Run-time Reconfigurable Multiprocessors

Zur Erlangung des akademischen Grades

DOKTOR-INGENIEUR (Dr.-Ing.)

der Fakultät Elektrotechnik, Informatik und Mathematik

der Universität Paderborn

genehmigte Dissertation

von

M. Sc. Madhura Purnaprajna

Paderborn

Referent: Prof. Dr.-Ing. Ulrich Rückert
Korreferent: Prof. Dr. Bertil Svensson

Tag der mündlichen Prüfung: 16.12.2009

Paderborn, den 16.01.2010

Diss. EIM-E/261

Acknowledgements

I take this opportunity to thank Prof.Dr.-Ing. Ulrich Rückert for giving me a chance to work at his research group. His support and encouragement were key to making the past four years, indeed the very best.

I am especially thankful Dr.-Ing. Mario Porrmann for his guidance and encouragement over the entire duration. A special thanks to Dr.-Ing. Markus Köster, I am glad I got in touch with him in ERSA 2004. I would also like to thank my colleagues Jens Hagemeier, Christopher Pohl, and Christoph Puttmann, for the many interesting discussions. All of these indeed lead to many motivational ideas.

I extend my gratitude to the research group of Programming Languages and Compilers, in particular, Micheal Hußmann, Michael Thies, and Prof. Uwe Kastens. A lot of my work was inspired through various discussions, lectures, and meetings with them.

I am also grateful to Prof. Bertil Svensson, who agreed review my thesis and to be a member of my examination committee. A special thanks to him for his invaluable suggestions. I also thank the members of my examination committee, Prof. Sybille Hellebrand, Prof. Uwe Kastens, Prof. Reinhold Haeb-Umbach, and Prof. Fezvi Belli.

I take this opportunity to thank my mentor Prof. Dr.-Ing Beate Meffert for the continuous encouragement, especially during the last phase of my PhD program. The Mentoring Program at the University of Paderborn headed by Prof. Scharlau was important in bringing in an idea of 'work-life' balance in my 'work-only' life. My gratitude to Gaelle Desbordes, her regular emails were indeed very important to me.

Finally, I thank my father (Dr. V. Purnaprajna), my mother (Mrs. Bharathi Prajna), my sister (Ms. Maanasa Purnaprajna), and my husband (Dr. Prashanth Athri). All this would have been impossible without their love and support.

Madhura Purnaprajna
Paderborn, Germany

Pujyaya Raghavendraya Satya Dharma Rathayacha
Bhajataam Kalpavrukshaya Namtham Kamadhenuve

Abstract

The advantage in multiprocessors is the performance speedup obtained with processor-level parallelism. Similarly, the flexibility for application-specific adaptability is the advantage in reconfigurable architectures. To benefit from both these architectures, we present a reconfigurable multiprocessor template that combines parallelism in multiprocessors and flexibility in reconfigurable architectures. A fast, single cycle, resource-efficient, run-time reconfiguration scheme accelerates customisations in the reconfigurable multiprocessor template. Based on this methodology, a four-core multiprocessor called QuadroCore has been implemented on UMC's 90nm standard cells and on Xilinx's FPGA. QuadroCore is customisable and adapts to variations in the granularity of parallelism, the amount of communication between tasks, and the frequency of synchronisation. To validate the advantages of this approach, a diverse set of applications has been mapped onto the QuadroCore multiprocessor. Experimental results show speedups in the range of 3 to 11 in comparison to a single processor. In addition, energy savings of up to 30% were noted on account of reconfiguration. Furthermore, to steer application mapping based on power considerations, an instruction-level power model has been developed. Using this model, power-driven instruction selection introduces energy savings of up to 70% in the QuadroCore multiprocessor.

Contents

Chapter 1

Introduction

In the present times of high performance demands, embedded computing is consistently overloaded with applications that call for ever increasing computational capabilities. Features that predominantly existed in the high performance computing domain have now begun to appear in hand held technologies, home appliances, entertainment systems etc. For such systems, meeting energy constraints and area restrictions is a critical prerequisite, in addition to performance demands. In this scenario the commercial success of a processor depends on a range of parameters such as design costs, design & verification time, and consequently the time-to-market, in addition to its architectural merits.

For processors, the design time involves two distinct domains – the application software and the hardware architecture. Progressive technology scaling as portrayed by Moore's law, has steadily aided performance acceleration in the hardware domain. Apart from technology scaling, tools for design and automation have assisted in reducing the time-to-market. New methods of programming, assisted by languages and higher levels of abstractions have contributed to easing software design. Languages have survived longer than processors, since the focus in software design has been on portability and reusability of legacy code. This constraint has to ensure compatibility to languages, programming, and the instruction set architectural compatibility with the next generation of hardware. As a result, each new processor generation has required designing new compilation tools for the same programming languages.

Reusability in the hardware domain has primarily relied on using the legacy instruction set architecture with additional features being included continually. To retain frontend designs, IP cores have enabled reusability and retargetability across technologies. However, the processor hardware or the backend has been re-fabricated to cope with the design updates and is a major bottleneck in the time-to-market. Building new hardware for every design change involves significant fabrication time. In this

scenario, hardware modifications that avoid design re-spins are appropriate solutions to circumvent these bottlenecks. Hardware portability and reusability necessitates development of architectural templates that provide in-field application-specific adaptability. These design reuse alternatives help in reducing the design and verification costs that together account to the high nonrecurring engineering costs.

Processor design has continuously evolved, both on account of progressive technology scaling and the continual need for high performance. With increasing computational needs, the complexity of the processor architecture has gone up from 29,000 transistors, 10 MHz, in Intel's 8086 (in the 70's) to a 731 million transistors, 3 GHz, in an Intel's QuadCore. However, these architectures still confine to the 'von Neumann' style of processing with a load-store processing model. Distinctly different from the 'von Neumann' model, are the parallel reconfigurable fabrics in FPGAs (Field Programmable Gate Arrays). Processor design is being saturated with increased frequency of operation but lower returns in performance and scalability. On the other hand FPGA-based designs have enormous advantages to cope with design changes and flexibility, but are inefficient for area, time, and power requirements. To cater to these two distinct architectural domains, this thesis presents a unified design template that merges parallelism at processor granularity with run-time reconfigurability to ease application-specific adaptability. Both these features together aim at reducing design & verification time, while retaining code portability. Run-time reconfigurability has been introduced as a feature for application-specific adaptability. Conventionally, reconfigurability enables a variety of implementation options for processor-based designs. For example, a reconfigurable device is used as a co-processor, an additional functional unit, or to provide resources to enhance the instruction set architecture. In contrast, application-specific integrated circuits have fixed architectures and are dedicated for a single application. Introducing flexibility within a processor or an ASIC often results in a loss in performance, in the absence of flexibility, a compute element is limited in use to a single application domain. Hence optimal performance in a fixed architecture is limited to a particular application. This restricts the architecture's range of usability. Thus, the goal of introducing reconfigurability is a compromise between architectural rigidity and application domain extensibility.

The focus of this thesis is to tackle the existing performance disadvantage in processor designs at one end and to address the overhead of reconfiguration (time, area, power) in the present day FPGA architectures. The aim is to enhance reusability of existing processors in order to avoid architectural redesigns without compromising on performance. In doing so, it merges the advantage of exploring parallelism by using multiple processors and introduces adaptability via run-time reconfiguration.

1.1 Contributions

Parallelism and adaptability are two distinct architectural design considerations in embedded processors. Parallelism in multi-core processors contributes to application acceleration. Adaptability via run-time reconfiguration provides application-specific customisations long after fabrication. To benefit from both these features, a reconfigurable multiprocessor architecture — ***QuadroCore*** has been developed. A novel reconfiguration mechanism has been incorporated that provides fast run-time adaptability in our 4-processor QuadroCore.

With the aim of redefining the application-to-architecture mapping paradigms, the main contributions in this thesis are as follows:

- A bottom-up design methodology is presented that introduces architectural flexibility to ease parallel programming and application-specific customisation. This is in contrast to top-down methodologies, where the primary focus is to ease programmability and code-portability. We aim at reducing design, verification, and fabrication costs of the target architecture in addition to retaining the application code (application description)
- From an application perspective, it addresses redefining applications to map onto customisable architectures and from an architectural perspective enables adapting the architecture to suit application characteristics.

In order to meet diverse design constraints in embedded processing, following are the contributions that address the performance objectives:

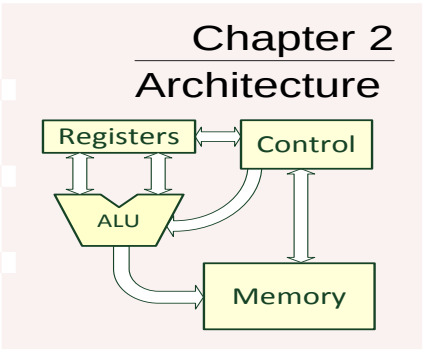
- Introduction of power as a criterion for application-specific instruction selection, scheduling, and resource allocation. This is in addition to the classical approach with time as a criterion.
- Provide an early performance feedback in multiprocessor designs, via feedback-driven application mapping.

Application diversity has been addressed with experiments in the following application domains:

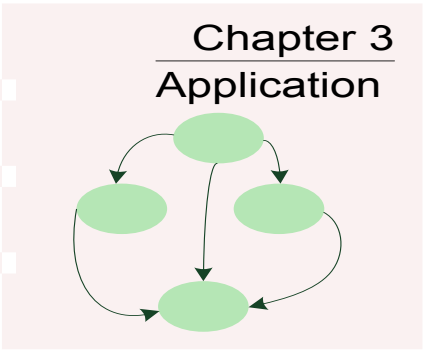
- Computation acceleration in DSP algorithms
- Power savings in a multiplier used in Elliptic Curve Cryptography (ECC)
- Energy-efficiency in a neural network based machine-learning algorithm called Self-organising Maps (SOM)

Additionally, a framework has been developed to ease architectural explorations and to introduce application-specific processor customisations.

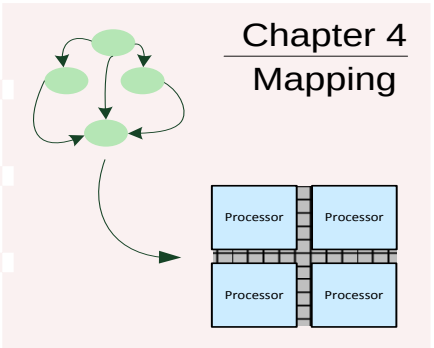
1.2 Organisation



Chapter 2 addresses the architectural considerations in processor design. Since the primary focus is on reusability, existing multiprocessor architectures, reconfigurable architectures, and methods of architectural explorations are studied. The focus is to understand their individual costs and benefits. Further, methods of introducing architectural flexibility and their corresponding impact on performance has been detailed. Finally, an overview of our reconfigurable multiprocessor template is presented that combines the advantages of existing multiprocessors and reconfigurable architectures.

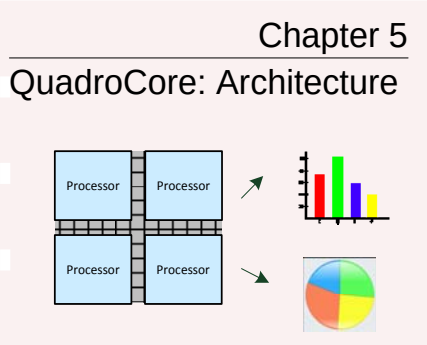


Application description complements architectural design. Chapter 3 addresses details of application design and surveys existing methods of application description. Application-level characteristics play an important role in performance of a given processor. In this chapter, a detailed analysis on the diversity in applications is modelled with respect to time and power. In order to meet performance demands, it is necessary to match the application's characteristics to the architectural features. With this perspective, three diverse sets of applications have been studied and their comparative performance characteristics are presented. Additionally, energy analysis are made by applying Amdahl's law to the time and power models developed in this chapter.

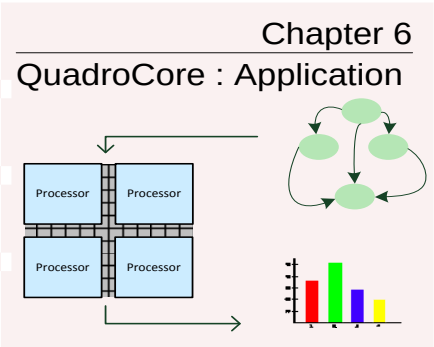


Application-to-architectural mapping binds application description to architectural constraints. Chapter 4 is dedicated to addressing the diversity in methods of mapping. Application and architectural variations give rise to a classification that identifies four sub-groups. Using this classification, the location of our QuadroCore multiprocessor is identified. Using the four quadrants of classification, mapping techniques in architectures that can be modified during run-time, viz., processors and FPGAs are studied closely. The similarities and diversities in the mapping techniques for processors and FPGA architectures are discussed in detailed. Since, the focus is to merge these two

diversities — a unified approach to merge spatial and temporal execution models is presented. Here, reconfiguration has been introduced as a method to steer run-time application mapping in order to enhance resource efficiency.



The concepts of multiprocessing and reconfigurability have been implemented in our reconfigurable multiprocessor – *QuadroCore*. Chapter 5 presents the details of the architecture, the reconfiguration mechanism, and the resulting extended reconfiguration design space. This architecture has been realised in UMC's 90nm standard cell technology and on Xilinx's FPGA. The performance details for these implementations are discussed in this chapter. Further, the instruction set architecture has been characterised in terms of power and an instruction-level power model has been developed for the base processor. The timing and power characterisation aids in application mapping with two mutually opposing performance objectives. Finally, detailed performance reports are presented to analyse the performance impact of introducing each of the proposed reconfigurable modes.



In Chapter 6 the performance impact on mapping three diverse applications on QuadroCore has been detailed. DSP algorithms, a computationally complex multiplier used in Elliptic Curve Cryptography, and a neural network-based machine learning algorithm are analysed with resource efficiency as the primary objective of application mapping. The reconfigurable capabilities make the architecture well-suited for resource efficient application mapping. In addition, the adaptable nature of the architecture enables it to be used within a framework for processor-specific run-time customisations and architectural validations. The quick reconfiguration mechanism introduces these architectural variations. Finally, the architecture can be used to validate parallel programs, which are independent of the target implementation platform. The in-built communication and synchronisation mechanisms in QuadroCore aid in executing parallel programs on this hardware platform for accelerated verifications.

Finally, Chapter 7 summarises the proposed concepts, methods, applications, and implementation reports for run-time reconfigurable multiprocessors. A section on future work presents directions for extensions and further research.

Chapter 2

Architecture

A framework of resources and their interactions that enable execution of user-defined applications is broadly termed - architecture. The architecture needs to meet user-defined functionality, satisfy resources constraints, and ensure application-specific performance considerations. Hence, it is composed of a distinct set resources suited for an application (or an application domain). The range of variations in the application domain accommodated in the target architecture determines the degree of flexibility and adaptability. Thus, the architecture itself encompasses a spectrum of resources specific to the application executed. In addition to the resources, the architectural framework also includes mechanisms to enable efficient interaction between resources. The architectural composition has constraints in terms of the type, number, functional, and physical characteristics. These resources and their interactions together compose the underlying architecture. These characteristics determine architectural attributes such as frequency of operation, power dissipation, area and memory requirements etc.

Processor architectures are further categorised into hardware and software specifications. Typically, the processor hardware relates to the functionality and resource constraints, whereas the software architecture administers application domain adaptability. The definite set of resources to allow capturing the required application-specific functionality defines the hardware architecture of the processing engine. The range of functional flexibility accommodated contributes to defining the programmable features. General-purpose processors appear at the far end of the design space with a large amount of architectural flexibility and application-specific hardware is the near end of the spectrum, limited to the application-specific functionality. The complexity of processor hardware necessitates the use of design automation to ensure faster design times, enable design verification, and provide a method of design space exploration. Multi-core architectures comprise several processors together on a single chip. Design and validation of multi-core processors not only involves verifying the functionality of

individual processors, but also requires validation of the interaction between multiple such processors, operating in synergy. Co-operative multiprocessing includes the infrastructure for inter-processor communication, synchronisation, and methods for resource and data sharing. To ensure fault-free operation in this scenario, defining a system-level model comprising individual processors, memories, hardware accelerators and evaluating its behaviour along with the inter-play of hardware and software becomes necessary.

This chapter starts with a discussion on architectural models in Section 2.1. It describes the advantages and disadvantages of the different performance estimation models available for designing processors. A classification of the methods of architectural customisations and the associated costs are detailed in Section 2.2. Section 2.3 discusses the costs and benefits of the different methods of architectural explorations used in literature. To understand the state of the art in multiprocessors, Section 2.4 is a survey of existing commercial multi-core processors. Section 2.5 looks at existing methods of application-specific architectural customisations. Finally, Section 2.6 provides a brief introduction to our reconfigurable multiprocessor — ***QuadroCore***.

2.1 Architectural Models

There is a wide range of models that assist system emulation. Such models include tools to capture the processor's functionality with instruction set simulators, bus functional models etc. In addition to ensuring functional correctness, performance models provide system-level estimates for area, power, and timing characteristics. Early functional and performance estimators enable faster fine-tuning of the processor architecture in order to meet application-specific requirements.

For multi-core architectures, the primary purpose of defining a model is to confirm the functionality and predict application-specific system performance prior to actual implementation. Functional validations are essential to address application partitioning, load-balancing etc. under varying input conditions prior to the actual architectural implementation. Performance estimations are necessary to ascertain the advantage of multi-core architectures with respect to time, power, and energy efficiency and suggest timely modifications to the architecture.

For processors, the range of the abstraction-levels for the models varies between high-level transaction-level definitions to low-level transistor characteristics. Two conflicting parameters govern the performance characteristics of architectural models, viz., functional accuracy, and simulation speed. Higher the accuracy of the system modelled; greater is its resemblance to the actual implementation. Nevertheless, higher the level of accuracy greater is the time and complexity associated to validate the design un-

der test. For example, an instruction accurate processor model is fast and accurate at instruction-level granularity, but neglects timing or clock-cycle considerations. In addition, it does not estimate system-level performance characteristics such as operating frequency, area, power etc. However, functional modifications such as additional instruction, altering the instruction functionality are easily incorporated. Further, instruction-level models are essential for early design evaluation, ahead of the design implementation. As a contrast, a gate-level processor model is much more accurate in estimating area, frequency of operation and power dissipation, but arriving at a gate-level model involves significant design efforts. In addition, introducing modifications to the design at a gate-level model is time consuming and error prone. Thus, the loss in accuracy in an instruction-level model is compensated by the speed of emulation. One of the main goals of defining a model is to enable early behavioural and performance prediction, for which a faster simulation speed provides a greater flexibility to introduce and verify design alterations. Using such models it is then possible to simulate entire systems composed of multiple such processors. ProtoFlex [1] and PTLSim [2] are examples of full system simulators.

Figure 2.1 shows a comparison of the types of models as a plot of functional accuracy versus simulation time. As also seen, the introduction of detailed physical characteristics require larger simulation time and elaborate models for simulation.

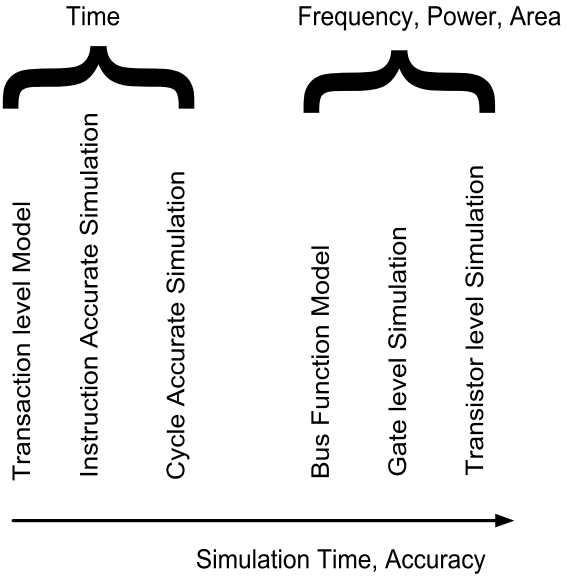


Figure 2.1: Comparing Simulation Time vs. Accuracy

With increase in precision, transistor-level specifications include detailed physical features such as gate dimensions, gate capacitances, performance characteristics such as timing, voltage, power etc. Gate-level netlists are constructed based on transistor-level characteristics and are composed of technology-specific gates, registers, flip-flops,

memories, which together compose processors. Further, collection of processors represents system on chip and multiprocessor system on chips. As can be see, design space explorations using these detailed models require higher design time in comparison to simpler models. Multiprocessor systems and system on chips comprise multiple such processors, requiring significantly larger time for emulation. In such cases, simpler models such as transaction level models are used to speedup emulation by approximating processor characteristics as transactions. A detailed comparison of simulation time for transaction-level models and RTL models is presented in [3].

Models for Multiprocessor Systems

Architectural description languages (ADL), such as UPSLA [4] (Universal Processor Specification Language), LISA [5], nML [6], and others [7] provide methods to specify the instruction set architecture of the processor. This functional specification is linked with parameters such as resource association, clock-cycles of execution, pipeline stages, etc. Using this description, an automated tool flow generates the instruction set simulator, compiler, linker, and assembler for the processor under design. A profile-driven automated tool flow enables easy modifications to the processor's instruction set architecture, as per application requirements. Although these models appropriately describe the function of the instruction set architecture, the absence of any system-level details such as area, power etc. limit usability of this model only for functional validation of the processor architecture. Including structural description such as registers, ALUs, and other resource constraints to realise the functionality is essential for system-level performance prediction.

In addition to the processor's instruction set architecture, the communication mechanisms between the processing elements have evolved significantly. In the present day processor architectures, the diversity in the communication infrastructure ranges between a shared bus and shared memory system, to multi-layered buses and crossbar switches, and further to packet-based communications using network-on-chips. The choice of the communication model makes a significant contribution to the overall system performance. The choice of the communication model is dependent on the type of application executed. For example, data-intensive processing in stream-processors (such as [8]) share a register file and each of the ALUs access the register file via a dedicated cross-connect. Network processors (such as BCM 1480 [9]) share a common data-cache, where cache coherency is enabled via a shared data bus. It is therefore necessary to include both the processor and the communication mechanism to ensure functional and performance validation of the entire system.

