors driving heterogeneous systems design. It is important for any model of these systems to enable the designer to optimize with respect to these parameters. The principle of Amdahl's Law [HM08] must be kept in mind - the speed-up of a program using multiple processors in parallel is limited by the time needed for the sequential portion of the program. Using the MURAC model, the programmer is able to focus on exploiting areas of parallelism within the application while delegating the normally sequential co-ordination and communication between these areas to an optimized machine beneath an abstraction layer. However, it is important to take into account the added overheads that are brought into the system whenever generality and abstractions are employed. This takes the form of additional translation mechanisms normally not required in a whole system optimization approach.

The theoretical and practical implementation considerations necessary to support the concepts of the MURAC model have been discussed in Chapter 3. Along with the designs presented in Chapter 4, Chapter 5 and Chapter 6, the practical implementation details are exposed through the system-level implementation of the MURAC model in these systems.

Real-world system design As a validation of the MURAC model, several real-world systems have been designed and implemented using commercial off-the-shelf hardware. The full system simulator has presented the design of a reconfigurable instruction set processor capable of executing mixed-architecture application with arbitrary processing architectures in Chapter 4. A design exploration has led to the development of an application specific instruction processor on an embedded System-on-Chip that is presented in Chapter 5. This system has been implemented on a standard COTS FPGA platform to investigate the hardware implementation overhead of supporting the MURAC model. Finally, Chapter 6 presents a full multitasking heterogeneous system that is realized in an operating system kernel module running on the tightly coupled Zynq platform. This system illustrates the design of a reconfigurable instruction set processor supporting the MURAC abstraction.

.

Real-world application design The design of applications for each of the systems presented in this work has demonstrated the real-world applicability of the model. Applications that benefit from multiple models of computation have been implemented using the MURAC model, particularly in the cryptographic (Listing 4.2 and Listing 5.1) and sequence alignment (Listing 4.3) domains. In addition, a synthetic workload application (Listing 6.1) is used to evaluate the operating system scheduler in Chapter 6. The system-level implementation of the systems developed in this work have been validated through these applications. A comparison of these applications shows that the programming model is highly invariant under the vastly different platforms and underlying systems, which demonstrates the consistent abstraction and portability enabled by the model.

Operating system support A study of the interaction between the operating system and the underlying machine has led to the design of a reconfigurable instruction set processor presented in Chapter 6. By targeting a commercial off-the-shelf (COTS) tightly-coupled reconfigurable platform, the MURAC model capability has been implemented in the software layer as an operating system kernel module. Through the single context process view provided by the MURAC model, this module enables the operating system to execute mixed-architecture applications within the standard process model. This has given users the ability to run applications across multiple architectures within a familiar operating system environment.

Multitasking environment scheduler Extending the runtime support provided by the MURAC model through the operating system extension allows for the implementation of a consistent multitasking environment on a heterogeneous machine. The CPU scheduler has been extended to become aware of the mixed-architecture nature of the processes. This simple extension has been made possible by the consistent environment gained through the principles of the MURAC model. As a demonstration of this environment, an evaluation of different scheduling strategies employed within this scheduler has been performed. This capability has enabled applications to gain the benefits of different execution models during runtime within a familiar multitasking operating system environment.

The MURAC model may serve as a foundation that will provide useful support to many of the tools and techniques available for heterogeneous system design. For example, the abstraction of the hardware/software interface can provide a target for compilers and software libraries and frameworks. Techniques such as pipe-lining and instruction analysis can benefit from the consistent interface and single-context model to produce efficient programs that will be able to provide high-performance applications that are able to fully take advantage of the underlying system architecture, without the need for

the application developer to worry about the low-level details of the hardware/software interface.

The use of this consistent model enables programmers to better access the available potential of the system, allowing reduced cost and time to market through the application of familiar modularisation and reusable design best practices. This model has enabled an increase in the user's productivity when working with heterogeneous systems. Through the unified machine model, MURAC enables new paradigms in heterogeneous computing where multiple heterogeneous accelerators may be employed to accelerate the same application. As the benefits of heterogeneity become increasingly important, future processors will be able to tightly couple multiple heterogeneous processors on the same die. It is anticipated that the MURAC model will also be able to serve as a guiding principle for this breed of heterogeneous processors of the future.