A cycle-accurate model using gate-level description provides a mechanism to emulate the entire system using vendor-specific libraries. However, the complexity and time

involved at these levels of accuracy necessitates methods of accelerated prototyping using programmable devices such as FPGAs. However, methods of prototyping using FPGAs are limited to functional validation at lower frequencies. System-level model provides a mechanism to evaluate design details and introduce modifications early in the design cycle. Finer, accurate models introduced later in the design cycle confirm the design's functionality and provide accurate performance estimates. Thus, a holistic system-level simulation is essential to ensure collective validation of both the hardware and software specifications.

2.2 Architectural Flexibility

Architectural flexibility has a benefit of circumventing frequent hardware redesigns by providing customisation capabilities in the post-fabrication phase. Adaptability is incorporated to expand the range of architectural reusability. As a counter effect, there is a cost involved in adding flexibility to the architecture. In order to accommodate a range of application-level diversities, this adaptability introduces a definite degradation in the performance parameters such as maximum operating frequency, area, power consumption etc.

2.2.1 Classifying Customisations

Design time or pre-field customisation refers to the architectural modifications that can be introduced early in the design cycle, to match the application characteristics. Further, the primary requirement for this type of customisation is the complete knowledge of the application characteristics during the time of design. Consequently, any additional modification involves a rerun of the entire design cycle, viz., design, verification, and fabrication time. This overhead directly relates to the additional costs and time-to-market. Design time customisations introduced are also referred to as configurability or extensibility [10].

Typically, the standard configurable parameters in a processor design cycle are modifications to existing building blocks such as introducing an additional ALU, varying the number of registers, or register ports, word-length of the ALU, additional memory etc. Similarly, extensibility refers to inclusion of architectural enhancements such as additional instructions, based on profile information obtained for a specific application. It is also restricted to application requirements known well ahead of execution time. Configurability and extensibility are both associated with incremental updates to an existing processor architecture introduced during design time. Further, they result in incremental design updates and modest verification times in comparison to a complete

design re-spin, since modifications are introduced to existing, pre-verified design blocks or IP cores. Processors from Tensilica [11] and ARC [12] are examples design time configurable and extensible processors. Configurable IPCores from vendors such as Xilinx are examples for application-specific configurable cores.

Run-time or in-field customisation refers to alterations that can be introduced in a device long after the chip has been shipped or even during in-field operations. Programmability and reconfigurability are two distinct options. Programmability refers to modifications introduced via application software to a fixed hardware. In this case, the underlying hardware has features that are programmed in the application-mapping phase. Most processors include programmable features that allow a range of applications to be mapped. This is in contrast to Application Specific Integrated Circuits (or ASIC), which are dedicated for a single application with limited programmable features.

As a contrast, reconfigurability refers to modifications to the hardware architecture introduced long after the chip is fabricated. Reconfiguration introduces modifications to both the control path and datapath of the underlying hardware architecture. FPGAs are examples for reconfigurable architectures, where the individual building blocks are configured to meet application-specific functional and performance requirements. Depending on the amount of modifications that are introduced and the complexity of the configurable blocks, they are further classified as coarse-grained or fine-grained reconfigurable architectures. In addition, dynamic run-time reconfiguration refers to modifications introduced during the operation of the device, without affecting its normal operation. Although the amount of flexibility that can be introduced during run-time makes an impact on the performance characteristics, it has the advantage of introducing design updates in the post-fabrication phase.

2.2.2 Cost of Flexibility

Introducing architectural flexibility often results in a performance compromise. Flexibility in the architecture is a trade-off between performance characteristics such as operating frequency, area, power, and energy efficiency in comparison to fixed architectures with features limited to a single application or an application domain. An example of processor flexibility can be seen in the programmable options in general processors that support a range of applications to be mapped. This is in comparison to dedicated digital signal processors that exhibit significantly higher performance, although limited to the signal processing application domain. In this scenario, a perfect solution is a single architecture entirely customised for one single application, thus exhibiting optimal performance characteristics for that particular application. Consequently, a *performance deviation* is noticed in comparison to the ASIC or a fully

custom implementation. A fully custom ASIC implementation is this perfect solution. Thus, adding flexibility results in a performance change in comparison to the perfect ASIC solution. This change in performance is termed as **performance deviation**. It refers to the loss in performance because of added flexibility.

Metrics : In order to illustrate architectural features as an index for comparison, the following metrics were chosen:

- *Clock Cycles* : The number of clock cycles required to execute a given application (or program), when mapped onto this architecture.
- *Frequency* : The operating frequency of the target hardware implementation. Execution speed is a product of two parameters, viz., number of clock cycles required for execution and the operating frequency.
- *Area* : The total gate count or the area occupied by the architectural implementation.
- *Power* : The total power consumption when a given application is mapped on this target architecture.
- *Customisations*: The range of application-specific customisations that can be introduced to enhance performance on the given architecture during design time or run time.

Architectures : In order to analyse diverse performance features between architectures, the following four representative architectures we chosen:

- *ASIC* : Fixed, application-specific dedicated device.
- *RISC processor* : Typical programmable processors used in embedded processing domain.
- *Configurable processor* : A RISC processor that can be customised as per application requirements, during design time.
- *FPGA* : Highly customisable architectures, fine-grained configurability during deployment.

Comparisons

The degree of performance deviation is indicated by **high**, **medium**, **low**, and **none**. They refer to the difference between the perfect solution (or an ASIC implementation) and the architecture under consideration. A high degree of difference indicates a large performance deviation compared to an ASIC implementation, which is a negative

trend towards performance merits. Similarly, a low performance deviation corresponds to a relatively low loss in performance. Consequently, a match in performance with the ASIC implementation is indicated by a degree of zero or none, which is a desirable feature for performance. Only in the case of customisations, a high degree of performance deviation is considered as a positive impact, since it results in ease of usability for a wide range of applications. Nevertheless, it results in other side effects as negative impact in other performance indices.

Table 2.1: Comparing Performance Deviation among Architectures

Architecture	Cycles	Frequency	Area and Power	Customisations
ASIC	None	None	None	None
RISC Procs.	High	Low	Medium	High
Config. Procs.	Medium	Low	Low	Low
FPGA	None	Medium	High	High

Table 2.1 summarises the *performance deviation* that can be expected in RISC processors, Configurable processors, and FPGAs in comparison to an ASIC.

An ASIC implementation represents a perfect solution. It exhibits no performance deviation and also has no configurable options. Thus, it is primarily applicable for a single application - single architecture scenario.

For RISC processors, the sequential mode of programming translates into a large number of clock cycles for implementing a given algorithm. Hence, a high performance deviation for the clock cycle in this case. The need for a large number of clock cycles for implementation is typically compensated by a high frequency of operation, which makes RISC processors slightly inefficient in comparison to ASICs with respect to the operating frequency. Further, as these processors are not customised for an application they do incur a medium performance deviation for the area and power reports, for the additional resources that exist. However, the high customisation capabilities extend the usability of RISC processors over a range of applications.

In the case of configurable processors, features such as instructions to access new functional blocks and resources can be introduced after knowing the application characteristics. Thus, the performance deviation for time, area, and power are reasonably lower than for RISC processors. However, these performance characteristics of a configurable processor are largely dependent on the additional features and suitability to the application domain. Nevertheless, customisation is higher than an ASIC implementation, but lower than that of RISC processors and FPGAs. Here, configurable processors are indicative of application-specific processors, fine-tuned for a single application.

FPGAs are representative of fine-grained programmable devices with extensive customisation capabilities. On account of the customisation capabilities, the realisation of an application is identical in terms of clock cycles required for execution. Thus, performance deviation for clock cycles required for implementation is absent. However, there exists a fair amount performance deviation in terms of frequency of operation and a significantly high area and power deviation. The identical performance in terms of clock cycles required for execution is a significant advantage for FPGAs.

Table 2.2 lists comparative figures for a RISC processor (MIPS 34K), a configurable processor (XtensaLX2), and an FPGA (Xilinx Virtex IV), to understand the performance deviation listed in Table 2.1.

Table 2.2: Examples for Performance Deviation among Architectures

Architecture	Processor	Frequency	Area	mW/MHz
RISC Procs.	MIPS 34K	500 MHz	0.93 mm^2	0.56
Config. Procs.	Tensilica LX2 [13]	400-475 MHz	0.14 mm^2	0.032 - 0.046
FPGA	Xilinx XC4VPFX100	$\approx$ 400 MHz	$\approx$ 300 mm^2	$\approx$ 16

2.3 Architectural Design Space Exploration

Within the processor design cycle, architectural explorations are performed at different stages with varying degrees of freedom. At design time, the entire processor can be redesigned or customisations to an existing processor may be introduced as required by the application. As a next stage, compilation and mapping is an important phase of design space exploration. Finally, during run-time, modifications are introduced to the application via software modifications or to the hardware via reconfiguration. It has to be noted that the degree of freedom or the amount of modifications that can be introduced via design space exploration reduces farther down the design flow, i.e., from design-time to run-time.

2.3.1 Classifying Architectural Explorations

Design time customisations introduce functional definition of resources, determining the number of resources, choosing the right communication mechanism for interconnectivity between resources etc. Internally, a processor comprises generic building blocks such as ALUs, register files, memory, and a mechanism for interconnecting these resources. With these components comprising a resource library, architectural customisation during design time configures a processor system. This library can be

used to build processors with varying number and types of resources, as depicted in Figure 2.2. The bottom-left of the figure shows conventional processors with predefined set of resources and a fixed mode of operation. The bottom-right shows the scalable template, where a hierarchy of processors can be built using a unified processor template. The communication between these processors is also configurable. The lower levels in the hierarchy show a point-to-point network and the higher levels extend to packet-based communication. As can be seen, a range of processors can be designed using this methodology. However, the performance of such an architecture is known completely only after an application has been mapped onto the architecture, i.e. during compile-time. Compile-time explorations or profile-driven fine-tuning is discussed in the following paragraph. In contrast to configurable processors, the instruction set architecture is also included within the exploration space. The presence of a resource library speeds up the design and verification time.

Since the library is predefined, the characteristics of each of the building blocks, such as functionality, area, time, power are known during design time. Design space exploration grades architectures based on these performance parameters to enable continuous modification of the architecture to suit application-specific requirements. In order to analyse the application-specific requirements, ***compile time*** explorations provide suggestions to tune the architecture. Figure 2.3 shows the profile information classified as register accesses, memory accesses and ALU operations for an application mapped to an architecture. Depending on the requirements, additions to the architecture are made to match the architectural features to conform to the application demands. Compile time analysis provides a feedback with respect to application-level statistic determined during compilation, which consider diverse performance parameters such as time, power, and area.

Meeting diverse application-specific demands requires a system-level performance feedback. During compilation, architectural design space exploration is assisted via a ***feedback-driven*** mechanism. Thus, early performance estimates are accurate in terms of clocks cycles, power, area, and operating frequency. This method of application-specific architectural modification can be extended as a method of prototyping to validate the new processors micro-architecture. Since each of the building blocks that compose the architecture use components from an existing library, they are individually verified and validated.

Although compile time analysis provides greater degree of freedom, each design enhancement corresponds to design changes, resulting in extended design and validation times, and inevitable fabrication costs. Post-fabrication and in-field design modifications can be introduced via ***run-time*** reconfiguration. Run-time reconfiguration is applicable in cases where the architecture has to be adapted to variations in the environment, to performance changes identified in the post-fabrication phase, or to

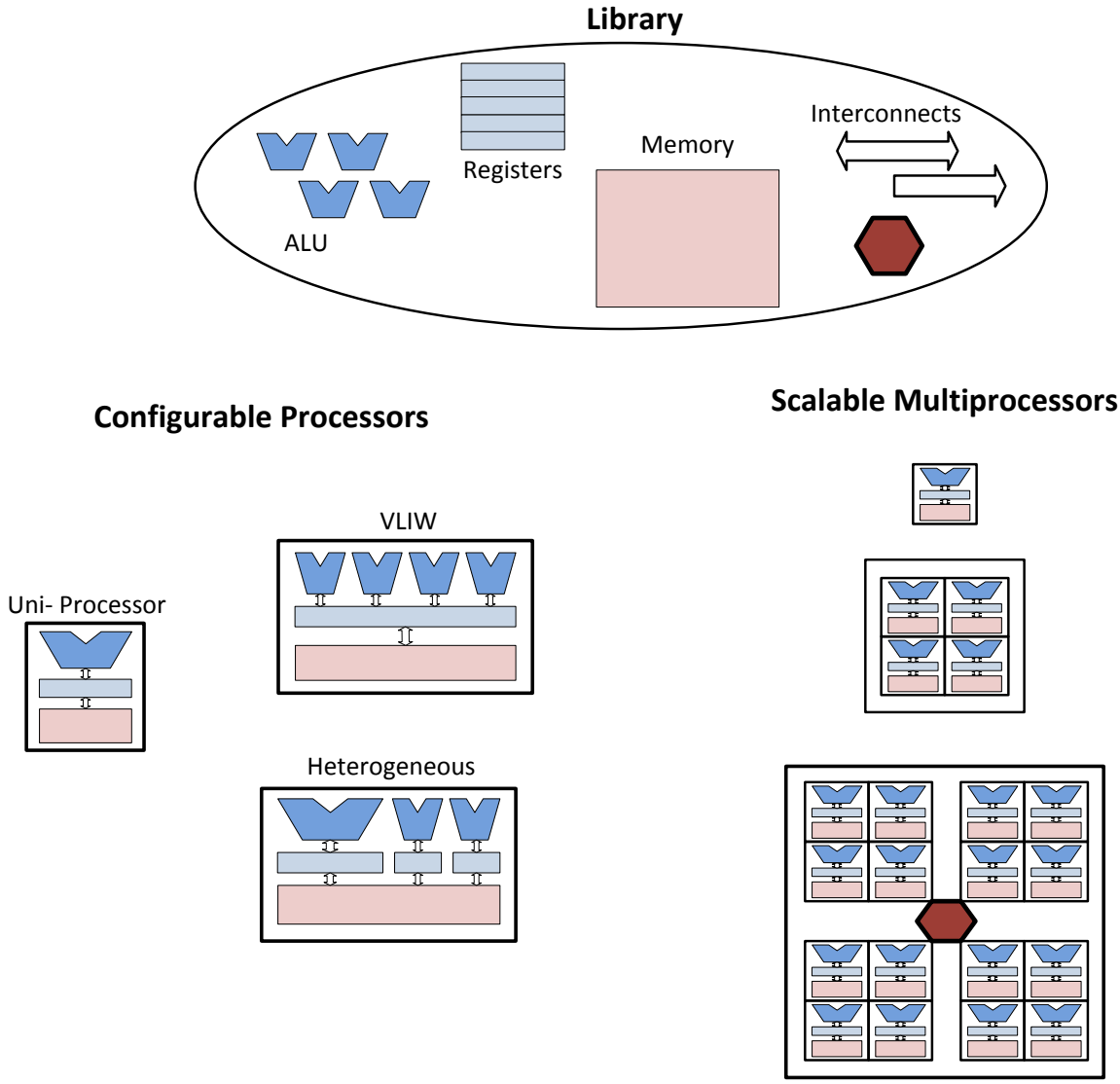


Figure 2.2: Design Philosophy for Processor Architectures

alter performance during in-field operation. Reconfiguration can be used to alter the time and power characteristics. Further it is most suited to address the domain of ***reliability***, where faulty parts are replaced by functional parts during run-time.

2.3.2 Ranking Architectural Merits

The performance of an architecture is specifically dependent on the performance of an application (or the range of applications) mapped onto this architecture. The application's performance merits such as instructions per cycle, total execution time, total gate count, power dissipation, and total energy consumption is used for comparison between different architectural platforms. However, independent of the application are ratings

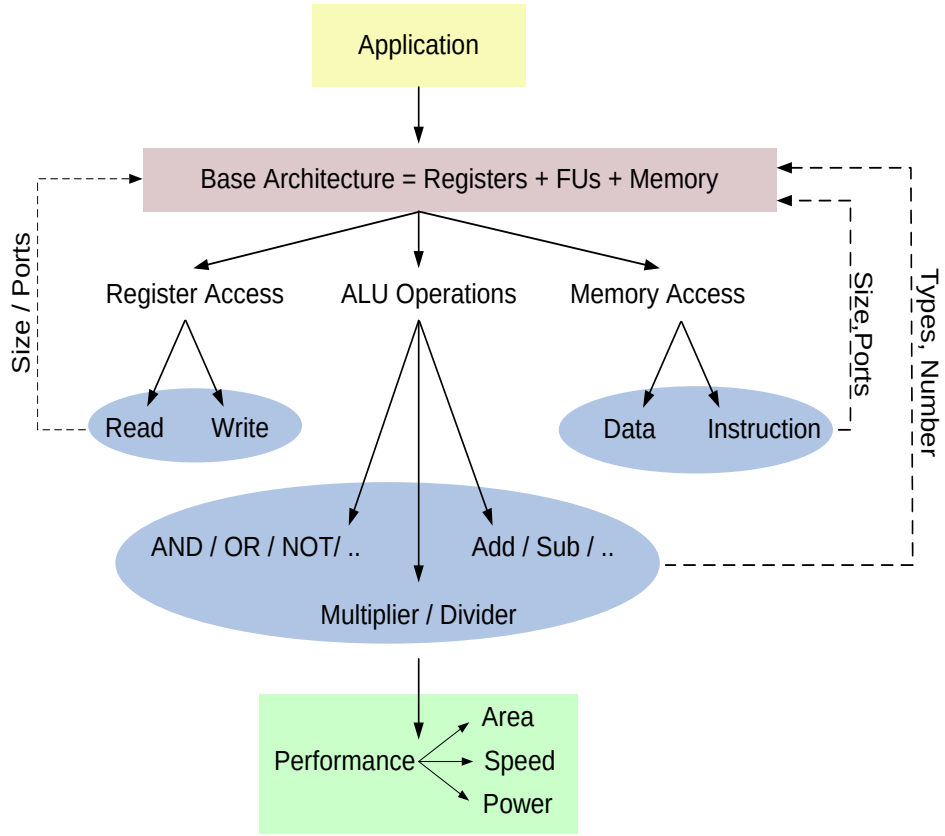


Figure 2.3: Profile-driven Architectural Tuning during Compile time

such as ease of programmability, range of adaptability, backward compatibility, customisation capability, scalability, and reusability, which influence the time-to-market and also the success or popularity of a particular architecture. Further, in order to enable design space explorations, the ease at which architectural enhancements can be introduced is also an index indicating architectural flexibility. The reasoning behind designing our reconfigurable multiprocessors QuadroCore is to merge the advantages of reconfigurable FPGAs and parallel multiprocessors. The focus is to minimise design time, enhance code-reusability, and ease application-specific customisation with a minimal impact on performance characteristics. In other words, the focus of designing QuadroCore is to ensure a lower *performance deviation* as compared to multi-core processors and reconfigurable FPGAs.

2.4 Multi-core Architectures

Multi-core processors are increasingly being used in almost all application domains. For network processing applications, multi-core network processors are advantageous

as individual packets are processed in parallel on multiple cores. In image and video processing, graphic processors provide data-level parallelism with large number of specialised processing elements. Similarly, even in general-purpose computing, multi-core processors are being used to enable simultaneous execution of tasks to accelerate processing applications with diverse characteristics. This section is a survey of existing commercial multi-core processors, and a summary of their individual advantages and disadvantages.

2.4.1 Commercial Multi-core Processors

Figure 2.4 is plot of existing multi-core processors that have been designed primarily with multiple instance of existing processors. In the figure a range of application domains have been considered to provide a consolidated overview of the number and speed (expressed as frequency of operation) for existing multi-core processors, viz., network processors, graphic processors, and general-purpose multi-core processors. The X-axis corresponds to the operating frequency of the processor, the Y-axis is a logarithmic scale of the number of processing elements in the processor, and finally the size of the bubble represents the power consumption of each of the processors.

Network processors with multiple cores have been used for a long time, since parallelism in packet processing is easy for distribution of tasks among multiple processing elements. Using the MIPS core as a building block has been commercial success for a number of network processor vendors such as RMI [14], Cavium [15], Broadcom [9], PMC Sierra [16], etc. Network processing involves managing simultaneous processing of multiple packets. Hence, a *homogeneous architecture* with multiple identical processing elements are characteristics of these network processors.

Graphic processors (GPU) can be categorised as domain-specific architectures, since primarily targeted for video and graphic processing. These applications are characterised by a sequence of identical graphic operations to be performed on parallel data streams. Hence, these processors comprise multitudes of graphic-specific hardware accelerators, such as programmable pixel and vertex shaders etc., for fast and simultaneous processing of parallel data streams. A state of the art graphic processor from nVidia called TESLA [17], targeted for high-performance applications comprises 960 scalar processors, where each processor supports execution of 240 concurrent threads. ATI's Radeon 4800 series [18] of graphic processors targeted for gaming applications comprises up to 800 streaming processors, with the core clock frequency of 600 MHz and a memory clock frequency of 800 MHz. However, a recent initiative extends the use of GPUs for general-purpose computing (GPGPU).