Bibliography

[ABC+06] K Asanovic, R Bodik, B C Catanzaro, J J Gebis, P Husbands, K Keutzer, D A Patterson, W L Plishker, J Shalf, and S W Williams. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, December 2006.

[ABCR10] Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA '10: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, page 89, New York, New York, USA, October 2010. ACM Request Permissions.

[AFP+11] Michael Adler, Kermin E Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *FPGA '11: Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, page 25, New York, New York, USA, February 2011. ACM Request Permissions.

[Agi09] Agility Design Solutions Inc. Mentor Graphics Corporation. *Handel-C Language Reference Manual*, 2009.

[AHK+14] A Agne, M Happe, A Keller, E Lubbers, B Plattner, M Platzner, and C Plessl. ReconOS: An Operating System Approach for Reconfigurable Computing. *Micro, IEEE*, 34(1):60–71, 2014.

[AK01] Perry Alexander and Cindy Kong. Rosetta: Semantic Support for Model-Centered Systems-Level Design. *Computer*, 34(11):64–70, November 2001.

[APA+06] Jason Agron, Wesley Peck, Erik K Anderson, David Andrews, Ed Komp, Ron Sass, Fabrice Baijot, and Jim Stevens. Run-Time Services for Hybrid CPU/FPGA Systems on Chip. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 3–12. IEEE Computer Society, December 2006.

[AS93] P M Athanas and H F Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, 1993.

[BBKG07] Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi N Gaydadjiev. Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[BCZ90] J K Bennett, J B Carter, and W Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, New York, New York, USA, February 1990. ACM Request Permissions.

[BGSH12] L Bauer, A Grudnitsky, M Shafique, and J Henkel. PATS: A Performance Aware Task Scheduler for Runtime Reconfigurable Processors. *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 208–215, 2012.

[BHLM94] Joseph Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation*, April 1994.

[BLD02] Francisco Barat, Rudy Lauwereins, and Geert Deconinck. Reconfigurable Instruction Set Processors from a Hardware/Software Perspective. *IEEE Transactions on Software Engineering*, 28(9):847–862, September 2002.

[BMCB14] Paolo Burgio, Andrea Marongiu, Philippe Coussy, and Luca Benini. A HLS-Based Toolflow to Design Next-Generation Heterogeneous Many-Core Platforms with Shared Memory. In *EUC '14: Proceedings of the 2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*, pages 130–137. IEEE Computer Society, August 2014.

[Bol04] M Bolstad. Design by contract: a simple technique for improving the quality of software. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 303–307. IEEE, 2004.

[BPBL06] P Bellens, J M Perez, R M Badia, and J Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 5–5. IEEE, 2006.

[Bre10] T M Brewer. Instruction Set Innovations for the Convey HC-1 Computer. *Micro, IEEE*, 30(2):70–79, 2010.

[BRS13] David F Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Communications of the ACM*, 56(4):56–63, April 2013.

[BSB+01] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, and Richard F Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, June 2001.

[BSC08] F A Bower, D J Sorin, and L P Cox. The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling. *Micro, IEEE*, 28(3):17–25, 2008.

[BSG10] Anthony Brandon, I Sourdis, and Georgi N Gaydadjiev. General Purpose Computing with Reconfigurable Acceleration. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 588–591. IEEE, 2010.

[BSH08] L Bauer, M Shafique, and J Henkel. Efficient Resource Utilization for an Extensible Processor Through Dynamic Instruction Set Adaptation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(10):1295–1308, 2008.

[BSH09] Surendra Byna, Xian-He Sun, and Don Holmgren. Modeling Data Access Contention in Multicore Architectures. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 213–219. IEEE, 2009.

[BTL10] B Betkaoui, D B Thomas, and W Luk. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 94–101. IEEE, 2010.

[BVCG04] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. *ACM SIGPLAN Notices*, 38(5):14–26, December 2004.

[CC01] Jorge E Carrillo and Paul Chow. *The effect of reconfigurable units in superscalar processors*. ACM, New York, New York, USA, February 2001.

[CCA+13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *Transactions on Embedded Computing Systems (TECS*, 13(2), September 2013.

[CG03] L Cai and Daniel Gajski. Transaction level modeling: an overview. In *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pages 19–24. ACM, 2003.

[Cha13] Anupam Chattopadhyay. Ingredients of adaptability: a survey of reconfigurable processors. *VLSI Design*, 2013, January 2013.

[CHW00] Timothy J Callahan, J R Hauser, and John Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, 2000.

[CLMA08] Anupam Chattopadhyay, Rainer Leupers, Heinrich Meyr, and Gerd Ascheid. Language-driven Exploration and Implementation of Partially Reconfigurable ASIPs, 1st edition. *Language-driven Exploration and Implementation of Partially Re-configurable ASIPs, 1st edition*, December 2008.

[CTM04] A Cheng, G Tyson, and T Mudge. FITS: framework-based instruction-set tuning synthesis for embedded application specific processors. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 920–923, 2004.

[CUD07] Nvidia CUDA. *Compute unified device architecture programming guide*, 2007.

[CW98] Timothy J Callahan and John Wawrzynek. Instruction-level parallelism for reconfigurable computing. *... Logic and Applications From FPGAs to ...*, 1998.

[CWT+01] J D Collins, P Wang, D M Tullsen, C Hughes, Y F Lee, D Lavery, and J P Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 14–25. IEEE Comput. Soc, 2001.

[DAF11] Mayank Daga, Ashwin M Aji, and Wu-chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, pages 141–149. IEEE, 2011.

[DeH96] Andre DeHon. DPGA Utilization and Application. In *Field-Programmable Gate Arrays, 1996. FPGA '96. Proceedings of the 1996 ACM Fourth International Symposium on*, pages 115–121. IEEE, 1996.

[DeH00] Andre DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, 2000.

[DF07] F Dittmann and S Frank. Hard Real-Time Reconfiguration Port Scheduling. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, pages 1–6. IEEE, 2007.

[DSGO02] F Doucet, S Shukla, R Gupta, and M Otsuka. An environment for dynamic component composition for efficient co-design. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 736–743. IEEE, 2002.

[DY08] Gregory F Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, page 197, New York, New York, USA, June 2008. ACM Request Permissions.

[EBSA+12] H Esmaeilzadeh, E Blem, R St Amant, K Sankaralingam, and D Burger. Dark Silicon and the End of Multicore Scaling. *Micro, IEEE*, 32(3):122–134, 2012.

[ECR+10] Y Etsion, F Cabarcas, A Rico, A Ramirez, R M Badia, E Ayguade, J Labarta, and M Valero. Task Superscalar: An Out-of-Order Task Pipeline. *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 89–100, 2010.

[EGEAH+08] T El-Ghazawi, E El-Araby, Miaoqing Huang, K Gaj, V Kindratenko, and D Buell. The Promise of High-Performance Reconfigurable Computing. *Computer*, 41(2):69–76, 2008.

[Egu10] K Eguro. SIRC: An Extensible Reconfigurable Computing Communication API. *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 135–138, 2010.

[FB89] D Fernandez-Baca. Allocating modules to processors in a distributed system. *Software Engineering, IEEE Transactions on*, 15(11):1427–1436, 1989.

[FC08] Wenyin Fu and Katherine Compton. Balanced allocation of compute time in hardware-accelerated systems. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 241–248. IEEE, 2008.

[FFY05] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, January 2005.

[Fly72] M Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, September 1972.

[FYAE14] Kermin E Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. The LEAP FPGA operating system. *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.

[GDGN03] S Gupta, N Dutt, R Gupta, and A Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466. IEEE Comput. Soc, 2003.

[GHF+06] M Gschwind, H P Hofstee, B Flachs, M Hopkins, Y Watanabe, and T Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[GHN+12] V Govindaraju, Chen-Han Ho, T Nowatzki, J Chhugani, N Satish, K Sankaralingam, and Changkyu Kim. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *Micro, IEEE*, 32(5):38–51, 2012.

[GK07] Hagen Gadke and Andreas Koch. Comrade - A Compiler for Adaptive Computing Systems using a Novel Fast Speculation Technique. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 503–504, 2007.

[GMHB11] Diana Göhringer, L Meder, M Hubner, and J Becker. *Adaptive Multi-client Network-on-Chip Memory*. IEEE, 2011.

[Göh14] Diana Göhringer. Reconfigurable Multiprocessor Systems: Handling Hydras Heads – A Survey. *SIGARCH Computer Architecture News*, 42(4):39–44, December 2014.