Sun's T1 Niagara [19] is an early example of multi-core general-purpose processor with eight identical processors, each capable of executing four simultaneous threads. Thus,

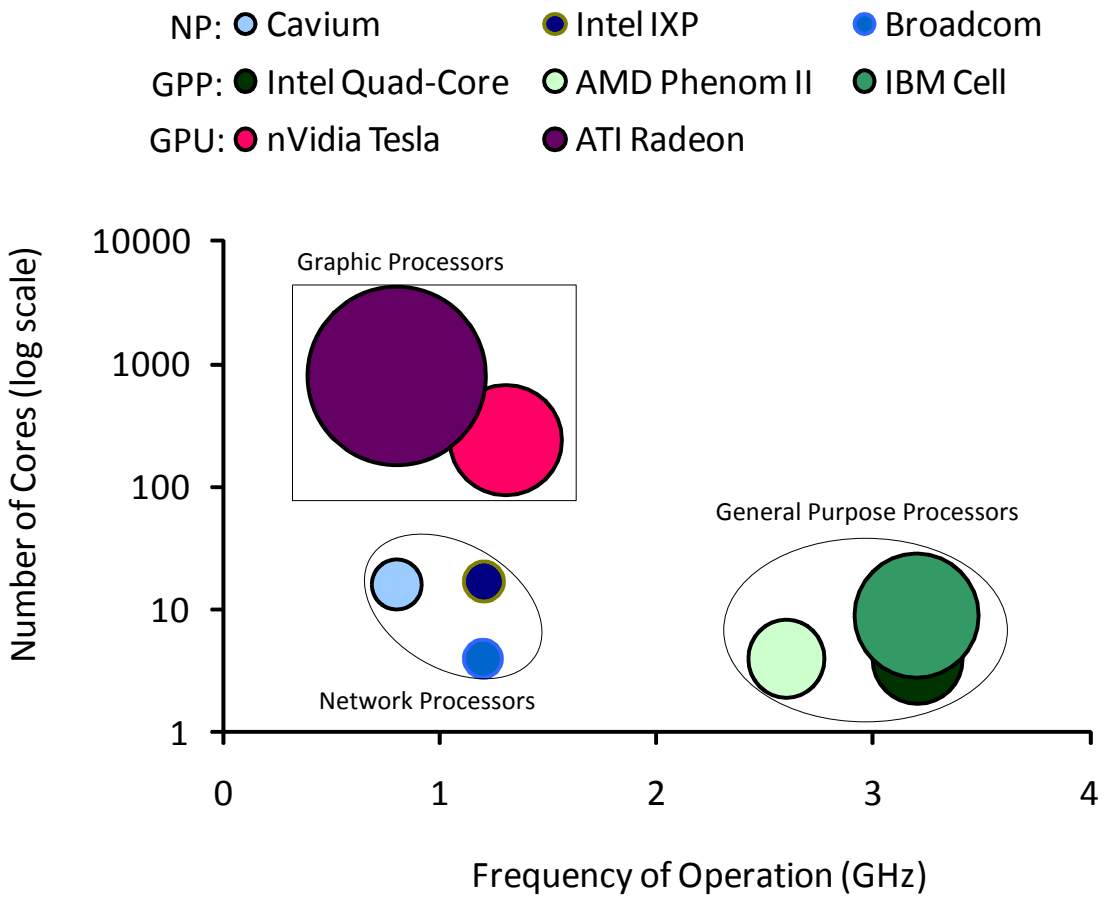


Figure 2.4: Design Space of Commercial Multi-core Processors

32 threads can be in execution simultaneously. Other examples of multi-core general-purpose processors are quad-core processors such as Intel's core i7 [20] and AMD's Phenom [21], with four-identical processors, which share a common L3-cache and support simultaneous multi-threaded applications. These are all cases of *Homogeneous Architectures*, where copies of the base processor with local cache are provided with an additional stage of shared cache, for data and program sharing.

In processors such as AMD's Fusion [22] the architecture comprises a primary general-purpose processor together with a graphic processor as a complete system-on-chip solution. These are examples of *Heterogeneous Architectures*, where processors with different instruction set architectures are combined together on the same chip. IBM's Cell architecture [23] is also a heterogeneous multi-core with a PowerPC as the primary processor together with eight additional RISC processing engines. The primary processor mainly ensures backward compatibility to existing software. Performance improvements are achieved by mapping compute-intensive applications among the parallel cores. Similarly, heterogeneity has been introduced in existing homogeneous

FPGAs with the inclusion of hard and soft processors, leading to the classical concept of hardware software co-design for design partitioning and design acceleration. Heterogeneity exists also when designing system-on-chips for specific applications also called *domain-specific architectures*. These architectures typically comprise specific hardware IPs, well suited for specialised application domains.

In addition to the processor's microarchitecture, the performance figures are strongly influenced by the communication mechanism among the processing elements and memory. *Inter-processor communication* enables transfer of instruction and data between resources within the multi-core organisation. The coupling between resources is an architectural decision based on the amount and frequency of communication between the resources. As an example, VLIW processors (Very Long Instruction Word) are predominantly used for applications with instruction-level parallelism, which requires frequent exchange of data between processing elements. In such a scenario, a multi-ported shared register file is typically incorporated. In streaming processors, where a large amount of data is continuously processed, deep pipelines and structuring the hardware such that intermediate processing results are forwarded or stored on-chip memory avoids frequent cache or off-chip memory access. This is in contrast to general-purpose CPU architectures. In general-purpose multi-core processors, to support task and thread-level parallelism, a shared cache is typically employed, where coherency is maintained via a shared bus. Often, the number of resources simultaneously active on the shared bus limits the bandwidth. The limitation of sharing the bandwidth in a bus-based architecture is overcome by using packet-based communication in a network-on-chip or a switched cross-connect. *On-chip interconnects* such as the Element Interconnect Bus (EIB) in IBM's Cell architecture facilitates communication between processing elements, memories, and other on-chip peripherals. As described in [24], the implementation resembles a ring with four unidirectional channels, where three concurrent operations can co-exist.

The data movement between the processing engine and location of the bulk memory decides the *memory hierarchy* within the processor organisation. Typically, a fast access to intermediate values computed is provided via a register-file, which is limited in size. Cache mechanisms provide a temporary stage for fast data or program memory access. Caches are also used for sharing of memory between independently operating processing element with their local memory. An intermediate stage of scratchpad memory provides a low-cost mechanism to store frequently accessed data and/or instruction in a locally available shared scratchpad, which is identified during compile time. Scratchpads help to reduce costs, both in terms of the access time and the energy consumption [25].

Reusability is gaining importance with increasing design complexity influenced by increasing transistor densities, and ever increasing demand for a fast time-to-market.

Hardware reusability promotes using the same processors architecture over a range of applications and processor families, in order to allow backward compatibility. Software reusability encourages code portability between processor generations and processor families.

2.4.2 Limitations of Existing Multi-core architectures

To summarise, existing commercial multi-core architectures can be predominantly characterised as multiple instances of existing processors, with a communication infrastructure, and a pre-defined shared memory hierarchy. The microarchitecture of the processing elements depends on the application domain and the application domain inter-processor information exchanges define the communication infrastructure. As can be seen from Table 2.3 each application domain has a diverse architectural model. Furthermore, application-specific customisations are restricted to the processor architecture. The processor architecture is customised with application domain hardware accelerators to enhance performance. The single most significant disadvantage of such architectures is the applicability only to data or task-parallel applications, and each of the multi-core architectures is restricted to its own application domain. In addition, the decision of when task or data can be partitioned onto the processing elements is limited to threads defined during application description. Further, for multi-threaded application, it becomes necessary to design a cache-coherent system, where the bandwidth of the shared cache is an important bottleneck for including additional processors. Thus, a single template that can be reused over a range of applications, ensure scalability, and easy adaptability is typically yet to be established.

Table 2.3: Commercial Multi-core Processors

	Processor	Cores	Clock Freq.	Power	Memory Access
GPP	Intel Quad-Core [20]	4	2.6 - 3 GHz	130 W	Shared L3cache, QPI
	AMD Phenom [21]	4	2.5 GHz	95 W	DCA and HyperTransport
	IBM Cell [23]	CPU & 8 SPE	3.2 GHz	250 W	EIB with Shared L2
	UltraSPARCT1 [26]	8	1.2 GHz	79 W	Crossbar, Shared L2
NP	Intel IXP 2850	16PEs + Xscale	1.4 GHz	27 W	Shared bus,Nearest neighbour
	Broadcom BCM 1480 [9]	4	0.8-1.2 GHz	23 W	Dedicated, HyperTransport
	Cavium Octeon 58XX [15]	4-16	500-800 MHz	15-40 W	Shared cache,Coherent Interconn.
DSP	Freescale MSC8144 [27]	4 + QUICC	1 GHz	8 W	Shared L2 cache
	TI TMS320C6474 [28]	3	1 GHz	8 W	Shared L2 Cache
	Analog Devices ADSP-BF561 [29]	2	600 MHz	2 W	Shared L2 Cache, scratch pard
GPU	nVidia Tesla C1060 [17]	240	1.3 GHz	160-200 W	
	AMD Radeon HD4800 [18]	800	800 MHz	500 W	

GPP: General-Purpose Processors
NP : Network Processors
DSP: Digital Signal Processors
GPU: Graphics Processors

2.5 Initiatives for Customisable Multi-core Processors

Customisations have been introduced to multi-core processors in order to adapt to application-specific requirements. This classification exemplifies alterations introduced to existing architecture or a generic template, to customise it to an application. In this section, we look at some of the multi-core processors used in the embedded processing domain and analyse their customisation capabilities.

In ***configurable processors*** such as Tensilica's Xtensa [11] and ARC's ARC700 [12], configurable options permits user-driven processor configuration, with parameters such as instruction length, cache sizes, number of functional units etc. With this base configuration, it is then possible to add instruction set extensions to meet application-specific performance requirements during design time. Although configurability allows a range of customisation capabilities, it does not assist in altering the architecture for in-field upgrades. Further, configurability necessitates new design re-spins for every application-specific customisations introduced. Overall, it provides a good mechanism for design space exploration early in the system design cycle. Another example of configurability is seen in SiliconHive's ULIW (ultra long instruction word) processors [30]. These processors are built using a single VLIW processor template and additions to the number of core cells or IO cells can be included as a part of the ultra long instruction word. SiliconHive's processors support instruction-level parallelism by introducing additional functional units, addressable as individual instruction slots.

Tile-based architectures, comprise copies of identical tiles, where each tile consists of a processing element with its own local memory. Processors such as like Tilera or RAW [31], Ambric [32], Picochip [33] etc. are examples of this classification. In these architectures, each tile or entire processors can be replicated to enable scalability, fault tolerance on account of homogeneity, and customisations on account of functional unit configurability. The Tilera (or RAW) architecture is composed of identical programmable tiles, where each tile consists of a MIPS-like processor, data cache, instruction cache, and routing logic. The communication mechanism is designed to allow only a single clock delay for travel across one tile. RAW supports variations in the interconnections topology between processing elements in an array like structure. The compiler manages the RAW hardware resources and implements run-time analysis for application-specific customisation. Using the same analogy, PicoArray [33] provides a flexible two-dimensional grid of processors using a 3-way VLIW processor, with four such processors in a cluster. A level lower in the design hierarchy is the pipeline processor from Rapport called Kilocore [34], which replicates the individual pipelines for enhanced parallel processing. A polymorphous architecture called TRIPS is presented in [35], which has an additional feature of configuring to application-specific granular-

Table 2.4: Customisations in Existing Architectures

Core	PE	Inter PE Communication	Customisation
ARC/Xtensa	RISC	Queue/FIFO based	Registers, Accelerators
SiliconHive	VLIW	Prog. Interconnect	Functional units, Registers, Instruction slots
Rapport Kilocore	8-bit CPU	Queue, Register file	Pipelines
PicoArray	3-way, VLIW	Bus and Switch Box	Inter-PE communication
Ambric	32-bit RISC	Queue based	Interconnects, Scalability
Tilera (or RAW)	RISC	Nearest neighbour	Inter-PE communication
FPGA	LUTs	Switchboxes, Routes	LUTs, Routing

ities and parallelism. The architecture is composed of large coarse-grained partitioned cores and avoids a centralised routing. The absence of a single routing unit avoids long wires and eases scalability. Point-to-point communication channels are incorporated to enable exposure to software for optimisation during application mapping. These regular processing fabrics provide coarse-grained architectural customisations in comparison to fine-grained FPGA fabrics.

Among ***reconfigurable architectures***, the FPGA is the most popular example. FPGAs are composed of regular homogeneous fabrics are representative of scalable architectures, where a single architectural fabric is used over an entire family of FPGAs. They are fine-grained regular fabrics with configurable logic blocks (CLB) as individual computation building blocks that can be scaled to increase the computational capability of the fabric. Each individual building block is configurable to application-defined functionality. Multiple such building blocks form intermediate hierarchies (slices) to enable building uniform clusters with multi-hop communication networks. All the components in an FPGA - the configurable blocks, the routing logic, and the memories are programmable to user-defined architectural requirements. Further, programmable features enable modifications to both the data and the control path, thus making this architecture completely reconfigurable. The introduction of hard-cores, DSP blocks, and multipliers within the regular configurable fabrics introduces heterogeneity.

Table 2.4 summarises the above mentioned architectures considering the range of configurability using existing processors, as a method of application-specific adaptability. Additional details such as application-to-architecture suitability is listed in [PPR09].

In contrast to methods of general-purpose reconfigurability in the above-mentioned architectures, are a few application-specific reconfigurable processors discussed in this section. Each of the method discussed in this section is distinct and application-specific. In [36] application-driven reconfigurable features have been introduced in a

single-chip ***reconfigurable*** FFT/IFFT ***processor*** in order to achieve energy savings and algorithmic flexibility. The architecture comprises a collection of FFT processors organized in ring topology. Reconfiguring the inputs, outputs, word-length of the datapath, and partitioning the processing units for hardware sharing enables adapting to application-specific data-flow demands. Another dynamically reconfigurable processors called, FlexCore [37], has an interconnect that reconfigures the datapath. However, this framework requires redefining the instruction format and necessitates the use of long instructions to adapt to the modified datapath. In [38], reconfiguration is used as a method for transforming a floating point unit in a superscalar processor into several independent ALUs, with minimal additional latency. This results in a performance gain of up to 56%. In [39], a reconfigurable datapath processor implements a synchronous pipeline computation model, where configurable processing elements are connected by a configurable datapath. Here, execution agility is achieved by conditional switching of datapaths. A processing cell architecture [40] supports synchronous VLIW, asynchronous Multiple Instruction Multiple Data Streams (MIMD), and a mixture of both modes of operation via reconfiguration. Through dynamic reconfiguration, the cluster can be divided into a plurality of processing cells or be merged into a single large cluster. The interconnection between processing elements is controlled by RAM-based configuration switches. Depending on the application's requirement, the switches can be reconfigured to allow neighbouring cells to exchange programs etc. Along the same lines, in [41], a multiprocessor architecture is described where a crossbar switch acts as a reconfigurable element for altering the processor-to-memory and processor-to-processor connections. This configuration environment allows switching a MIMD operation to Single Instruction Multiple Data Streams (SIMD) operation.

Pros and Cons of Existing Reconfigurable Architectures

Reconfiguration is a mechanism that customises the hardware architecture during the deployment phase. The advantage of this scheme is to permit user-driven customisations to be introduced without having to redesign and re-fabricate the device. Ideally, the reconfiguration scheme should be capable of introducing application or algorithm-specific customisations to the architecture without incurring a significant reduction in performance and reconfiguration management. The base architecture needs to be application independent, wherein reconfigurability during run-time permits application-specific customisations with a low *performance deviation*(defined in Section 2.2.2). The reconfiguration mechanism in FPGA-like architectures has three main disadvantages. Firstly, the time required to reconfigure is large. For present day FPGAs, it is in the order of a few milliseconds. Secondly, a dedicated mechanism or a reconfiguration manager is explicitly required to control and manage the process of reconfiguration. Finally, an additional resource such as an external memory or dedicated storage is

required to store the configuration streams. These are some of the drawbacks of using FPGA architecture for run-time customisations. Moving beyond the fine-grained reconfigurability in FPGAs, the possibility of reducing the reconfiguration overhead is introduced by increasing the granularity of the logical blocks. Coarse-grained reconfiguration schemes avoid the overhead involved in architectural redesign and reconfiguration management, but are restricted in usage to a specific application domain, or a given processor architecture. Ideally, the reconfiguration scheme needs to be quick, easy to manage and make minimal impact on account of customisations integrated into the base architecture. In the following section, we present our concept of the reconfigurable multiprocessor template that introduces reconfiguration as a method of run-time application-specific adaptability. This scheme is generic and can be applied to most in-order processor architectures. The customisable features can be introduced via high-level programming. The focus is to manage reconfiguration without additional overheads as observed in existing reconfigurable architectures.

2.6 The Concept: Run-time Reconfigurable Multiprocessors

A fixed architecture often results in resources being unused or inefficiently used, depending on the application that is being executed. This optimality in resource utilisation influences resource efficiency, power consumption, and consequently the overall system performance. In order to achieve a balance between application domain customisation while efficiently using the resources in a multiprocessor, we have developed a generic reconfigurable multiprocessor template. The template in Figure 2.5 shows an example of multiple loosely coupled processors in a reconfigurable fabric. The basic processor building blocks act as fixed resources in the reconfigurable fabric. These resources together comprise the compute engine of the processor. Here, reconfigurable connectivity between the existing resources is added as an architectural feature. The connectivity between resources is reconfigurable, thereby permitting user-defined processors to be configured. The template itself can be scaled to suit application demands using multiple such copies of processors. This feature extends the usability of the multiprocessor architecture to applications with varying resource requirements and degrees of parallelism. Since, the basic building blocks of a processor remains untouched, the added flexibility does not interfere with the original functionality and the instruction set architecture. The original processor's functionality remains unaltered. The reconfigurable interconnects allow interchangeable resource connectivity. The building blocks within a processor are treated as distributed resources, accessible by all (or a subset of) processors. This arrangement allows cooperative resource sharing within

multiprocessor hierarchy. The modularity and generic structure of the architecture allows easy reusability and scalability to suit application requirements.

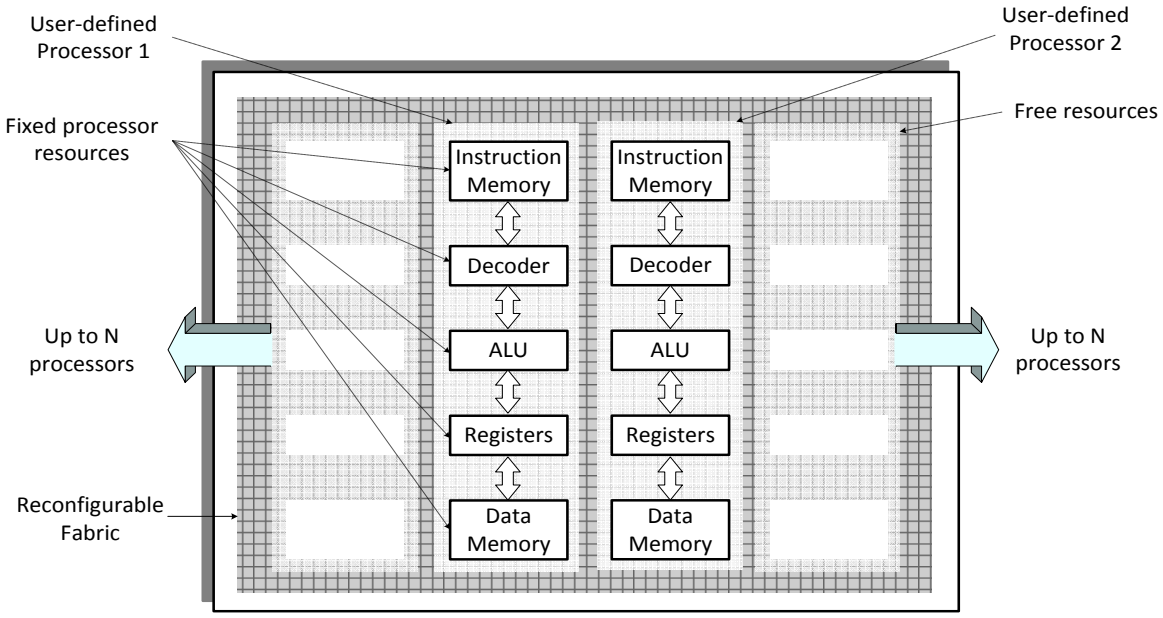


Figure 2.5: Reconfigurable Multiprocessor Template

The reconfigurable fabric allows dynamic alterations to the interconnections between the processor building blocks. The decision of altering the existing structure is driven by the instruction executed. Hence, the choice of resources and its variations are determined during compilation and executed during run-time. Scalability, easy adaptability, and extensibility are the main advantages of this multiprocessor template.

2.6.1 Reconfiguration Mechanism

A new method of reconfiguration has been developed to avoid the drawbacks in the present day reconfiguration schemes. Here, a quick, single cycle run-time reconfiguration is implemented to minimise the time required to reconfigure the resource connectivity. The datapath or the control path can be reconfigured at run-time, as directed by the instruction stream. Additionally, the reconfiguration stage can be introduced as an additional pipeline stage to enhance the architectural flexibility. The two mechanisms are discussed in the next section.

Instruction to Administer Run-time Reconfiguration

A special reconfiguration instruction executed during run-time connects resources, depending upon the resources demanded by the application. The hardware modification slightly alters the delay in the critical path, but flexibility is the achieved trade-off.

Reconfiguration as a Pipeline Stage

The stage of reconfiguration is introduced as an additional pipeline stage. The additional pipeline stage allows an entire clock cycle for configuring the resource connectivity. Figure 2.6 shows the additional stages, which results in an increase in the instruction length affecting the latency of execution. An explicit pipeline stage avoids any change in the operating frequency on account of the modified critical path.

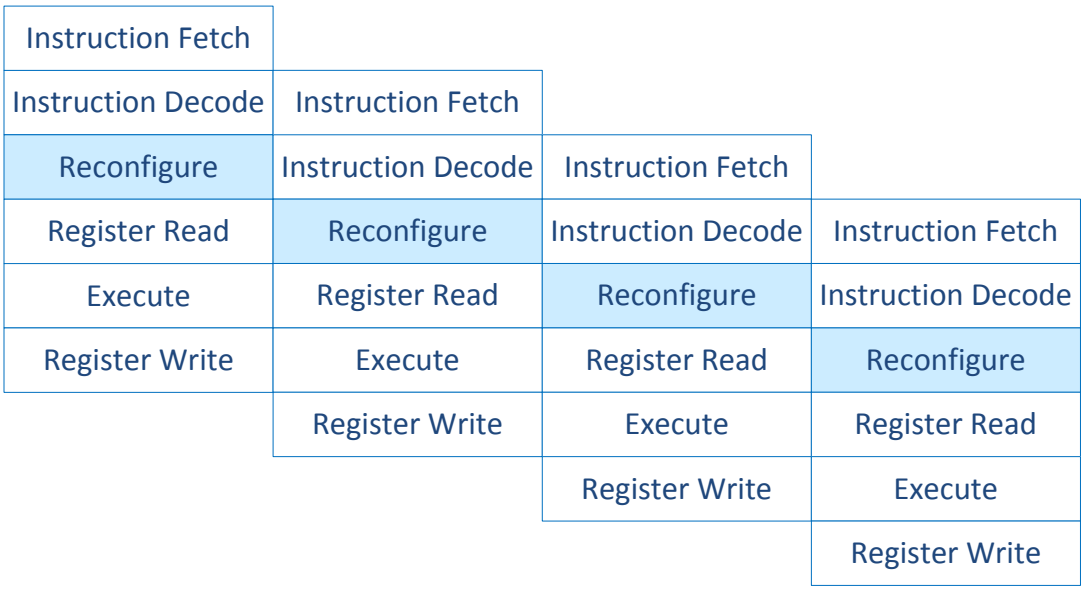


Figure 2.6: Reconfiguration as a Pipeline Stage

2.6.2 Advantages of the New Reconfiguration Mechanism

The aim of the design modification is to introduce run-time customisation of a single multiprocessor template. Additionally, the ease of reconfiguration is also a design concern. Here, the configuration information required to introduce modifications to the template is limited to the control path and datapath information. This is in contrast to configuration information in fine-grained FPGA architectures, where it comprises functional details of the individual configurable blocks and the interconnect information. Since the functional definition and performance of the individual resources of

the processor remain unaltered, the change in performance because of reconfiguration is limited to this additional interconnect.