[GPHB09] Diana Göhringer, Thomas Perschke, Michael Hübner, and Jürgen Becker. A Taxonomy of Reconfigurable Single-/Multiprocessor Systems-on-Chip. *International Journal of Reconfigurable Computing*, 2009(3):1–11, 2009.

[Hau98] Scott Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 65–74, New York, New York, USA, March 1998. ACM Request Permissions.

[HD07] Scott Hauck and Andre DeHon. Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, November 2007.

[Hea14] Steve Heath. *Microprocessor Architectures: RISC, CISC and DSP*. Elsevier, 2014.

[HGT+12] M Huebner, Diana Göhringer, C Tradowsky, J Henkel, and J Becker. Adaptive processor architecture - invited paper. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 244–251. IEEE, 2012.

[HM08] M D Hill and M R Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.

[IEE05] IEEE, 3 Park Avenue, New York, NY 10016-5997, USA. *IEEE Standard SystemC Language Reference Manual*, 2005.

[IJJ09] Ciji Isen, Lizy K John, and Eugene John. A Tale of Two Processors: Revisiting the RISC-CISC Debate. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*. Springer-Verlag, January 2009.

[JFK12] Matthew Jacobsen, Yoav Freund, and Ryan Kastner. RIFFA: A Reusable Integration Framework for FPGA Accelerators. In *FCCM '12: Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, April 2012.

[JK13] Matthew Jacobsen and Ryan Kastner. RIFFA 2.0: A reusable integration framework for FPGA accelerators. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE, 2013.

[JTE+12] K Jozwik, H Tomiyama, M Edahiro, S Honda, and H Takada. Comparison of Preemption Schemes for Partially Reconfigurable FPGAs. *Embedded Systems Letters, IEEE*, 4(2):45–48, 2012.

[JW89] N P Jouppi and D W Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *ASPLOS III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 272–282, New York, New York, USA, April 1989. ACM Request Permissions.

[KG04] Georgi K Kuzmanov and Georgi N Gaydadjiev. The MOLEN processor prototype. *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 296–299, 2004.

[KHT07] Dirk Koch, Christian Haubelt, and Jürgen Teich. *Efficient hardware checkpointing: concepts, overhead analysis, and implementation.* concepts, overhead analysis, and implementation. ACM, New York, New York, USA, February 2007.

[KJL+10] John H Kelm, Daniel R Johnson, Steven S Lumetta, S J Patel, and Matthew I Frank. A Task-Centric Memory Model for Scalable Accelerator Architectures. *Micro, IEEE*, 30(1):29–39, 2010.

[KL08] John H Kelm and Steven S Lumetta. HybridOS: runtime support for reconfigurable accelerators. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 212–221, New York, New York, USA, February 2008. ACM Request Permissions.

[KP84] Brian W Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984.

[KPSW93] A A Khokhar, V K Prasanna, M E Shaaban, and Cho-Li Wang. Heterogeneous computing: challenges and opportunities. *Computer*, 26(6):18–27, 1993.

[KS09] Jacek Kobus and Rafał Szlarski. Completely Fair Scheduler and its tuning, 2009.

[KSS+12] William V Kritikos, Andrew G Schmidt, Ron Sass, Erik K Anderson, and Matthew I Frank. Redsharc: a programming model and on-chip network for multi-core systems on a programmable chip. *International Journal of Reconfigurable Computing*, 2012, January 2012.

[Lam79] L Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *Computers, IEEE Transactions on*, (9):690–691, 1979.

[LBM+06] P Lysaght, B Blodget, J Mason, J Young, and B Bridgford. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6. IEEE, 2006.

[LCD+10] Ilia Lebedev, Shaoyi Cheng, Austin Doupnik, James Martin, Christopher Fletcher, Daniel Burke, Mingjie Lin, and John Wawrzynek. MARC: A Many-Core Approach to Reconfigurable Computing. In *2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2010)*, pages 7–12. IEEE, 2010.

[LCH00] Zhiyuan Li, Katherine Compton, and Scott Hauck. Configuration caching management techniques for reconfigurable computing. *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 22–36, 2000.

[LCKR03] J Liu, F Chow, T Kong, and R Roy. Variable instruction set architecture and its compiler support. *Computers, IEEE Transactions on*, 52(7):881–895, 2003.

[LCWM08] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. *ACM SIGARCH Computer Architecture News*, 36(1):287–296, March 2008.