Execution of a single instruction is sufficient to control the functionality of the reconfigurable interconnects. The layer of interconnects are controlled via instruction set extensions to alter the control and dataflow between the decode, the execute, and the register access stages. These enhancements ensure that the base instruction set architecture is reused and reconfiguration is managed at high-level of abstraction, as suggested using the above-mentioned programming model. Introducing these instructions at boundaries where a change in architecture is anticipated results in reconfiguring the processors. With this scheme of reconfiguration, the reconfiguration information is generated during compilation and introduced during run-time with a reconfiguration overhead of a single cycle. Thus, the need for an additional reconfiguration controller or manager is entirely avoided.

2.7 Summary

In this chapter, architectural aspects viz., methods of processor specification and functional validation have been discussed. These design stages are essential but time consuming and together influence the time-to-market. Considering processor design in particular, methods of accelerating the architectural specification have resulted in faster simulators, architectural models, etc. These methods have in turn reduced the validation time and the process of design verification. In order to avoid frequent redesigns, flexibility to adapt to application characteristics has been introduced via design time and run-time customisations. However, introducing flexibility has a corresponding performance impact. With these issues under consideration, a method of compile-time design space exploration has been presented. To learn from existing commercial multi-core processors, a section studies their advantages and disadvantages. Just as multi-core processors have been successful on account of their inherent parallelism; reconfigurable architectures have been widely used for the flexibility. A section on reconfigurable architectures and customisable multi-core processors attempts to distinguish the architectural features and identify the type of customisation incorporated in each of the processors. Both multi-core processors and reconfigurable architectures have unique features that address two distinct aspects, viz., parallelism and flexibility respectively. The focus of this thesis is two merge these two unique features in a single multiprocessor template. Finally, our concept of the reconfigurable multiprocessor template is presented, which addresses both these aspects. An additional advantage of this template-based design is that it re-uses existing processor cores and introduces minimal architectural changes. This results in reduced design, verification, and validation time. In addition, a low overhead reconfiguration mechanism has been presented

that enables run-time customisation of the reconfigurable multiprocessor template. The reconfigurable template and the quick reconfiguration scheme together provide a mechanism to adapt to application-specific requirements.

Chapter 3

Application

At any time instance, the behaviour of the architecture depends on the application that is being executed. Often, languages and programming models provide mechanisms, constructs, primitives, syntax, and semantics to ease application description. These methods are used to describe tasks, data, and their interactions, which together define the application's overall functionality. For dedicated applications, domain-specific programming languages provide detailed constructs and means to enhance quality, productivity, and reusability via application abstraction for a particular application domain. To aid reusability and code portability, application description is generally maintained agnostic of the target architecture. Finally, for architecture-specific adaptability, mapping and compilation tools transform application functionality onto architecture-specific resource constraints.

In this chapter, Section 3.1 classifies architectures on the basis of their nature of programmability into clock cycle programmable and frequency programmable architectures. In Section 3.2 existing methods of application description and programming models that focus on parallel programming are discussed. This section also includes a discussion on methods of communication and synchronisation inherent in parallel programming models. Application analysis that is independent of the target-specific architectural constraints is made in Section 3.3. Based on this analysis, architecture-independent application characteristics for computation, communication, and synchronisation are extracted in the same section. Architecture-independent application characteristics are compared for three diverse applications in Section 3.4. Additionally, for a set of processors architecture-inherent characteristics are identified. A match between applications and architectures based on their inherent characteristics is suggested. When considering parallel processors, application partitioning makes a significant impact on speedup, power, and energy. Based on these analyses, Amdahl's law is re-calculated and extended with architectural parameters, viz., power and energy in

Section 3.5. Finally, a holistic methodology for application-driven performance optimisation using computation complexity, amount of communication, and frequency of synchronisation is presented.

3.1 Programmability

Programming methods provide the user with abstractions and mechanisms to define the application's functionality. Application mapping aims at customising the underlying architecture to the specific functional requirement, i.e., use the programmable features of the architecture in order to realise the application's functionality. Thus, application mapping encompasses fine-tuning customisable architecture-specific programmable parameters. However, programs are typically oblivious of the target architecture. In this aspect, programmability in architectures has two distinct objectives: *clock cycles of execution* or*operating frequency*.

Processors are representative of *clock cycle programmable* architectures. For applications mapped onto processors, the optimisation objective is typically the number of clock cycles required for execution, since the operating frequency remains fixed. Processors provide a fixed datapath and a programmable control path. The control flow is determined by the sequence of instructions executed. The critical section of the application is the compute-intensive part that dominates the execution time. Further, the memory requirements such as code size and access time for instruction and data also make an impact on application performance. In a multi-core scenario, strategies also involve determining the optimal number of cores to meet the computational requirements for a given application. These features influence the power and energy consumption of the system. Thus, design space exploration for application mapping using processors involves determining the best suitable features of the processor in order to reduce the execution time or the number of clock cycles of execution. Consequently, application description is sequential, aiming at reducing the time required to implement a given functionality. With a fixed operating frequency, the focus is mainly at clock-cycle-based optimisation.

In contrast to clock cycle programmable processors are *frequency programmable* FPGAs. FPGAs provide programmable control and datapaths. Here, the *programmable feature is the frequency of operation*, which varies based on area, which is a contrasting trade-off. The largest combinatorial logic computed in a clock cycle determines the maximum operating frequency and the critical path in an FPGA. Since optimisation in this context involves addressing time - i.e., operating frequency and latency, and area - i.e., resource requirements, it results in a large design space, with multiple objectives. Here, application description emphasises on increasing the maximum frequency

of operation and reducing the gates (or CLBs and memory for FPGAs) required. This strategy aims at a space-frequency optimisation with fixed clock cycles of execution defined in the application source code. The larger design space necessitates strategies with a higher degree of complexity. To aid the parallel application description, the notion of concurrency and parallel task execution is embedded in this programming model (e.g. HDLs).

3.2 Methods of Application Description

The programming model used for application description is dependent on the application domain. The diversity ranges from sequential programming styles, concurrent functional programming, and parallel hardware description languages. Figure 3.1 summarises the diversity and overlaps with existing methods in application description. Programming languages such as C, FORTRAN belong to the class of sequential 'von Neumann' style of programming. These languages are characterised by the use of variables to represent storage elements, control statements, and assignments for fetch, store and arithmetic operations, as described in [42]. These characteristics represent sequential word-at-a-time programming technique. In contrast to 'von Neumann' programming language is functional programming (such as Hasktell, Lisp [43]), where a program is composed of series of state-less functional evaluations, without variables. The mathematical abstraction of program evaluations via functions forms the basis of functional programming, in contrast to imperative programming styles. Occam [44] is a concurrent programming language based on the paradigm of communicating sequential processes (CSP). Applications are described as concurrent functions, with message passing between functions. The application of CSP dates back to the Transputer [45] in 1980s.

Hardware description languages (HDLs) such as Verilog, VHDL are used to model parallel circuits and hardware. The uniqueness in HDLs is the ability to describe an application explicitly as sequential and parallel components. Thus, HDLs have also been used to program applications targeted to FPGAs. The concurrency and parallelism available in hardware is characteristic of this programming model. However, HDLs are distinctly different from sequential programming languages, thus affecting portability of legacy software and fall short of being applied directly to programming parallel processors.

In order to ease the complexity involved in programming FPGAs, electronic system level (ESL) design approaches has been introduced by increasing the input abstraction level. These approaches comprise languages that are subsets of ANSI C (such as ImpulseC [46], HandelC [47]) to ensure ease of programming by reusing the same

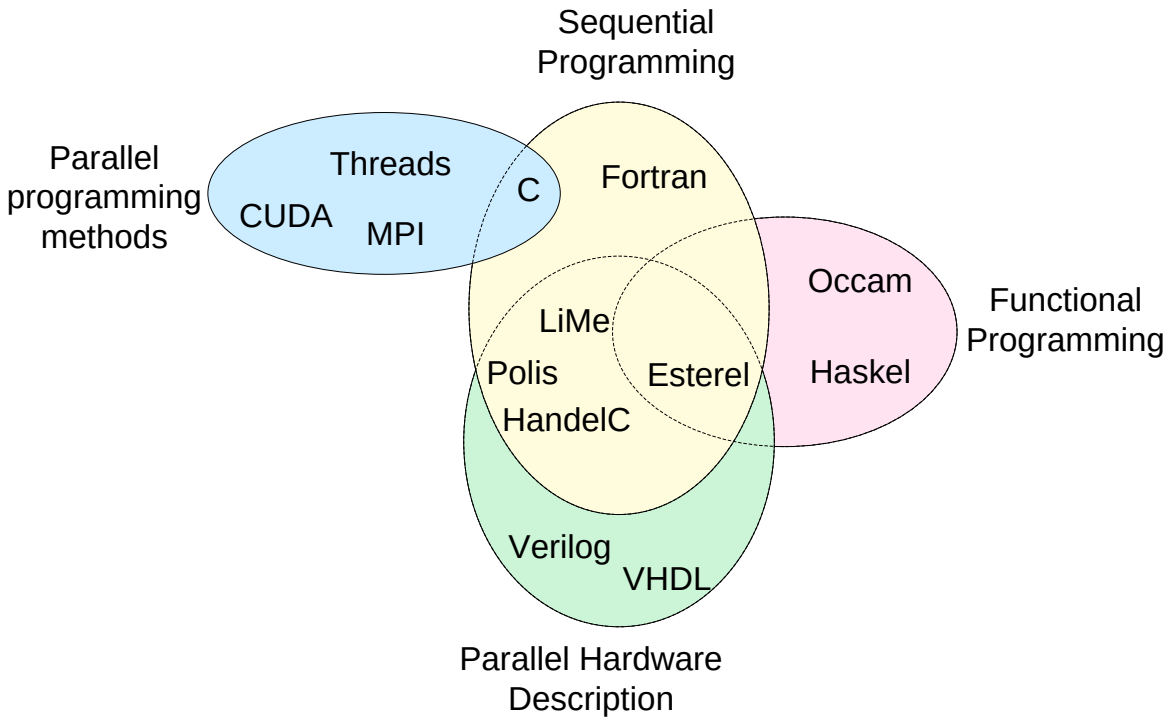


Figure 3.1: Abstractions for Parallel Application Description

programming style in C. In addition to the standard constructs, HandelC uses constructs such as *par* and *seq* to describe distinct parallel and sequential parts of the application, as in Occam.

The automated translation of algorithmic specifications to architectural details is termed as *algorithmic synthesis*. Tools such as Bluespec [48], Cynthesizer [49] from Forte, NEC's Cyberworkbench[50], System Studio from Synopsys, Catapult C [51] from Mentor are examples that aim at transforming high-level algorithmic specifications to architectures. A prominent example is Bluespec, which is a high-level functional hardware description languages aimed at reducing design, development, and verification effort. The abstraction, which is primarily an extension of Haskell (a functional programming language) does not compromise on the quality of the synthesis results [52]. Using atomic transactions, Bluespec automates the generation of control and arbitration logic to manage concurrent access of shared resources.

Stream processing is a programming paradigm that is used for data-centric parallel programming. The main advantage of stream programming is the emphasis on independent operation on local data. Applications use independent functional units, avoiding the need for explicit communication and synchronisation between resources. Stream processing has been extended using vendor-specific languages such as CUDA

for nVidia's graphic processors [53] and Stream SDK for AMD's graphic processors, which extend the usage of graphic processors for general-purpose computing.

Unified programming approaches such as LiMe [54], combines applications description for both processors and application-specific hardware (or hardware-software co-design). Similarly, Polis [55] is an approach for specification, design, and validation environment for unified hardware-software applications. Also Esterel [56] is a synchronous programming languages, that is now capable of generating both C and VHDL descriptions. The uniqueness of such frameworks is the ability to support both clock-cycle programmability and space-frequency adaptability.

Although a range of programming models and languages exist for application description and parallel programming, some questions and methods remain unaddressed:

- Impact of application description on performance: Impact on area, speedup, power, and energy consumption
- Influence of programming language on extraction of parallelism: Diversity between sequential, functional, or parallel programming paradigms
- Expressing granularity of parallelism and its impact on performance: function-, task-, data-, or thread-level parallelism
- Methods for explicit space-time multiplexing

3.2.1 Application Description for Parallel Processors

Parallel programming relies on exploiting application-specific parallelism. Typical paradigms of parallelism include task-, data-, or thread-level parallelism etc. It is necessary to match the inherent characteristics of an application with the target architecture in order to achieve performance efficiency. E.g., a data-parallel application needs an architecture that supports execution of parallel data streams in order to ensure computational efficiency. For such an application, developing efficient methods of synchronisation or communication between the parallel data-streams is not the primary objective. Thus, the decision of how an application is described is entirely dependent on *how an application is expected to behave and what the target architecture should resemble*. In this context, it becomes necessary to expose the application's inherent target-independent characteristics.

Threads, e.g. POSIX threads, provide a mechanism to express concurrent execution within a single process. The basic philosophy governing threads-level parallelism in a single processor scenario is to permit time-multiplexing of tasks in the presence of shared resources. This is a method of forcing non-deterministic parallel execution in a deterministic sequential programming model. Although threads provide the capability

of executing parts of the application concurrently, they exhibit an overhead in having to switch between threads. In a multi-core scenario, threads are executed concurrently on the multiple cores. The main drawback of using thread-level parallelism in a multi-core scenario is the requirement for explicit coherency protocols and a shared memory system, since threads operate on shared resources.

In contrast to the shared memory system of threads, is the distributed memory model of message passing, in Message Passing Interface (MPI [57]). MPI provides a language independent mechanism for parallel computers to communicate. It provides means of topology definition, synchronisation mechanisms, and a means of communication between participating computers, organised with distributed memories. The main advantage of such a system is scalability and portability. This method of expressing applications as independent tasks, which communicate via message passing, targets task-parallel applications. The programming model of communicating sequential processes or CSP extends the concept of message passing to programming embedded processors. In Ambric's processors, message passing is incorporated via FIFO channels and there exists no shared memory. Writes to buffers are stalled when full and the sender stalls until the receiver is ready, as detailed in [58]. This model avoids resource sharing and replaces it by data/control exchange model, thus avoiding costly memory sharing protocols. In place of coherency and context switching between threads, methods to enable fast information exchange is necessitated. Predominantly, these methods are deterministic in nature.

For applications that exhibit fine-grained parallelism, such as DSP algorithms, automatic extraction of instruction-level parallelism is made using VLIW compilation or hardware-driven superscalar processors. Such applications have implicit communication and synchronisation mechanisms. In VLIW processors, synchronisation and communication paradigms are introduced during compilation and the individual processing elements are always lock step synchronous. The main drawback of VLIW processors is the limited ILP that can be extracted, and hence the limited scalability of such processors. In addition, the compilation is entirely responsible for managing the computation, communication, and synchronisation. Similarly, for superscalar processors, data dependencies and scheduling are managed by hardware during run-time. Here, significant amounts of resources are dedicated to run-time management. Further, speedup using these processors entirely relies on acceleration of dependency checks and extracting parallelism within the instruction stream.

3.2.2 Managing Communication and Synchronisation

In this section, methods of programming meant for communication and synchronisation mechanisms are discussed.

In shared memory systems, thread-based programming relies on explicit declaration of shared variables and shared memories. Synchronisation between threads is managed via features such as semaphores and mutexes. These are lock-based mechanisms to avoid corruption of data elements and ensure prioritised use of data. Further, cache coherency protocols ensure consistency between shared memory and the local copy of data. Depending on the language specifics and the mechanism applied, the time required is expected to vary. For example in CUDA, *syncthreads()* ensures that the current thread block waits for all the other threads before exiting this function, in order to ensure data or control protection. Thus, the synchronisation time depends on the arrival time for all the threads at this synchronisation function. Further, it increases linearly depending on number of times the *syncthreads()* is invoked, which corresponds to the number of synchronisation points during program execution. In MPI, synchronisation between tasks is enabled via barrier synchronisation. All participating tasks wait until the barrier is reached before proceeding. The function *MPI_Barrier* requires each process to wait until all other processes in the present task have encountered the barrier. Both these methods are well suited for streaming applications and task/data parallel applications, as limited inter-task synchronisation and communication is expected. Unlike in the above-mentioned extremely parallel applications, this method synchronisation can account to a significant delay when using CUDA or MPI for general-purpose applications with frequent data exchange.

Overall, synchronisation primitives or functions are governed by the following parameters:

- Time required to synchronise n participating elements
- Frequency of synchronisation (how often is synchronisation required)
- Number of elements that can be synchronised at once (scalability)

Communication between threads is managed via explicit declaration of shared variables or shared regions of memory, protected via lock based mechanisms. In MPI, send and receive are functions in-built, where passing data between processes initiates communication. Additionally, MPI also supports collective communication mechanisms such as scatter-gather that enables broadcasts to all and receives from all. Thus, the necessary parameters that govern the quality of the communication mechanism include:

- Time required for exchange of data/control among participating elements
- Frequency of communication
- Number of elements that can benefit from this method of communication or scalability of the mechanism

3.2.3 Drawbacks of Existing Methods

Typically, to aid portability application description is made independent of the application mapping and the architectural constraints. Since the extraction of parallelism is done prior to application mapping, a significant impact on performance is to be expected on account of the architectural constraints. Consider thread-level parallelism, although parallel threads are defined during application description, the number of concurrently executing threads depends on the processors capability and the processor load at that particular instance. Further, the overheads involved in thread stalls and resumes are architecture-specific. In addition, executing threads in a multi-core scenario demands the use of shared caches and cache-coherency policies for data-coherency. Further, since the performance for an application expressed in thread-level parallelism is non-deterministic, the speedup when using multi-cores cannot be predicted a priori. The same issues arise when using MPI or other schemes, where the target architecture may have widely differing characteristics.

To understand the importance of application-dependent characteristics, the following sections characterise applications with respect to computation, communication, and synchronisation requirements. Based on these characteristics, the performance impact of including additional processing elements is analysed with the focus on time, power, and energy reports.

3.3 Architecture-Independent Application Characteristics

Applications are composed of collection of tasks executed in a particular sequence, often describing an algorithm. These tasks have specific input and output demands, computational complexities, inter-task communication, and storage requirements. These functional requirements are independent of the target architecture. Thus, an application is composed of a finite set of states representing the control flow, and a fixed set of computational characteristics corresponding to the data-flow requirements. Depending on the target architecture and its resource constraints, the resulting performance is determined.

Application-specific characteristics need to match architectural constraints in order to meet performance requirements and ensure resource efficiency. Here, we define attributes that are representative of the application, irrespective of the application domain and the target architecture. Then, the objective is to analyse the impact on performance depending on the architectural attributes. These characteristics are

classified on varying degrees of granularity of parallelism and computational complexity by decomposing the application into its parallelisable functional sub-units.

In processors, typically, an application is translated into a series of instructions; the sequence in which they are executed is determined by the control flow. Also in processors, the data-flow is fixed, but control flow is programmable. The schedule of instructions determines the sequence at which they are executed. Conventional scheduling for in-order processors relies on time-based resource multiplexing. Increasingly as transistors get smaller and area becomes cheaper, space multiplexing is expected to take over time multiplexing, where scheduling in time can be replaced by scheduling in space. In such a scenario, representing applications as collections of space-multiplexed functions in place of sequences of time-multiplexed instructions becomes necessary. The following sections the characterisation of applications is independent of these two models. Hence, it is applicable for application mapping in both these domains.

3.3.1 Model for Computation

Application-level characteristics and the underlying architectural features together determine the performance of the system. Here, the application-level formulation determines the total time required for all the computational operations in an application. The total computation time is given by:

$$T_{comp} = \sum_{i=1}^{k} C_{comp_i} \cdot N_{comp_i} \cdot \frac{1}{f_{comp_i}} \tag{3.1}$$

In the above equation, the total computation time (T_{comp}) is the total time required to compute all the operations (k), where N_{comp} is the number of computational operations, viz., the number of instructions, tasks, or processes depending on the granularity of computation. The execution time of each operation is expressed as a product of the number of clock cycles required for computation (C_{comp}), the number of such operations (N_{comp}), and the reciprocal of operating frequency of the computational unit (f_{comp}) for every operation i.

In a sequential computing model (such as processors), C_{comp_i} corresponds to the number of clock cycles required for an instruction type i, N_{comp_i} relates to the number of such instructions in the application, and f_{comp_i} in this model is the processor's operating frequency. However, f_{comp_i} can be variable for each computation operation i. As an example, consider a computation function that translates to 10 addition operations and that each addition operation requires 1 clock cycle. In that case, the parameters for Equation 3.1 correspond to : $N_{comp} = 10$, $C_{comp} = 1$, $k = 1$, and f_{comp} is the processor's operating frequency.

In a parallel computing model (such as FPGAs), C_{comp} is the execution time for a task. In the best case C_{comp} can be a single clock cycle, where the entire task can be computed in a single clock cycle due to the availability of abundant parallel resources. In this case, assuming just one instance of the task to suffice, $N_{comp} = 1$, and the operating frequency (f_{comp}) is determined by the critical path of the computational block. Thus, the Equation 3.1 can be used for varying granularities and computing models.

Similarly, the total power consumed for all the computational operations ($P_{total-comp}$) is given by:

$$P_{total-comp} = \sum_{i=1}^{k} P_{comp_i} \cdot N_{comp_i} \tag{3.2}$$

where, N_{comp} is the number of computations and P_{comp} is the power requirement per computation i, and k is the total number of computations for a particular application. Task execution using multiple processing elements leads to lower computational load per processing element (lower P_{comp} per task i), but the power consumption increases (with increase in cumulative N_{comp}). Overall, although partitioning an application to multiple compute elements reduces the execution time, power scales linearly with N_{comp}. Thus, overall impact on energy is nullified, unless a reduction in time is higher than the number of additional N_{comp} (super-linear speedup).

In a sequential model used in processors, P_{comp_i} corresponds to the instruction-level power for instruction type i and N_{comp_i} is the number of instructions of type i, where k is the total of instruction types. In a parallel computing model, P_{comp_i} corresponds to the power consumed by the task i and N_{comp} is the number of such tasks.

When using clock-cycle programmable architectures, application description can influence the number of operations (N_{comp}) required for computation. Whereas, the number of cycles required for computation (C_{comp}) and frequency of operation (f_{comp}) remains unchanged, as they are fixed for a given processor architecture. The number of cycles required for computation depends on the availability of computational elements, the functionality of the computational elements, and the number of operations required to fulfill the given functionality. E.g., this number can be altered by employing multiple processing elements. Similarly, for an FPGA or an ASIC, where the architectures adheres to the application's characteristics the timing characteristic are identical for the two architectures. However, the power characteristic is dependent on the architectural features such as frequency of operation. This explains why the execution time of an application implemented as an ASIC and FPGA concur but not the frequency of operation and the power characteristics, as listed in Table 2.1.

3.3.2 Model for Synchronisation

For parallel computations, the synchronisation mechanism employed determines the overhead involved before the actual control or data exchange is initiated. To generalise the timing model for synchronisation, it may be termed as the time required for synchronising parallel operations, where the time required for synchronising each operation may be variable. The application's functionality determines the amount of synchronisation and the architecture-specific mechanisms determine the delays involved for each synchronisation. Hence, the total synchronisation time or T_{sync} is given by:

$$T_{sync} = \sum_{i=1}^{k} C_{sync_i} \cdot N_{sync_i} \cdot \frac{1}{f_{sync_i}} \tag{3.3}$$

where, N_{sync} is the number of times the parallel operations need to synchronise and is dependent on the application characteristics and its computational granularity. C_{sync} is the number of clock cycles required for each synchronisation operation and it is dependent on the underlying architecture and the associated delay involved in the arrival time for all tasks to the synchronisation point. k corresponds to the number of synchronisation schemes available for every synchronisation instance i. f_{sync} is the operating frequency for each synchronisation mechanism i. Typically, in processors the operating frequency is the same for computation, communication, and synchronisation. As an example for application characteristics that determine synchronisation consider extremely data-parallel applications. For such applications there is no exchange of information between parallel tasks, which implies $N_{sync} = 0$.

Similarly the total power consumption for synchronisation $P_{total-sync}$ is given by :

$$P_{total-sync} = \sum_{i=1}^{k} P_{sync_i} \cdot N_{sync_i} \tag{3.4}$$

where, P_{sync} is the power consumption for each mechanism i and N_{sync} is the number of such synchronisation operations.

Thus for a fixed architecture, with a single synchronisation scheme ($k = 1$), the estimation of synchronisation time and power depends on the transaction counts or N_{sync}. Optimising for time and power involves minimising the application-dependent component N_{sync}, which results in minimising both the total power consumption and execution time.

3.3.3 Model for Communication

The number of inter-task data or control exchange is based on the granularity of tasks and number of parallel tasks. The total communication time is defined as the time required for exchange of data (or control) for all the communication operations. Variations in the inter-task communication are entirely dependent on the architecture, available bandwidth, and the associated physical resource constraints. E.g., the delays involved in a bus-based infrastructure are protocol dependent and implementation dependent. Whereas, for multi-ported, shared register files there is no communication delay involved in data exchange. However, the number of such transactions required is specific to application characteristics. Thus, the communication time is expressed as:

$$T_{comm} = \sum_{i=1}^{k} C_{comm_i} \cdot N_{comm_i} \cdot \frac{1}{f_{comm_i}} \tag{3.5}$$

where, T_{comm} is the total communication time and N_{comm} is the number of times the parallel operations exchange data and is dependent on application characteristics and its granularities. Since there could exist multiple communication schemes even within a single architecture, k is a parameter for each type of communication i. The number of clock cycles required for each communication operation is given by C_{comm} and is architecture-dependent (implementation-specific). E.g., it corresponds to the delays in a bus-based access or in a point-to-point communication. f_{comm} is the operating frequency of the communication operation.

Power characteristics for the communication component ($P_{total-comm}$) may be given by :

$$P_{total-comm} = \sum_{i=1}^{k} P_{comm_i} \cdot N_{comm_i} \tag{3.6}$$

where, N_{comm} is dependent on the application-specific attributes. E.g., the frequency of communication or data-exchange depends on the degree of parallelism. P_{comm} is architecture-specific and relates to the communication mechanism. E.g., for VLIW processors the power consumed for data-exchange is a register file read (and/or write), whereas for a bus-based multi-core it includes the communication mechanism (bus or switch, etc.) followed by the read/write mechanism.

3.4 Comparing Application-specific Attributes

Attributes such as granularity of tasks, amount of inter-task communication, and frequency of synchronisation identify application-specific attributes. An application is composed of computing clouds inter-connected via communication and synchronisation mechanisms. The type of computing elements and the communication infrastructure is determined by the target architecture. However, application specification and computational granularity influences the execution time and power consumption even in the absence of the architectural influence, as seen in the previous sections.

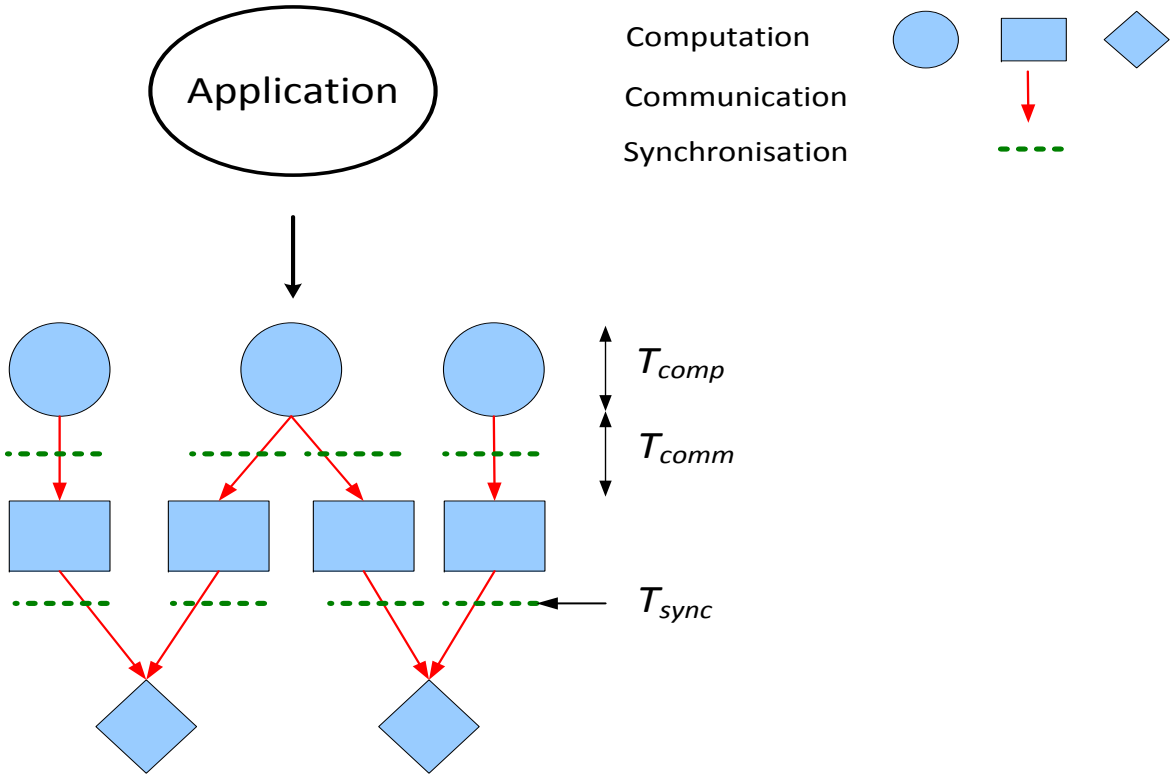


Figure 3.2: Generic Timing Components

In this section, we classify applications based on computation requirements and its associated communication and synchronisation overheads. Figure 3.2 is representative of the components of an application. The quantitative relation between the amount of computation, communication, and synchronisation is a decision making factor for choosing the appropriate architecture. Although the model of the communication protocol can be deterministic, the dependence of communication and synchronisation on computation and vice-versa influence the performance accuracy of the models.

Equation 3.7 shows the total execution time for an application (T_{total}) with k tasks, with T_{comp} as the execution time for each task and T_{comm} as the communication time between tasks, and T_{sync} as the synchronisation time between tasks.

$$T_{total} = \sum_{i=1}^{k} (T_{comp_i} + T_{comm_i} + T_{sync_i}) \tag{3.7}$$

For an application with *fine-grained tasks*, *frequent inter-task communication*, T_{comp} corresponds to instructions and k represents the total number of instructions, as given by Equation 3.7.

$$T_{total} = \sum_{i=1}^{n} T_{comp_i} + \sum_{i=1}^{k} (T_{comm_i} + T_{sync_i}) \qquad |n \leqslant k \tag{3.8}$$

For an application where the tasks are composed of multiple such instructions, the application can be subdivided into multiple tasks or functions. Applications could comprise multiple tasks, where the granularity of a task comprises multiple such instructions. In this case, T_{comp} represents the execution time for tasks and n represents the total number of tasks and k is the associated inter-task communication and synchronisation. These applications are representative of task-level granularity, given by Equation 3.8.

$$T_{total} \approx \sum_{i=1}^{n} T_{comp_i} \qquad | \sum_{i=1}^{k} (T_{comm_i} + T_{sync_i}) \approx 0 \tag{3.9}$$

For data-level parallelism, tasks operate independently on exclusive data-sets, with equal length tasks and no inter-task communication or synchronisation, as represented in Equation 3.9. For an application with *thread-level parallelism*, T_{comp} corresponds to instructions in a thread and k represents the total overhead for switching between threads. However, this delay is non-deterministic since it depends on the caches policies, cache hit/miss rates, thread stalls etc.

In the following sections, for a set of applications attributes these attributes, viz., amount of computational requirements, communication demands, and frequency of synchronisation are extracted. As will be seen, these attributes are representative of the type of application and the granularity of parallelism.

3.4.1 DSP Applications

In this section two applications, viz., matrix multiplication and FIR filter, commonly used among other DSP applications are analysed for their computation, communi-

cation, and synchronisation characteristics. These characteristics are inherent to the algorithm and independent of the target architecture.

Matrix Multiplication

Consider multiplication of two $n \times n$ square matrices A and B, where the resulting product matrix C is given by:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j} \tag{3.10}$$

where each element of the resulting matrix requires n product computations and $n-1$ summations. The algorithm itself operates on a total of $2 \cdot n \cdot n$ data elements for computation and an additional $n \cdot n$ data elements for storing the result. The resource availability and the granularity of computation define the amount of communication and synchronisation within the algorithm. As seen in Table 3.1, the application exhibits zero communication and synchronisation demands, hence well suited for application partitioning onto multiple elements with computation as the priority. Additionally in this case, the performance is unaffected by the target-dependent communication and synchronisation schemes.

Table 3.1: Matrix Multiplication: Comparing Computation, Communication, and Synchronisation

Granularity	N_{comp}	N_{comm}	N_{sync}
n Multipliers, $n-1$ Adders	n^2	0	0

FIR Filter

Realising an FIR filter is given as follows by Equation 3.11 and the realisation is shown in Figure 3.3. As seen, the first stage of computation requires n multiplications. A stage of data exchange or communication follows this stage of computation in order to perform a summation operation. A synchronisation operation is also necessary to avoid any violations. The final stage is a summation operation. This results in a total of $n+1$ computations (N_{comp}), one communication per computation element (N_{comm}), and one synchronisation operation (N_{sync}).

$$y(i) = \sum_{k=1}^{n} a(k) * x(i-k) \tag{3.11}$$

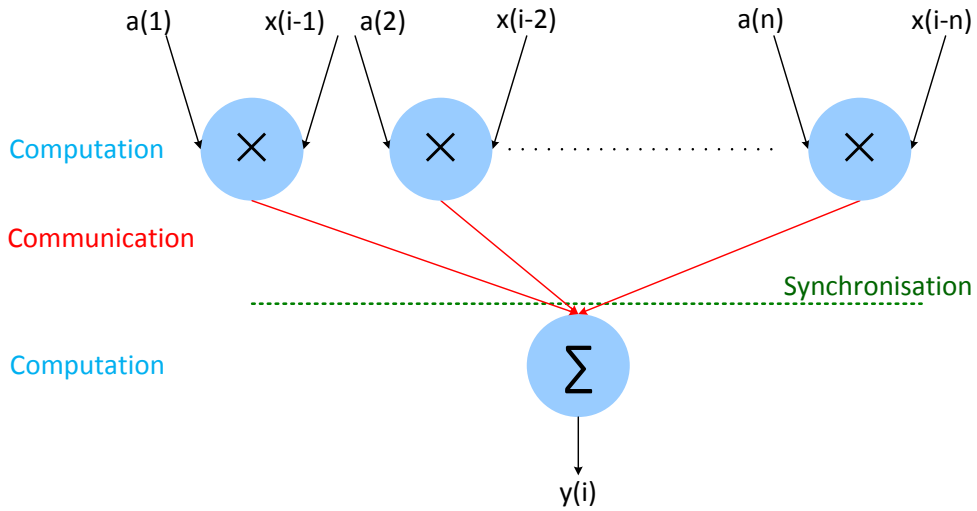


Figure 3.3: FIR Filter: Computation, Communication, Synchronisation

where *a(k)* is the coefficient of the FIR filter at tap k, *x(i)* is the input, *y(i)* is the output at time i and k is the length. The computation and communication demands are dependent on application's complexity and is proportional to n, as shown in Table 3.2. Whereas, synchronisation requirement is negligible in comparison to the other components.

Table 3.2: FIR Filter: Comparing Computation, Communication, & Synchronisation

Granularity	N_{comp}	N_{comm}	N_{sync}
1 Multiplier, 1 Adder	$n+1$	n	1

Hence in this case, the computation and communication infrastructure of the underlying architecture play an important role in achieving optimal performance. The synchronisation requirement is a minor component for the overall execution time.

3.4.2 Multiplier used in Elliptic Curve Cryptography

The finite field multiplication in $GF(2^{233})$ is represented as follows:

$$\begin{aligned} A(x) &= \sum_{j=0}^{n-1} A_j \cdot X^{jw} \\ B(x) &= \sum_{j=0}^{n-1} B_j \cdot X^{jw} \\ C &= A \cdot B \end{aligned} \tag{3.12}$$

where $n = 8$ and $w = 32$. Here, w is word width and n is 8 for a 233-bit word length. Using Karatsuba [59] method iteratively, the multiplication of binary polynomials of degree 232 can be calculated with 27 finite field multiplications at word-level. The algorithmic realisation is represented in Figure 3.4, where the base nodes are the multipliers (represented in blue). Considering the computations, it has 27 tasks with inter-task dependencies as shown in the figure. In terms of communication, the first stage has 27 data exchanges, followed by 9 in the second stage, and 3 exchanges in the final stage (represented in red). To avoid data corruption, the first stage requires 9 synchronisations, the next stage needs 3 synchronisations, and finally one synchronisation towards the end (represented in green). Identifying the optimal granularity of the tasks is essential to obtain the number of computations (N_{comp}), the number of inter-task communications (N_{comm}), and synchronisations (N_{sync}).

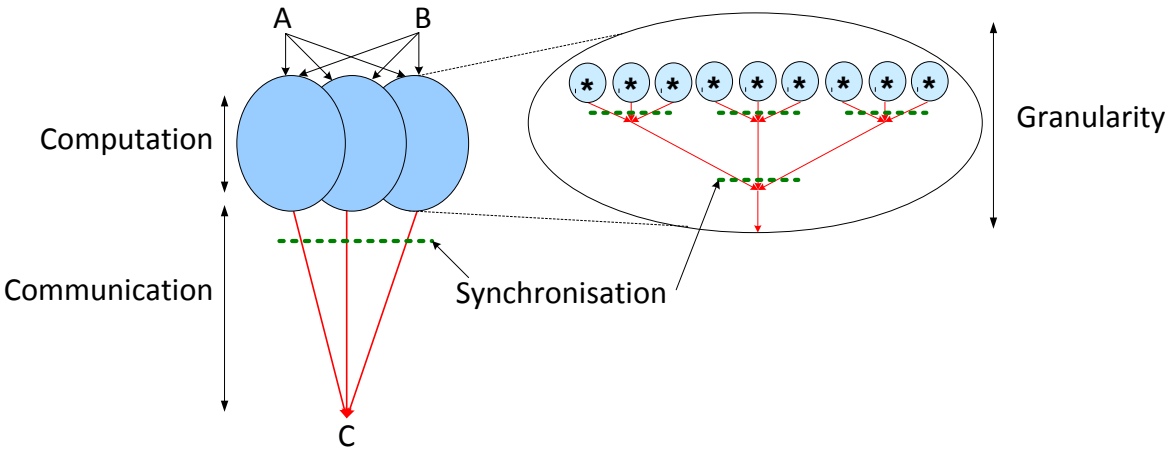


Figure 3.4: ECC: Computation, Communication, Synchronisation

Table 3.3: ECC: Comparing Computation, Communication, and Synchronisation

Granularity	N_{comp}	N_{comm}	N_{sync}
Tasks = 1	27	39	13
Tasks = 9	3	3	1

The number of communication and synchronisation operations varies depending on the granularity of the computational blocks. Table 3.3 shows the variations in numbers for varying granularity. The optimal granularity is chosen such that the number of computations is either equal or greater than the inter-task communication. Hence, when a granularity of 9 tasks is chosen, the inter-task communication reduce to a total of 3 exchanges and a single stage of synchronisation.

3.4.3 Self-organising Maps

Self-organising maps are neural network based machine learning algorithms used for data analysis for high dimensional data sets [60]. A SOM comprises an N-dimensional grid of neurons or processing elements that are adapted to an input data-set X.

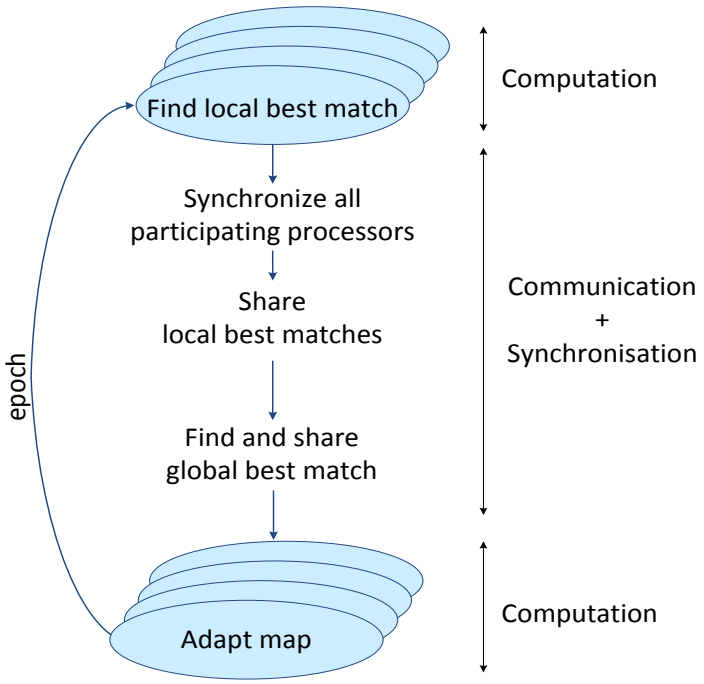


Figure 3.5: Parallelising the SOM Algorithm

This is a case for data parallel algorithm execution and is divided into the following three steps:

1. Initialise: The weight vectors $\overrightarrow{m_i}$ are initialised for all neurons N
2. Locate Best match: A vector $\overrightarrow{x}(t)$ is randomly selected from X and the distance between $\overrightarrow{x}(t)$ and all $\overrightarrow{m}_i$ is calculated. The neuron with the shortest distance to the input is called the best match.
3. Adapt Map: The weight vectors $\overrightarrow{m}_i$ are adjusted towards the input $\overrightarrow{m_i}$ based on a neighbourhood function (often a Gauss-kernel)

Steps 2 and 3 are repeated for all vectors $\overrightarrow{x} \in X$, where each iteration for all vectors is called one epoch. Depending on the requirements of the dataset, several such epochs may be required to form a properly organised map. Thus, the execution time of the SOM algorithm is mainly dependent on the number of neurons, the number of vectors, the dimension of the vectors and the number of epochs, as shown in Figure 3.5 and described in detail in [PPPR09].

For simplification the total execution time (T_{total}) is given by:

$$T_{total} \propto |N||X| \dim(\overrightarrow{x}) \cdot epochs \tag{3.13}$$

In terms of computation, all the $|N|$ neurons perform the same set of operations, hence the parallelism only limited by the target architecture. The sequential communication and synchronisation that is incurred is a very small portion of the code [61]. To explore data-level parallelism, two possible dimensions can be explored. Table 3.4 shows the characteristics categorised as computation time, communication demand, and corresponding synchronisation requirement for both the approaches. The computation requirements are linearly proportional to the number of neurons, vectors and the number of epochs. Similarly, the algorithm is composed of two computation intensive tasks, where exchange of data is expected. Hence, the amount communication and the frequency of synchronisation are both proportional to the number of epochs and the approach chosen.