[LH02] Zhiyuan Li and Scott Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*. ACM Request Permissions, February 2002.

[LHLT10] Trong-Yen Lee, Che-Cheng Hu, Li-Wen Lai, and Chia-Chun Tsai. Hardware Context-Switch Methodology for Dynamically Partially Reconfigurable Systems. *Journal of Information Science and Engineering*, 2010.

[LK00] Holger Lange and Andreas Koch. Memory Access Schemes for Configurable Processors. In *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*. Springer-Verlag, August 2000.

[LK04] Holger Lange and Andreas Koch. HW/SW Co-design by Automatic Embedding of Complex IP Cores. In *Field Programmable Logic and Application*, pages 679–689. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[LK07] Holger Lange and Andreas Koch. Design and System Level Evaluation of a High Performance Memory System for reconfigurable SoC Platforms. In *High Performance and Embedded Architecture and Compilation*, Ghent, Belgium, 2007.

[LK10] Holger Lange and Andreas Koch. Architectures and Execution Models for Hardware/Software Compilation and Their System-Level Realization. *IEEE Transactions on Computers*, 59(10):1363–1377, October 2010.

[LMSG02] TGS Liao, G Martin, S Swan, and T Grötker. System design with SystemC, 2002.

[LP07] E Lubbers and M Platzner. ReconOS: An RTOS Supporting Hard-and Software Threads. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 441–446. IEEE, 2007.

[LSLB00] M H Lee, H Singh, G Lu, and N Bagherzadeh. Design and implementation of the MorphoSys reconfigurable computing processor. *Journal of VLSI Signal Processing Systems*, 24:147–164, 2000.

[Mac01] Don MacMillen. Nimble Compiler Environment for Agile Hardware. Volume 1. Technical Report AFRL-IF-WP-TR-2002-1536, Synopsys Inc., 2001.

[Meh84] Kurt Mehlhorn. *Sorting and Searching*. springer-Verlag, Berlin, 1984.

[Mey92] B Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[MLIB08] P Mahr, C Lorchner, H Ishebabi, and C Bobda. SoC-MPI: A Flexible Message Passing Library for Multiprocessor Systems-on-Chips. In *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, pages 187–192. IEEE, 2008.

[Mol07] Ingo Molnar. Modular Scheduler Core and Completely Fair Scheduler [CFS], April 2007.

[MS11] Jiayuan Meng and Kevin Skadron. A Reconfigurable Simulator for Large-scale Heterogeneous Multicore Architectures. In *Software (ISPASS)*, pages 119–120. IEEE, 2011.

[NCS13] Ho-Cheung Ng, Yuk-Ming Choi, and Hayden Kwok-Hay So. Direct virtual memory access from FPGA for high-productivity heterogeneous computing. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 458–461, 2013.

[Olu05] Kunle Olukotun. A new approach to programming and prototyping parallel systems. In *HiPC'05: Proceedings of the 12th international conference on High Performance Computing*, pages 4–4, Berlin, Heidelberg, December 2005. Springer-Verlag.

[PAA+06] Wesley Peck, Erik K Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David Andrews. Hthreads: A Computational Model for Reconfigurable Devices. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–4. IEEE, 2006.

[Pab09] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, August 2009.

[PJC+13] Khoa Dang Pham, A K Jain, Jin Cui, S A Fahmy, and D L Maskell. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 219–226. IEEE Computer Society, 2013.

[PP84] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *the 11th annual international symposium*, pages 348–354, New York, New York, USA, 1984. ACM Press.

[RFC09] Kyle Rupnow, Wenyin Fu, and Katherine Compton. Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking. In *FCCM '09: Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE Computer Society, April 2009.

[RGG05] Mehrdad Reshadi, Bita Gorjiara, and Daniel Gajski. Utilizing Horizontal and Vertical Parallelism with a No-Instruction-Set Compiler for Custom Datapaths. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*. IEEE Computer Society, October 2005.

[RL77] C V Ramamoorthy and H F Li. Pipeline Architecture. *ACM Computing Surveys (CSUR)*, 9(1):61–102, March 1977.

[SB10] Mojtaba Sabeghi and Koen Bertels. Interfacing operating systems and polymorphic computing platforms based on the MOLEN programming paradigm. In *ISCA'10: Proceedings of the 2010 international conference on Computer Architecture*, pages 311–323, Berlin, Heidelberg, June 2010. Springer-Verlag.