Table 3.4: Self-organising Maps: Computation, Communication, Synchronisation

Approach	N_{comp}	N_{comm}	N_{sync}
Neuron Parallel	$\|N\|\|X\| \dim(\overrightarrow{x}) \cdot epochs$	$\|N\| \cdot epochs$	$\|N\| \cdot epochs$
Vector Parallel	$\|N\|\|X\| \dim(\overrightarrow{x}) \cdot epochs$	$\dim(\overrightarrow{x}) \cdot epochs$	$\dim(\overrightarrow{x}) \cdot epochs$

3.4.4 Priorities: Computation, Communication, or Synchronisation

Figure 3.6 shows the variations in the computation, communication, and synchronisation requirements among the above discussed applications. As can be seen, the choice of the target architecture needs to consider the application's computation, communication, and synchronisation demands. E.g., matrix multiplication is dominated by computational demands. Since synchronisation and communication demands are absent, it can be mapped even onto architectures that have high communication and synchronisation costs. Whereas, an FIR filter requires an infrastructure with fast communication and synchronisation mechanism, since the contribution is significant and cannot be neglected. For the ECC application, fast computation and communication mechanisms are essential when the granularity of tasks is low, since the cumulative effect of the two dominates the execution time. The synchronisation time is not a primary concern, since it is a minor component. As the granularity of tasks increase, a corresponding reduction in the synchronisation and communication costs are seen.

However, both computation time and communication time need to be taken into consideration for this application. For the SOM algorithm, as the application exhibits data-level parallelism operating on large amounts of data, a higher granularity of tasks are chosen to reduce the amount of inter-task communication and synchronisation requirements. Hence, in this case, the cost of synchronisation and communication are a second-level priority in comparison to the computation infrastructure.

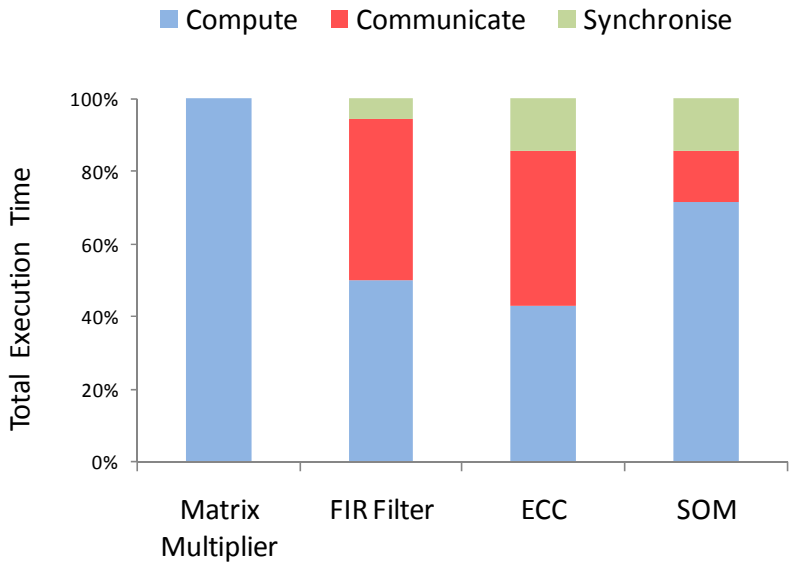


Figure 3.6: Application-Specific Computation, Communication, and Synchronisation

Matching Application Characteristics to Architectural Attributes

Table 3.5 compares the inherent computation, communication, synchronisation infrastructures in some existing processors. The processors chosen are the massively parallel processor Ambric, a VLIW processor developed at the research group called CoreVA, and the QuadroCore multiprocessor discussed in this thesis. In order to compare these processors, the parameters discussed in Section 3.3 were chosen. For the processors under consideration, N_{comp} corresponds to the number of simultaneous computations, N_{comm} is the number of simultaneous communications, and N_{sync} is the number of simultaneous synchronisations supported by the architecture at any given time instance. These parameters only consider the architectural features for simultaneous operations, and do not consider the implementation details viz., clock cycles required and the operating frequency.

The application shown in Figure 3.2 has been mapped onto a single processor, a multi-issue fine-grained processor resembling a VLIW processor, and multiple copies of a processor forming a multiprocessor, like QuadroCore. Figure 3.7 compares the

Table 3.5: Comparing Parallelism among Processors

Architecture	N_{comp}	N_{comm}	N_{sync}
Ambric [32]	8	8	1
CoreVA [62]	4	4	4
QuadroCore [PPR+]	4	4	4

variations in the computation, communication, and synchronisation characteristics for changes in architectural characteristics.

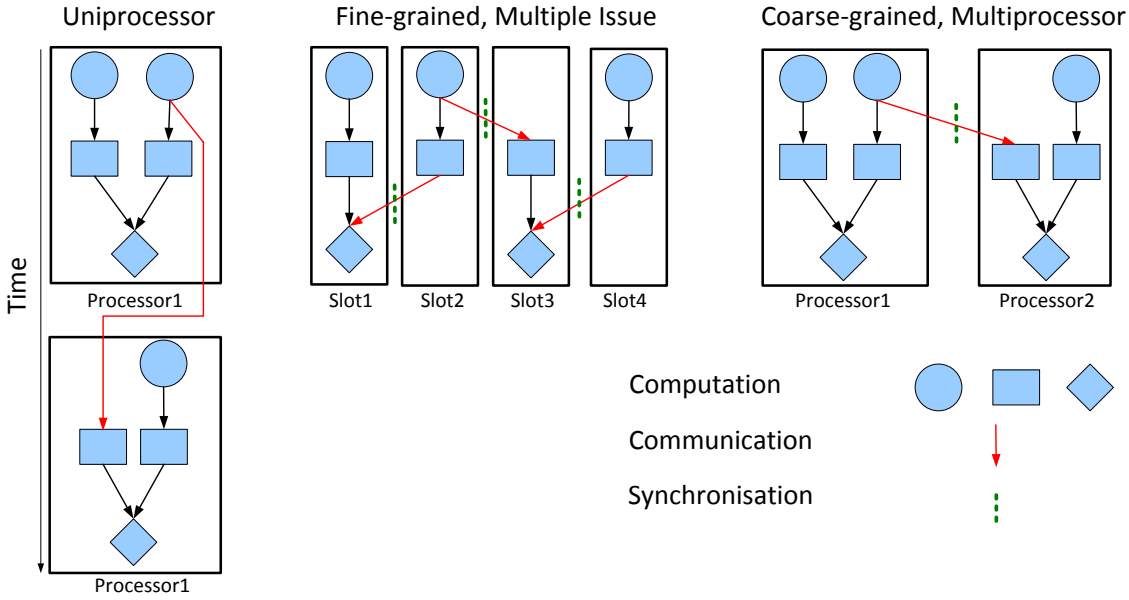


Figure 3.7: Comparing Processors with Variable Granularities

For Ambric, each tile comprises eight programmable elements, hence has an N_{comp} of 8. Each tile has eight parallel communication channels, which corresponds to an N_{comm} of 8. The asynchronous mechanism of synchronisation corresponds to one synchronisation between processing elements at any given time instance, hence has an N_{sync} of 1.

For our VLIW processor CoreVA, which has four parallel ALUs, the N_{comp} is 4. The shared register file and a lock step synchronisation mechanism corresponds to an N_{comm} of 4 and an N_{sync} of 4.

The QuadroCore processor also has four processing elements, and is representative of chip multiprocessors (CMP). The processors exchange data via a shared multi-port register file. This infrastructure corresponds to an N_{comp} of 4 and an N_{comm} of 4. Additionally, up to four processors can be synchronised simultaneously, which makes $N_{sync} = 4$.

Even in the presence of a fixed frequency of operation and identical application characteristics (N_{comp}, N_{comm}, and N_{sync}), the architecture-specific parameters (C_{comp}, C_{comm}, and C_{sync}) influence the choice of applications best suited for an architecture. At this stage, the number of pipeline stages that affects frequency of operation, which is implementation-specific is not taken into consideration. To match these architectural metrics, the application characteristics are categorised into applications with instruction-level parallelism (ILP), task-level parallelism (TLP), and data-level parallelism (DLP). Thus, the most optimal architecture for a given application is chosen such that the computational operations dominate over the communication and synchronisation overhead, also stated in Equations 3.7, 3.8, and 3.9. This can be obtained by ensuring $T_{comp} \gg T_{comm} + T_{sync}$.

Table 3.6 lists the differences for the three specific processors, where the difference in the C_{comm} and C_{sync} changes the matching application domains. Here, the granularity of computation is chosen at instruction-level, resulting in a single-cycle instruction execution. With the correct choice of communication and synchronisation mechanism Ambric can be used for instruction-, task-, and data-parallel applications. The CoreVA VLIW processor is ideal for fine-grained instruction-level parallelism or in applications with repeated identical operations like in data-parallel applications. Finally, as can be seen, the QuadroCore is well suited for task- and data-level parallelism.

Table 3.6: Comparing Clock Cycles among Processors

Architecture	C_{comp}	C_{comm}	C_{sync}	**Matching Applications**
Ambric	1	1-4	1	ILP, TLP, DLP
CoreVA	1	0	0	ILP, DLP
QuadroCore	1	4-15	1	TLP, DLP

Adaptive mapping strategies to modify these parameters depending on application characteristics (N_{comp}, N_{comm}, and N_{sync}) are discussed later in Section 4.3.

3.5 Restating Amdahl's Law

The process of partitioning has to ensure a that the computation time dominates over the communication and synchronisations times. The computation time is specific to the architecture of the computing element or the processing element. E.g., it refers to the processor's microarchitecture (RISC, superscalar) and memory structure in a multiprocessor organisation. In addition to the computation architecture, the communication infrastructure contributes to the communication delay, which also means that merely accelerating the compute engines is not sufficient, the communication and

memory needs to scale accordingly. Similarly, the frequency of synchronisation also influences the overall performance improvement. This is in addition to the effort of analysing power and energy issues by extending Amdahl's law in [63, 64]. Our analysis is generic and only considers issues in parallelism affected by application partitioning. Here, architectural constraints such as frequency of operation and device voltage remain unaltered.

3.5.1 Speedup: Comparison to Amdahl's Law

Consider an application mapped onto N processors, where the parallelisable portion of its total execution time is given by T_{par} and the non-parallelisable portion (or the sequential portion of the parallelisabe part) given by T_{seq}. As per Amdahl's law, apart from the sequential component (T_{seq}) the parallel component (T_{par}) reduces with increase in N (which is the number of processing elements), given by the following equation:

$$Speedup = \frac{T_{seq} + T_{par}}{(T_{seq} + \frac{T_{par}}{N})} \tag{3.14}$$

The overhead involved because of data-distribution, collection, memory access, and the associated synchronisation times introduced on account of application partitioning may be termed as the overhead of partitioning. Thus, Amdahl's law can be re-written as follows:

$$Speedup = \frac{T_{seq} + T_{par}}{(T_{seq} + \frac{T_{par}}{N} + T_{sync} + T_{comm})} \tag{3.15}$$

where, T_{seq} is the sequential component of the total execution time, T_{par} is the parallelisable component of the total execution time. T_{sync} and T_{comm} are the non-parallelisable overhead in time, incurred on account of application partitioning. In other words, for a parallelisable portion p, and a corresponding sequential portion $(1-p)$, the speedup is given by:

$$Speedup = \frac{1}{1 - p + \frac{p}{N}} \tag{3.16}$$

Figure 3.8 shows the impact of the timing overhead of partitioning algorithms onto four processing elements. The X-axis shows the percentage of time overhead in the application and the Y-axis shows the speedup observed. Impact of increase in communication and synchronisation code (additional non-parallelisable code) due to application partitioning is plotted. The execution time of the communication and synchronisation

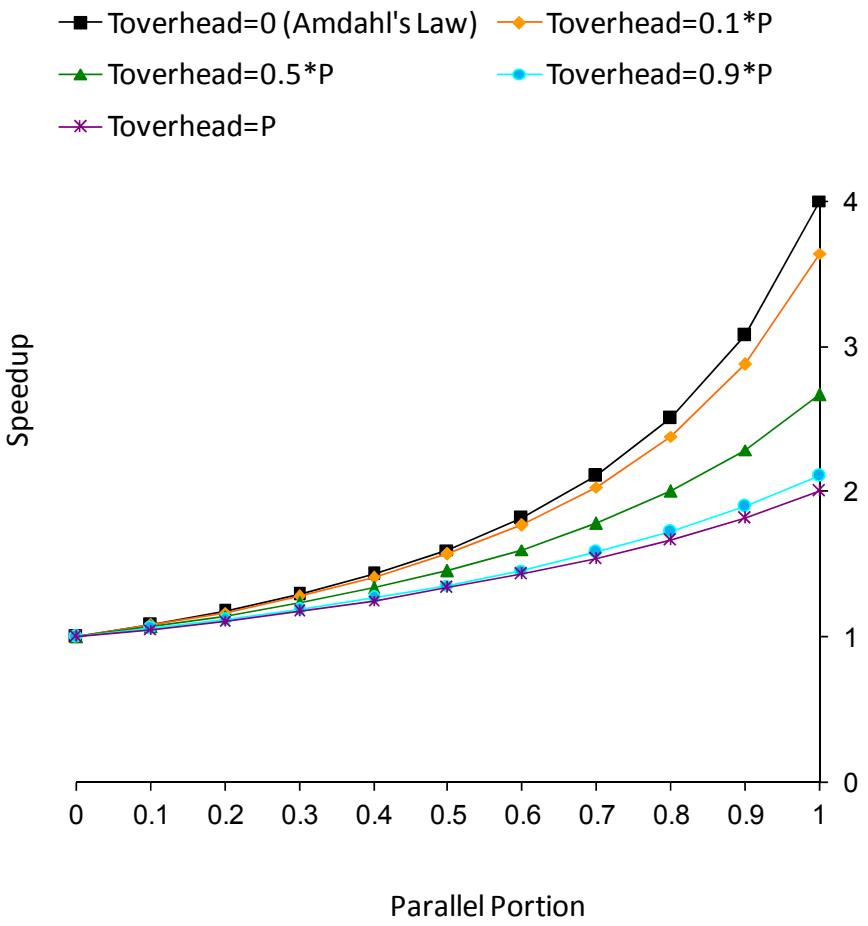


Figure 3.8: Impact of Sequential Code on Application Partitioning

overhead component ($T_{overhead}$) is assumed to be a fraction (0.1, 0.5, 0.9, and 1.0) of the total parallelisable portion (P). Additionally, the total communication and synchronisation code increases with the increase in the number of processors. As seen, with an increase in overhead, the observed speedup deteriorates significantly. It also shows that the sequential overhead on account of added parallelism cannot be ignored. A performance deterioration compared to the original sequential execution is observed, when the overhead of parallelism exceeds the achieved speedup. As seen in the plot, an increase in the number of processors only makes an impact when the amount of parallelism is high, irrespective of the amount of serial code. Further the same characteristics are noticed for an increase in the number of processors, resulting in increased overhead of parallelisation (viz. synchronisation and communication times). Apart from the constant sequential component in any algorithm, a non-parallelisable component that comes into play for every additional processing element included during application partitioning. This additional component contributes to the overall speedup computation.

As an example, consider partitioning an FIR filter onto the 4-processor QuadroCore using the new reconfiguration design space. The equation for realising an FIR filter is given as follows by Equation 3.11.

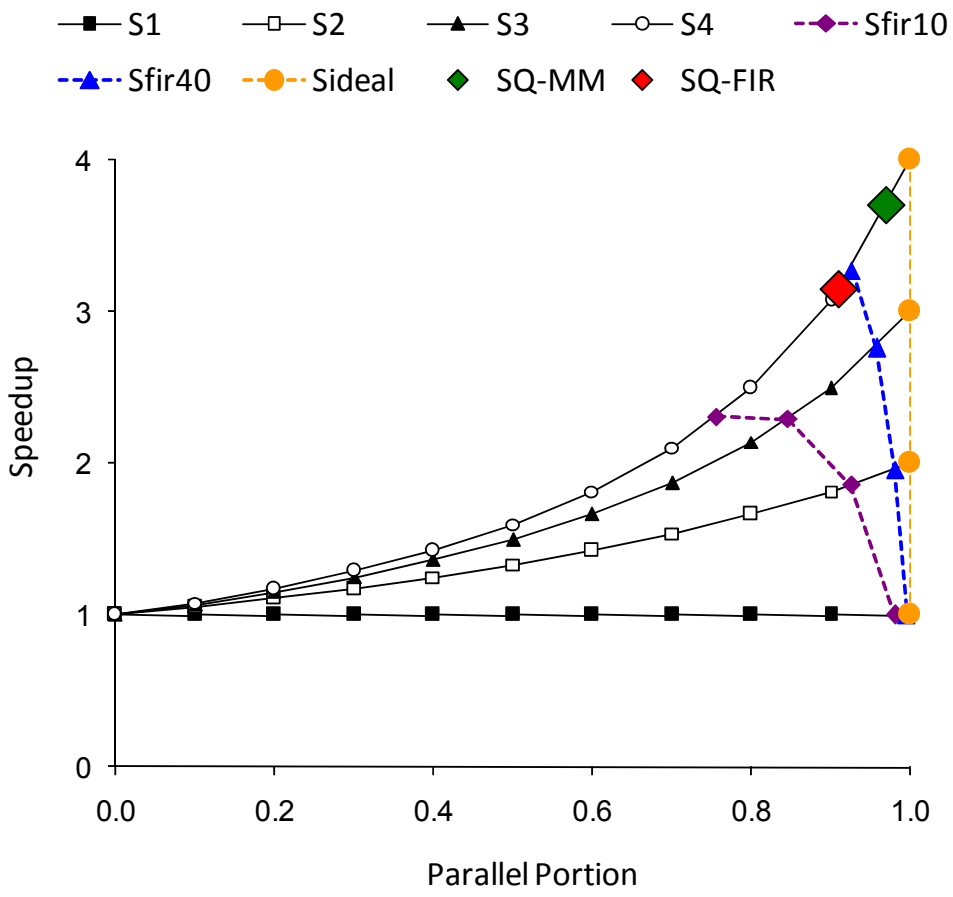


Figure 3.9: Impact of Application Mapping on QuadroCore

Figure 3.9 shows the overlay of the classical Amdahl's law with increasing number of processors. The theoretical estimation of speedup achieved using Amdahl's law applied to one, two, three, and four processors are represented by $S1$, $S2$, $S3$ and $S4$ respectively. The ideal case of N-fold speedup is represented by the plot S_{ideal}, which also represents the case of a matrix multiplier with data elements present locally for each of the processors. The speedup achieved for a 10-tap FIR filter is represented by S_{fir10} and for a 40-tap filter by S_{fir40}. The intersection of the plot S_{fir10} (and S_{fir40}) with the graphs for Amdahl's law coincide with the parallelisable fraction for one, two, three, and four processors respectively. As can be seen, with increase in the number of processors, the parallelisable fraction decreases. The change in the serial to parallelisable ratio is accounted for the additional communication and synchronisation overhead incurred on account of application partitioning. The actual implementation

reports obtained on mapping the two applications (viz., matrix multiplication and fir filter) on the QuadroCore multiprocessor are shown as solid points S_{Q-FIR} and S_{Q-MM}, which coincide with the plot $S4$. A slight deviation in the implementation reports and the theoretical estimations may be accounted to the variations in the shared memory access times on account of bus arbitration.

3.5.2 Power: Comparison to Amdahl's Law

Power consumed in digital CMOS circuits is given by:

$$P_{total} = \alpha(C_L \cdot V_{dd}^2 \cdot f_{clk}) + I_{SC} \cdot V_{dd} + I_{leakage} \cdot V_{dd} \tag{3.17}$$

where α is the probability of signal transition or the application-specific switching activity, C_L is the loading capacitance, V_{dd} is the supply voltage, f_{clk} is the operating frequency, I_{SC} is the short circuit current, $I_{leakage}$ is the leakage current.

For a constant operating frequency, the application-specific inputs influence the switching activity or α. Application partitioning distributes the activity over a number of processing elements. Thus, the total power consumption or dissipation is proportional to the number of processing elements used. Consequently, power increases linearly as the number of processors. Although, using additional processing elements reduces the amount of work done by each processor and hence the execution time of each of the processors, the overall power consumed scales linearly. Further, if additional power is consumed due to the communication and synchronisation overhead, power consumption increases accordingly.

Considering p as the parallelisable portion of the application being executed on a cluster of N processors, the total power (P_{total}) consumed is a combination of the parallel execution component and a sequential component given by:

$$P_{total} = p \cdot P_{par} + (1 - p) \cdot P_{seq} \tag{3.18}$$

where, P_{par} is the power consumed by the parallelisable portion of the application, which contributes to a percentage p of the total power consumption. P_{seq} is the power consumed by the sequential portion and it contributes to a ratio of $(1 - p)$ times the total power consumption.

In Equation 3.18, the assumption is that a single processor is sufficient for executing sequential code and only the parallel component is partitioned onto multiple processors. Additionally, during the execution of the sequential code, static power of the unused parallel processors is assumed to be negligible. Further, during the execution of sequential code, only one processor dissipates power and the unused processors do

not contribute to power. In this equation p is the ratio of the parallel portion in the original application.

3.5.3 Impact on Energy

The combined impact of time and power influences energy calculations. Thus, the total energy (E_N) consumed by N identical processors, is given by:

$$E_N = [P_{seq} \cdot T_{seq}] + [P_{par} \cdot \frac{T_{par}}{N}] \tag{3.19}$$

where, P_{seq} is the power consumed by the sequential component of the application, P_{par} is the power consumed in the parallelisable portion, and N is the number of parallel processing elements.

Energy consumption during execution of the entire application on a single processor is given by E_1:

$$E_1 = P_1 \cdot [T_{seq} + T_{par}] \tag{3.20}$$

where, P_1 is the power consumed on one processor.

Ideal Scenario

In the ideal case, leakage power, and additional dynamic power is absent when the sequential portion of the algorithm is executed. Additionally, the partitioning of application onto the N processors does not incur an addition power or communication or synchronisation timing delay. In this scenario, the power consumption for N identical processors are the summation of power for all the processors executing the parallelisable portion of the code and the sequential code is executed on one of the processors, given by:

$$\begin{aligned} P_{par} &= P_1 \cdot N \\ P_{seq} &= P_1 \end{aligned} \tag{3.21}$$

where, P_1 as the power consumed by one processor.

Similarly for Time,

$$\begin{aligned} T_{par} &= \frac{p}{N} \cdot T_1 \\ T_{seq} &= (1-p) \cdot T_1 \end{aligned} \tag{3.22}$$

Combining Equations 3.18, 3.19,3.20, 3.21, and 3.22,

$$E_N = [(1-p) \cdot P_1 \cdot T_1] + [N \cdot P_1 \cdot \frac{p}{N} \cdot T_1] \quad (3.23)$$

Comparing energy consumption of N processors to a single processor,

$$\frac{E_N}{E_1} = [1] \quad (3.24)$$

The change in energy or savings in energy is absent when partitioning applications onto N processors, even in the presence of a N-fold speedup.

Non-ideal Scenario

In this case, during the execution of sequential code on one of the processors the power consumption is not entirely cut-off on the unused processors. Hence, the resulting energy consumption is the same as that of a single processor, in the best case scenario. Additionally, in the best case it is assumed that $T_{sync} + T_{comm} \approx 0$, $P_{sync} + P_{comm} \approx 0$. In Figure 3.10, variations in time, power, and energy for increasing percentage of parallel component are plotted. $time(1)$, $time(2)$, $time(3)$, and $time(4)$ represent the execution time on one, two, three, four processors respectively. Similarly, $energy(1)$, $energy(2)$, $energy(3)$, and $energy(4)$ represent energy consumption. The time plots ($time(1)$, $time(2)$, $time(3)$, and $time(4)$) follow Amdahl's law, where the best performance is observed in the absence of any serial component and the entire application can be partitioned equally among the available processors. Thus, time(4) shows the best speedup. For the power, it is a linear relationship to the number of processors, where power consumption scales with the number of processors. As a consequence, the best energy reports observed from the energy plots ($energy(1)$, $energy(2)$, $energy(3)$, and $energy(4)$) is in the case of $energy(4)$, where the time reports are the best. However, as can be seen, the energy characteristic in the best case of performance speedup ($energy(4)$) is merely as good as a single processor ($energy(1)$). The benefits of speedup are lost on account of the multi-fold energy increase offset. Additionally, these plots entirely ignore the overhead of partitioning, which further worsens performance gains and corresponding energy savings.

Thus, following are the conclusions based on the formulations for time, power, and energy:

- Time: Energy savings can be achieved when the speedup for N processing elements is greater than N, i.e. a super-linear speedup. However, extracting these performance speedups is not always easily possible.

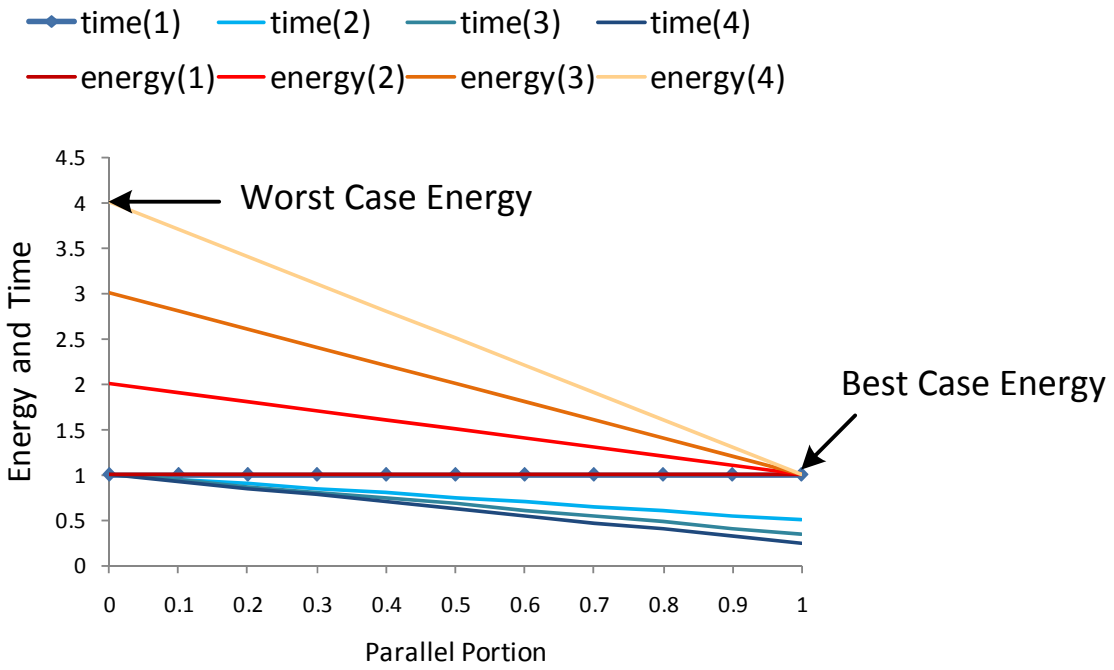


Figure 3.10: Analysis of Time, Power, Energy Characteristics

- Power: Increasing the number of processor results in a linear increase in power. Thus, in the absence of voltage and frequency scaling strategies, increasing the number of processors does not account to any energy savings.

Hence, it is essential to introduce new strategies, both in terms of time and power, in a parallel processing scenario. These methods need to introduce energy savings even with increasing number of processors. The following chapters focus on application, architecture, and mapping methods to achieve energy savings in multiprocessor architectures.

3.6 Summary

Based on the previous discussion it is apparent that application requirements need to be extended beyond computational characteristics. For applications mapped onto multiple processing elements, it needs to include the communication and synchronisation overhead incurred due to application partitioning. During application description, architecture independent application characterisation helps to identify the appropriate architectures. Such an application-driven optimisation needs to take into account the following considerations:

- Considerations for run-time matching of computational, communication, and synchronisation changes for application to architectural mapping
- Introduce optimisation strategies that consider the overhead of application partitioning and are independent of the underlying instruction set architecture
- Methods to alter the architecture to match application requirements and vice-versa
- Introduction of power characterisation to make an overall impact on energy consumption

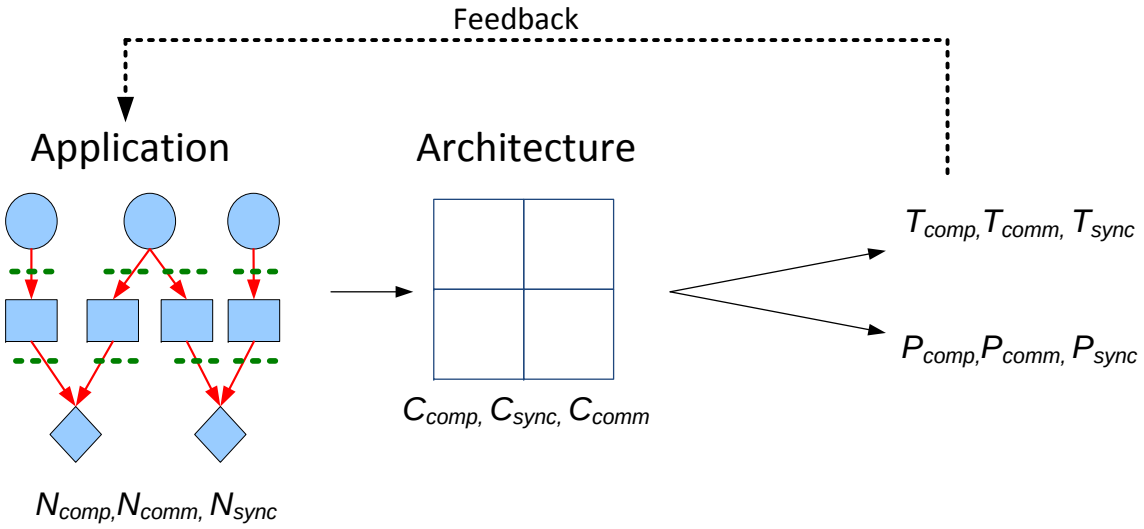


Figure 3.11: Feedback for Application Modification

Figure 3.11 summarises the method of application description and feedback-driven optimisation strategy. Based on the feedback obtained, application-specific description such as task-granularity, inter-task communication, and consequently synchronisation can be fine-tuned. These performance characteristics enable efficient application partitioning onto multiple processing elements. In addition to timing characterisation for computation, communication, and synchronisation, it becomes essential to consider the power components for each of the above cases. In the following chapters, architectural optimisations that address the above open questions are discussed. In QuadroCore, architectural features enable power optimisation such that the overall energy consumption for a four-processor cluster is lower than the energy consumption of a single processor, even in the absence of a super-linear speedup.

Chapter 4

Application to Architectural Mapping

Transforming application-level description to meet architectural constraints constitutes application-to-architectural mapping. This stage binds the two independent domains, viz., application description and architectural constraints, also described in the Y-chart [65]. Generally, application description is distinctly independent of the underlying hardware. This approach is adopted to maintain universality, retain target-independent code description, and provide code portability across architectures. Hence, mapping is target-specific and is responsible for converting the application described to suit the architecture.

In processor architectures, high-level programming abstractions in languages such as C, aid in application description. After gauging the application's complexity, selecting the appropriate number and type of resources is a decision made during application mapping. Compilation transforms the target independent application description to combinations of instructions, defined in the processor's instruction set architecture. Target-specific constraints are introduced at this stage. Since the processor's programming model relies on sequential processing, scheduling decides the time instance at which a resource is used based on the instruction executed. The instruction memory stores the sequence of instructions that need to be executed to accomplish the application's functionality with the corresponding data elements residing in the data memory. The levels of memory hierarchy for the data flow exists between the register file, as first level, the scratchpad memory, caches, and finally the main data memory.

With increasing complexities and transistor densities, application-specific integrated circuit designs typically involve hardware synthesis. This is a process of translating the high-level functional description to gate-level specification using process and technology specific libraries. This translated netlist is placed and routed, using elaborate design tools to meet performance constraints. This process is followed by a huge design time for chip fabrication. This entire process has a large design time, verification

time and consequently a large time-to-market. However, the single most important performance advantage is the optimal application-to-architecture translation. The resulting architecture is fine-tuned for a particular application, thus resulting in optimal performance characteristics such as time, area, power.

FPGAs are off-the-shelf devices that avoid the delay and costs involved in fabrication. User-defined applications and application updates can be introduced to the available device, without having to fabricate each time. However, mapping in this context involves transforming applications described in hardware description languages to suit the architectural constraints of the target FPGA architecture. Customisations are introduced by configuring these devices, which shortens the time-to-market. Configuring, in this context, involves customising the FPGA's components using HDL synthesis, followed by placement, and routing of the synthesised design. Configuring an FPGA involves defining the functionality of the individual configurable functional blocks and programming the routing structures connecting them.

In this chapter, we focus our discussion to application mapping in devices that can be customised in the post-fabrication phase, viz., processors and FPGAs. Mapping strategies for these two devices address two diverse domains, viz. time and space. The following section associates architectures and applications to provide a better understanding of the diversity in mapping strategies for variations in applications and architectural features. Design space exploration in this context involves finding the best match between a given application and its target architecture. In this context, Section 4.2 details and compares the objectives for application mapping in the two design flows, viz., compilation and FPGA mapping. This discussion leads to a method of merging the two design flows for applications mapped onto our reconfigurable multiprocessor QuadroCore. A generic technique to merge two conventional mapping styles of spatial and temporal design flows is discussed. Section 4.3 presents a technique that uses reconfiguration as a method for adaptive mapping in our reconfigurable multiprocessor. Two schemes of adaptive mapping, viz., static and dynamic reconfiguration are presented in this section. Finally, Section 4.4 summarises the entire chapter.

4.1 Applications and Architectures: Fixed vs. Alterable

The application's performance requirements, compatibility, time-to-market, and ease of application mapping for the application that needs to be executed decides the target architecture. Figure 4.1 summarises the diversities that arise in application to architecture mapping. These paradigms are limited to possibilities, where in-field modifications are possible. Hence, they are compile-time or run-time modifiable architectures

and applications. This classification includes programmable processors and reconfigurable FPGAs and excludes design-time configurable processors such as Tensilica's Xtensa [11], SiliconHive [30], as they require a phase of device fabrication.

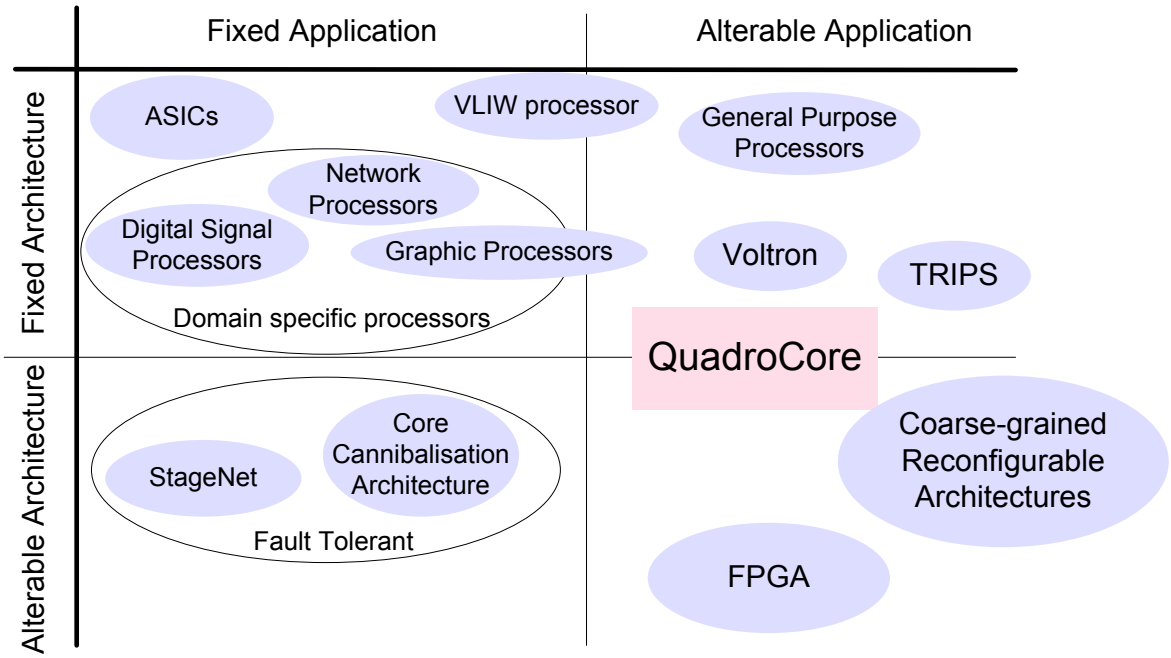


Figure 4.1: Architecture and Application Diversities : Fixed versus Alterable

4.1.1 Fixed Applications, Fixed Architecture

The top-left quadrant represents the deterministic case of a one-to-one translation of a fixed application to an architecture. Typically, the constraint of a single application requires no flexibility or in-field programmability for application-level variations. A fixed architecture generally suits such a scenario, leading to the design of Application-Specific ICs.

In contrast, if a fixed application domain is targeted, it can be classified as domain specific architectures. Examples include graphic processors, network processors, and digital signal processors designed to match specific application domains. They comprise customised blocks such as programmable shaders in graphic processors, pattern matchers in network processors, and fast multiply-accumulate units in digital signal processors. These additional features distinguish domain specific processors from general-purpose processors. On account of their customisations, the performance and resource efficiency is fine-tuned for that particular application domain.

The advent of GPGPU [66] (general-purpose processing using graphic processors) is a cross-domain use of domain-specific architectures for general-purpose computing.

The hardware accelerators or special function units in graphic processors can be used to accelerate certain non-graphic applications. For example, the use of shaders to implement matrix multiplication is a case of non-graphic application. These initiatives demand increasing programming support and hardware flexibility to ease general-purpose programming.

In short, these processors have:

- Domain-specific programmable features and reusable application libraries
- Programmability limited to domain-specific parameters
- Absence of portability between application domains

4.1.2 Alterable Applications, Fixed Architecture

In order to reuse a single architecture over a varied range of application characteristics necessitates designing flexible general-purpose processors. The case where the processor architecture is not customised to a single application, results in a relative loss in performance for adaptability to large range of application-level diversity. The advantage of such architecture is the ease of programmability, with high-level programming language support, easy mapping support, and backward compatibility. This is possible at a compromise on performance, a necessary trade-off for a generalisation. Further, application-mapping process needs to conform to application-specific performance. For example, general-purpose processors operate at very high frequencies to compensate for the additional clock cycles required to suit the diversity in application functionality.

With a restriction in the number of resources available, the application description and the mapping process need to manage the available resources efficiently. This leads to introducing mechanisms for effective scheduling in order to manage disparity in the resource requirement and resource availability. Typical examples of general-purpose processors are the x86 processors. The time multiplexing of resources demands higher operating frequencies, which in turn introduces a gap between the processor and the memory's operating frequencies. Thus, demanding a need for fast memory access mechanisms like caches and scratchpads. These requirements in turn justify the need for programming language support, such as threads-level parallelism, as mechanisms to hide the memory access delays via parallel execution.

Fixed architectures like TRIPS [35] and Voltron [67], which adapt to application-specific granularities and parallelism also belong to this classification. QuadroCore can also belong to this classification as a range of applications can be mapped on to a single multiprocessor template. The processor's instruction set architecture is fixed;

hence, it belongs to this category. However, QuadroCore has additional features that includes it in the class of alterable architectures, as discussed in sections that follow.

To summarise, the questions that arise in a domain with alterable applications and a fixed architecture are:

- Area versus frequency trade-off to meet functional requirements
- In-field programmability and adaptability
- Functional validation or prototype implementation
- Domain-independence and flexibility

4.1.3 Fixed Application, Alterable Architectures

In a real world scenario, in-field architectural variations are inevitable during device deployment. The basis of these architectural variations can be categorised into three types. Firstly, an intentional architectural variation may be introduced for performance enhancements, to adapt to newer technologies, or newer generations of processors. In order to support application mapping irrespective of the change in underlying hardware, the mapping scheme needs to adapt accordingly. In this case it is advantageous that application description is independent of the target architecture. Retargetable compilation addresses this capability of performance optimal code generation for a varied set of target architectures. Target-specific strategies are introduced to meet performance demands based on the new resource constraints. Existing open source initiatives such as SUIF [68], LLVM [69], and open source initiative called Trimaran [70], are target-independent compilation frameworks.

The second type of architectural variation is in the form of environmental changes encountered on account of faults. Physical faults may deter normal device operation. Additionally, faults may also occur due to device aging and process variations. Thus, the mapping tool needs to address device reliability and ensure fault free application translation. These operational impediments lead to introducing techniques such as self-healing in the presence of additional redundant resources or methods of graceful degradation to extend device operation in the presence of faults. Architectures such as Core Cannibalisation Architectures [71], StageNet [72] are examples of processors with enhancements to withstand faults encountered during device operation.

Finally, in the third case, performance indicators introduce changes in the architectural features. These performance indicators include device temperature, power variations, frequency versus energy trade-offs etc, which are primarily encountered in field, during device run-time, or prolonged device usage. For the same application description, mapping tools need to incorporate an in-field re-mapping strategy based on the feedback

obtained from the specific performance indicators. Performance-driven re-mapping includes strategies for power-aware scheduling, dynamic voltage, and frequency scaling, avoiding hot spots that are recognised during run-time. Strategies for run-time adaptations are identified and modified solely by the mapping algorithm.

Thus, mapping tools for in-field adaptability have the following requirements:

- Capability to adapt to varying architectural characteristics
- A fast, run-time mechanism for adaptability, with very little performance overhead

4.1.4 Alterable Applications, Alterable Architecture

This is a special case of universal flexibility for both applications and architectures. Changes in application characteristics introduce changes to the architecture or vice-versa.

Flexibility to adapt to application-specific changes is an essential feature for architectural prototyping. These architectures are suitable candidates for design space exploration, during run-time. Design space exploration extends beyond arriving at the best architecture, to also identifying the range of applications that may be well suited for an architecture and vice-versa. The adaptive mapping strategy enables continuous fine-tuning of the architecture to adapt to performance and environmental conditions

FPGAs are another extreme case of general-purpose processing. With programmable resources available in abundance, these devices can be programmed to suit frequency and timing requirements, which are otherwise limited using sequential processors. With the same features of programmability, and area as the incurred trade-off, resources are mapped in parallel to meet the speed requirements. In addition, reconfiguration mechanism aid in time multiplexing of the resources available. However, present day schemes of reconfiguration typically incur a large overhead. To circumvent the reconfiguration overhead, coarse-grained reconfigurable architectures such as ADRES [73] and Tilera [74] have been designed. These architectures can be adapted to variations in application characteristics and hence can be categorised alongside FPGAs.

QuadroCore is also representative of such architectures, since it adapts to application-dependent variations. With a single ISA, it is extensible over run-time to suit to varying ISAs. This concept of *architectural prototyping* encourages usage in scenarios that require functional validations of ISAs, for example, mapping a new instruction set onto the base instruction of the QuadroCore. This technique resembles the case of cross-compilation, code morphing, and virtualisation.

The following features characterise this case of application-to-architecture co-relation:

- Fixed ISA, changeable data and control path
- Introduces run-time alterations to the architecture
- Continuous fine-tuning

4.2 Application Mapping: Objectives and Methods

Application mapping is a continuous process, which is generally steered by the application-specific performance objectives. Diverse performance objectives such as area, time, energy etc., together determine the quality of application mapping. For a fixed architecture, these performance reports provide a feedback to alter the application description and mapping strategies. For example, code-generation for processors is directed towards optimising code size, time, or power. Each of these objectives has contrasting performance impact; hence need to be considered in combination. For a given application, depending on the mapping objective variations in performance can be observed. Even in the presence of a fixed architecture and definite application considerations, mapping objectives introduce variations in the achieved performance. Processors and FPGAs are two diverse architectures that can be steered with distinct performance objectives, in the post-fabrication phase. Here, the focus is on processors that represent a sequential processing model and FPGAs representing parallel execution model. In the following sections the two mapping strategies, viz., compilation flow for processors and the FPGA design flow are detailed, with a comparison of computational complexity of the individual phases for both the design flows.

4.2.1 Compilation Flow

A standard processor design flow for transforming application description to architectural specifics — called compilation is shown in Figure 4.2. As described in Chapter 3, application description is often made entirely independent of the target processor architecture. The process of compilation has to ensure error-free transformation of the application code to the target processor code.