[SBL+14] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *Transactions on Embedded Computing Systems (TECS*, 13(4s), April 2014.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)*, 27(3):18, August 2008.

[SG10] Craig Steffen and Gildas Genest. Nallatech In-Socket FPGA Front-Side Bus Accelerator. *Computing in Science & Engineering*, 12(2):78–83, March 2010.

[SI09] John M Scott III. Open Component Portability Infrastructure (OPENCPI). Technical Report AFRL-RI-RS-TR-2009-257, Mercury Federal Systems, Inc., November 2009.

[Sin11] Satnam Singh. Computing without processors. *Communications of the ACM*, 54(8), August 2011.

[SPM+08] M Saldana, A Patel, C Madill, D Nunes, Danyao Wang, H Styles, A Putnam, Ralph Wittig, and P Chow. MPI as an abstraction for software-hardware interaction for HPRCs. *High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on*, pages 1–10, 2008.

[STB06] Hayden Kwok-Hay So, Artem Tkachenko, and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *CODES+ISSS '06: Proceedings of*

the 4th international conference on Hardware/software codesign and system synthesis, pages 259–264, New York, New York, USA, October 2006. ACM.

[SW81] T F Smith and M S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[TEL95] D M Tullsen, S J Eggers, and H M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 392–403, 1995.

[TSB+09] Martin Thuresson, Magnus Själander, Magnus Björk, Lars Svensson, Per Larsson-Edefors, and Per Stenstrom. FlexCore: Utilizing Exposed Datapath Control for Efficient Computing. *Journal of Signal Processing Systems*, 57(1):5–19, October 2009.

[Vaj11] András Vajda. *Programming Many-Core Chips*. Springer US, Boston, MA, 2011.

[Val11] Leslie G Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154–166, January 2011.

[Vas07] S Vassiliadis. *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, Berlin, 2007.

[VPI06] M Vuletic, L Pozzi, and P Ienne. Virtual memory window for application-specific reconfigurable coprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(8):910–915, 2006.

[VPNH10] J Villarreal, A Park, W Najjar, and R Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134, 2010.

[VSL08] F Vahid, Greg Stitt, and R Lysecky. Warp Processing: Dynamic Translation of Binaries to FPGA Circuits. *Computer*, 41(7):40–46, 2008.

[VYB07] A J Virginia, Y D Yankova, and KLM Bertels. An empirical comparison of ANSI-C to VHDL compilers: Spark, Roccc and DWARV. *Anual Workshop on Circuits*, 2007.

[WA10] M A Watkins and D H Albonesi. ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 497–508, 2010.

[Wal91] David W Wall. Limits of instruction-level parallelism. In *ASPLOS IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. ACM Request Permissions, April 1991.

[WAT14] Alexander Wold, A Agne, and J Torresen. Relocatable Hardware Threads in Run-Time Reconfigurable Systems. *Reconfigurable Computing: Architectures*, 2014.

[WCC+07] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. *ACM SIGPLAN Notices*, 42(6):156–166, June 2007.

[Web04] Steve Weber. *The success of open source*, volume 368. Harvard University Press, Cambridge, MA, 2004.

[WGB+04] Stephan Wong, Georgi N Gaydadjiev, Koen Bertels, Georgi K Kuzmanov, and E M Panainte. The MOLEN polymorphic processor. *Computers, IEEE Transactions on*, 53(11):1363–1375, 2004.

[WH95] M J Wirthlin and B L Hutchings. A dynamic instruction set computer. In *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pages 99–107. IEEE Comput. Soc. Press, 1995.

[WvAB08] Stephan Wong, T van As, and G Brown. ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 369–372. IEEE, 2008.

[YERJ99] A Yoaz, M Erez, R Ronen, and S Jourdan. Speculation techniques for improving load related instruction scheduling. In *International Symposium on Computer Architecture*, pages 42–53. IEEE Comput. Soc. Press, 1999.

[ZBA+05] Marvin Zelkowitz, Victor Basili, Sima Asgari, Lorin Hochstein, Jeff Hollingsworth, and Taiga Nakamura. Measuring Productivity on High Performance Computers. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05*, pages 6–6. IEEE Computer Society, September 2005.