Firstly, program analysis introduces static modifications to the application code and transforms it to an intermediate representation. The optimisations introduced at this stage are target-independent and referred to as frontend optimisations. These transformations include strategies such as dead code elimination, copy propagation, and common sub-expression elimination. The goal of this stage is to introduce optimisations to eliminate redundant code and reduce the number of instructions required to replicate a high-level program. These passes are often found both in HDLs and in sequential programming languages.

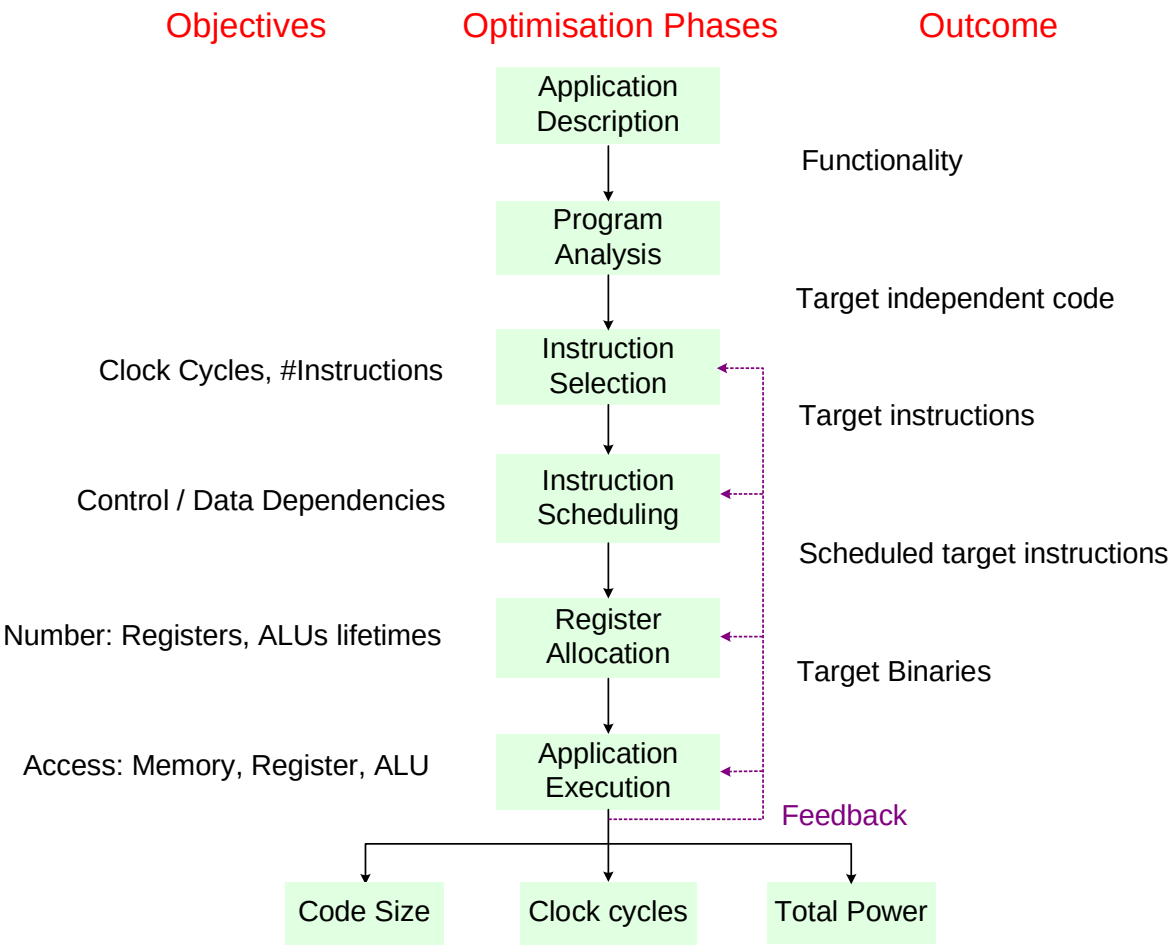


Figure 4.2: Compilation Flow

The resulting intermediate representation of control and data flow is then transformed using a series of target-dependent optimisation passes. Instruction selection ensures transforming the intermediate representation to the target instruction set architecture. This is a tree-pattern matching process; ensuring optimal set of instructions is obtained for every specific application code. The intermediate representation is a tree defining the control and data flow of the application. The processor's instructions represent the target patterns. The goal of tree pattern matching is to meet the performance objective, while realising the application's functionality. The required time depends on the complexity of the tree and the number of patterns available to cover a tree pattern. This transformation is the first stage of application-to-architectural mapping, where the target processor's characteristics define the bounds of mapping. The exploration space encompasses patterns of instructions and the cost function to evaluate the quality of tree cover for instruction selection.

As a next step, these instructions are scheduled in order to share resources in a time-multiplexed fashion. The objective of scheduling is to associate a time component to

resource availability in order to meet application-specific resource demands. It ensures that a given resource is available at a given time, while meeting the application's timing dependencies. In the processor's sequential model, resource constrained scheduling is performed on these instructions to meet the target processor's resource constraints. Meeting timing requirements and ensuring data-dependencies are the objective of the scheduler. Resource allocation associates the instruction's resource requirements to the processors resources. Additionally, memory and register allocation phases ensure minimising memory accesses and efficient utilisation of the available registers. The objective of register allocation is to maximise the utilisation of registers and ensure minimal register-spills. Both these objectives aim at minimising execution time of the application code. Scheduling and resource allocation are together responsible for time-multiplexing the application's resource requirements among the finite set of resources available in a processor. Furthermore, processor code-generation is restricted to sequential programming model, where majority of optimisation revolves around ensuring efficient time multiplexing of the finite set of resources.

Feedback : Optimal application-to-architectural mapping using the compilation flow involves identifying the sequence of instructions that result in efficient resource utilisation. In order to introduce a user-driven steering of application mapping that enables altering the performance objectives, a feedback-driven mechanism is shown in Figure 4.2. Feedback steers the objective of each of the stages based on time, code size and power reports obtained as a consequence of the intermediate stages of the design flow. This feedback-driven design flow is predominantly done via manual intervention in present day compilation techniques. An automated, iterative compilation flow enables accelerated application mapping.

Computational Complexity in Compilation

Instruction selection, scheduling, and register allocation are the individual compute intensive tasks during compilation. Bottom-up rewrite system or BURS [75] is a standard technique used for instruction selection from the intermediate language representation. Here, in the first bottom-up pass, all low cost patterns that match the intermediate code are labelled. The second pass is top-down traversal that identifies the best sequence of instructions to match the patterns. In the third and final pass, the instruction patterns selected in the second pass to output a sequence of instructions to replicate the original tree-grammar. Further details can be found in [76]. The complexity involved in the process of instruction selection is proportional to the nodes in the source tree (m) and the nodes in the target tree (n). Thus, it has a maximum time complexity of $O(mn)$ for pattern matching, [77].

The next stage of resource constrained scheduling is an established case of *NP*-complete problems [78]. The register allocation phase is based on live variable analysis. The variables in the application code are mapped to the limited set of registers available in the processor. Here, the life span of a variable in a register decides reusability of a register. Register allocation is an *NP*-complete problem and heuristics such as graph colouring are used to minimise the register requirements.

Consequently, the computational complexity of the compilation process in processors is a function of number of instructions, number of resources, number of registers, and the pipeline stages.

4.2.2 FPGA Flow

The application mapping flow, typically used in FPGA-based designs is shown in Figure 4.3. For designs that use FPGAs, translating application functionality or circuit description for the target FPGA is application mapping. Firstly, the HDL-based application description is translated into a target independent netlist via the process of logic synthesis. This stage of logic synthesis involves language-specific optimisations such as common sub-expression elimination, operator re-ordering, resource sharing, and loop unrolling. This netlist is then mapped onto FPGA-specific components (such as LUTs, flip-flops, and memory) via technology mapping. These FPGA resources define the configurable space for realising diverse functionalities. Clustering ensures grouping of technology-specific elements into the same basic logic block, called configurable logic blocks in Xilinx FPGAs. This stage aids placement, which identifies locations for the technology-specific elements on the FPGA area. The stage of routing follows placement of resources, where the interconnections between the placed resources are established. Each of these transformations needs to ensure that the functionality remains unaltered. The performance objectives focus typically on the combined optimisation of maximum delay, area, and the total power consumed.

Hardware compilation differs from standard FPGA design flows, where the application description is made using software programming languages such as C. Here, the objective is to map applications described in C or other sequential high-level languages to hardware that are inherently parallel in nature, such as FPGAs. This process of mapping requires transforming the notion of time in application description to space (or resources) in the target architecture.

Scheduling in time with resource constraints is transformed into a case of scheduling to meet frequency constraints in the presence of abundant resources, only limited by the FPGAs resources. Examples of C-to-FPGA initiatives include from c-to-silicon compiler from Cadence [79], and AHIR [80]. FPGA-mapping has expanded its design space with the inclusion of partial run-time reconfiguration.

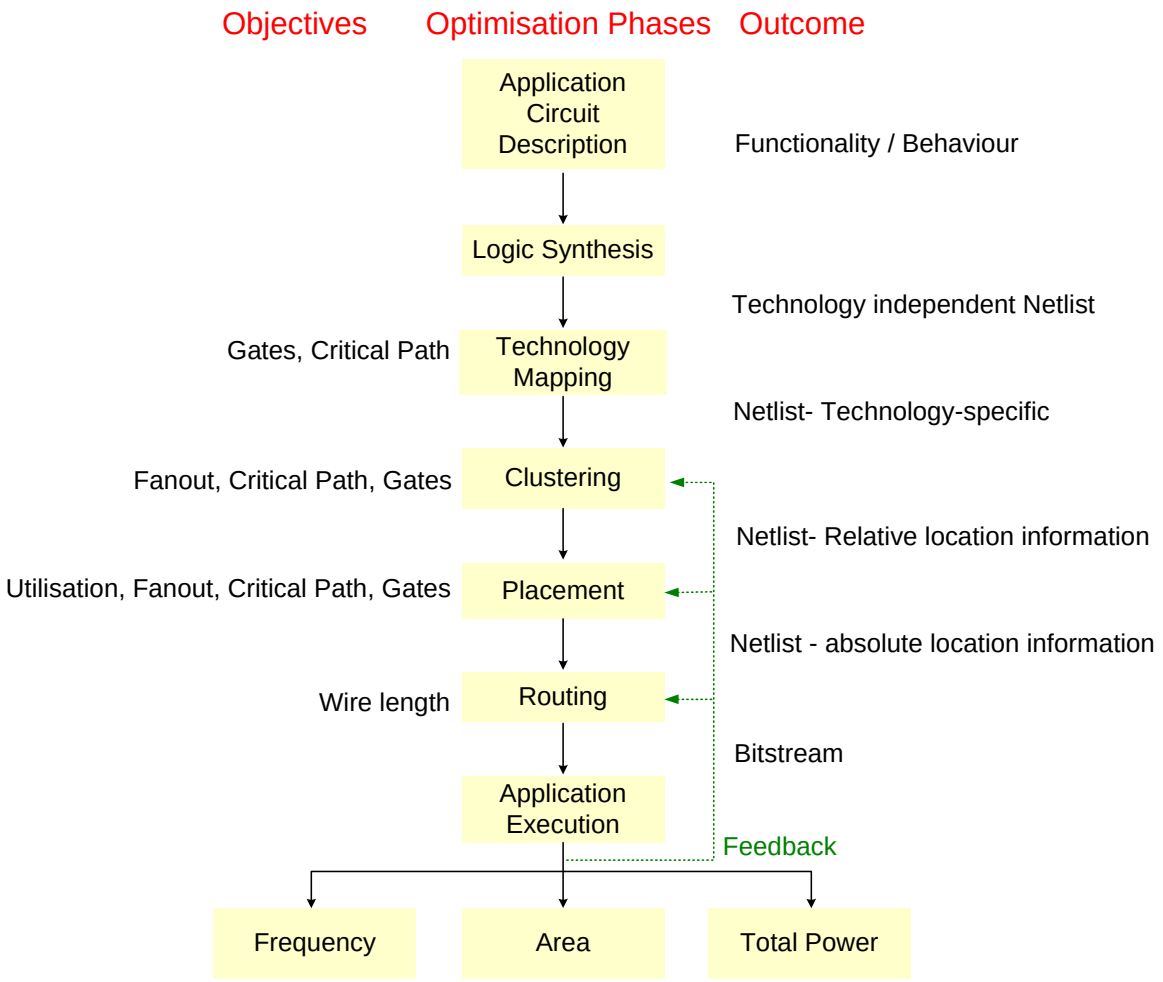


Figure 4.3: FPGA Design Flow

Feedback : Within the FPGA mapping flow, a feedback introduces a higher degree performance at each stage. Figure 4.3 shows stages that enable altering the design flow to suit the user-driven application objectives such as fanout in contrast to critical path or gate count. The performance objectives change with variations in application characteristics and hence the design objectives can be modified based on performance statistics obtained during each stage of application mapping.

Computational Complexity in FPGA Design Flow

The first stage of synthesis mainly involves technology-independent optimisation. The problem of technology mapping resembles the case of tree-pattern matching during instruction selection. Thus, it has the same level of complexity and addresses for multi-objective optimisation of delay, area, and power. Clustering is an extension of the technology-mapping problem, which also addresses the same multi-objective

performance parameters. Placement problem is an established *NP*-complete problem. Heuristics such as simulated annealing, genetic algorithms, etc. have been used for optimisation. Routing in two-dimensional FPGAs has also been recognised as an *NP*-complete problem. Heuristics, graph-based algorithms, and evolutionary iterative algorithms have been used for routing [81]. Overall, the computational complexity is a function of configurable space of the FPGA blocks, viz., LUTs, and the routing network.

4.2.3 Comparing the two Design Flows

Compared to the compilation flow, the FPGA (or standard cell) design flow involves a larger design space, higher number of design objectives, hence greater computational requirements, resulting in larger mapping times. The granularity of application mapping is distinctly different. FPGAs operate at user-defined granularity, which are realised using LUTs and combinatorial logic blocks, whereas processors operate at instruction-level granularity. Table 4.1 compares the design inputs, outputs and standard performance objectives for compilation techniques and FPGA/standard cell design tools.

Table 4.1: Comparing Compilation and Synthesis Design Flows

Design Flow	Entry	Granularity	Performance Objectives
Compilation	HLLs	Instruction	Cycles, Code size, Power
Synthesis	HDLs	Logical Operation	Frequency, Power, Area

In the compilation flow, time is an objective for optimisation in each of the stages. Program analysis techniques include control flow analysis (such as dead code elimination, loop optimisation, code motion etc.) and data-flow analysis (such as copy propagation, strength reduction, common sub-expression elimination etc.). Further, instruction selection, scheduling, and register allocation all incorporate strategies that directly make an impact on timing optimisation with respect to clock cycles required for execution. In the FPGA flow, during logic synthesis and technology mapping, timing optimisations are introduced via logic/register duplication. Timing and power-driven placement and routing approaches have also been integrated for overall performance optimisation.

Area optimisation using compilation techniques attempt to minimise the total code size, which comprise techniques such as retaining loops and eliminating dead-code. Code size reduction is an important optimisation phase in embedded processors, a survey of existing code-compression techniques can be found in [82]. During synthesis, techniques for area optimisation are primarily introduced to enhance resource

reusability. Area and timing are mutually opposing performance objectives. The area versus speed trade-off corresponds to a case of spatial versus temporal computing. For reconfigurable architectures, the same parameters help steer application-driven reconfiguration.

During FPGA synthesis, power is introduced as an application mapping objectives using methods such as, switching activity feedback, clock-gating, operand isolation, and fanout reduction. Additionally, power-driven placement and routing algorithms aim at minimising the switching activity and the interconnect length, which influences the interconnect capacitance, and consequently the power consumed. Power-driven compilation in [83] aims at reducing the hamming distance between consecutive instructions (and hence switching activity) by compiler-driven register name adjustment and dead register re-assignment. This approach has an overall power saving in the range of 25 to 44% of the core power. Instruction scheduling described, e.g., in [84], [85] aims at reducing the switching activity based on the instructions selected and the operands used. The focus is to vary the schedule of instructions selected to influence the inter-instruction or the circuit-state effect.

Table 4.2 summarises the differences in the compilation and synthesis design flows, with time, area, and power as the mapping objectives.

Table 4.2: Comparing Processor Compilation and FPGA Synthesis Objectives

Design Flow	Time	Area	Power
Compilation	Clock Cycles	Code Size	Switching activity
Synthesis	Critical Path	Gates	Fanout, Wire length, Switching activity

Considering strategies that can be applied to devices in field, or post-fabrication, the design space is limited to the application mapping in processors and FPGAs. Co-relating the two, the first stage can be termed as the stage of transforming application code to a target-specific netlist or architecture-specific instructions, as the case may be. The next stages involve spatial or temporal scheduling to meet performance requirements. The process of instruction selection in compilation and technology mapping during synthesis are both based on tree-covering or graph-covering algorithms. The stages following this translation diversify as time multiplexing during compilation or space scheduling in synthesis. As can be seen, application mapping in multiprocessors can be considered as a solution that merges both these types of multiplexing. Scheduling in space is a choice of the number of processors and multiplexing in time exists for the resources within each of the processors. Hence, scheduling in this context may be performed for varying number of resources, where the number of processors corresponds to the resource specification. Similarly, if the placement and routing algorithms in FPGAs were to have fixed delays, and if the number of resources was

limited to a fixed subset, the mapping procedure in FPGAs would resemble that of multiprocessors.

4.2.4 Merging Compilation and Synthesis Design Flows

Predominantly at the architectural level, FPGAs have been closely coupled with processors. In the CPU-co-processor approach, FPGAs have been introduced for application-specific acceleration. With closer coupling, FPGA-like reconfigurable fabric has been integrated within the datapath in the Stretch architecture [86]. In these examples, the processor is still a sequential processing element and the reconfigurable fabric aids in extending the processor for design acceleration.

The granularities in recent FPGA architectures have advanced from homogeneous, fine-grained LUT-only fabric to include integrated DSP blocks, multipliers, and even hard processor cores. Nevertheless, the task of partitioning the application to processors and FPGA fabric is entirely user-driven. In contrast, the QuadroCore design flow ensures a unified compilation flow since the target architecture comprises only processors. Adaptability in the parallel domain revolves around the granularity of parallelism composed of computation, communication, and synchronisation. Our application mapping approach in QuadroCore is to use the parallelism at processor-level granularity. Tasks that can be executed in parallel are mapped onto individual processors, where each processor still adopts a sequential processing model. To summarise, FPGAs comprise programmable computational blocks and programmable interconnects, to explore parallelism in the space-domain. Whereas processors have fixed computational blocks and fixed interconnects, with programmability permitted only in the time domain. To merge the advantages of both the diversities, QuadroCore uses fixed computational blocks with programmable interconnects to permit programmability in the time and space domains.

Figure 4.4 summarises the design flow used in QuadroCore. The design flow uses the philosophy of parallel mapping in FPGA synthesis as the first stage of application mapping and second stage of the compilation flow in processors. Consequently, the design flow is an inter-play of processor compilation and the FPGA synthesis.

4.2.5 Considerations for Merging Spatial and Temporal Design Flows

The presence of multiple processors enables exploring spatial parallelism. Additionally, the identical nature of each of the processors enables temporal resource sharing. Thus, spatial and temporal paradigms exist simultaneously in multi-core architectures, which

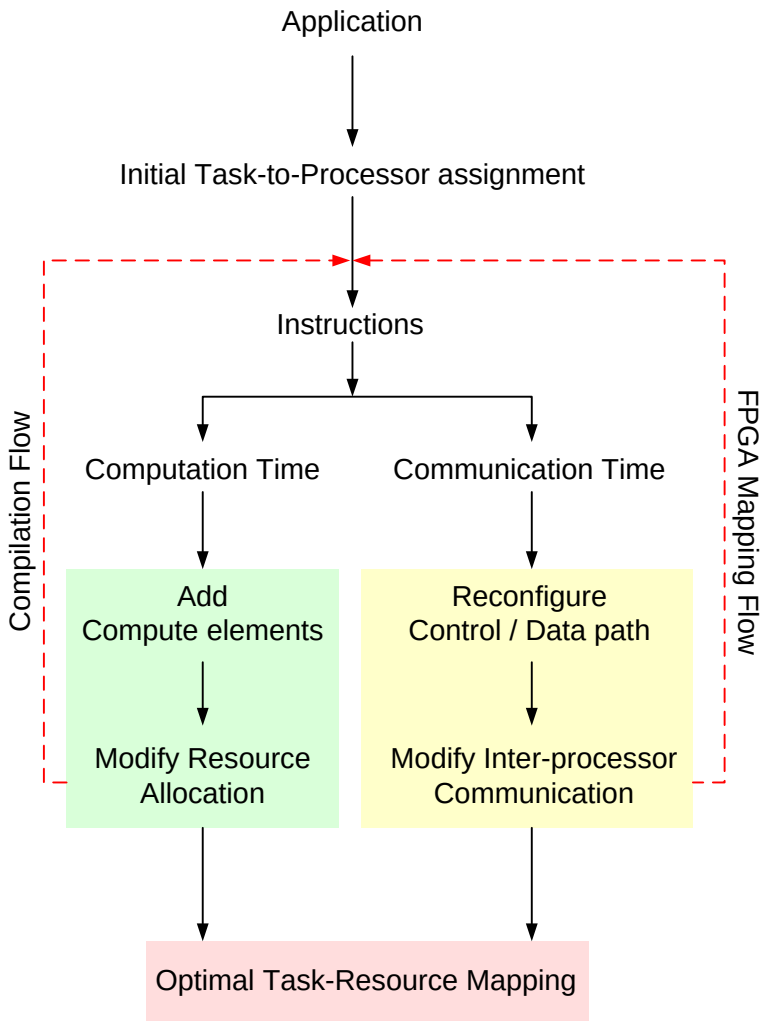


Figure 4.4: Merging Compilation and Synthesis

need to be selectively introduced to achieve resource efficiency and optimal application-to-architectural mapping. This computing model represents the case of communicating sequential processes [87] (CSP). The independent processes are executed in parallel, with inter-process communication, and each of the individual processes execute sequentially. Task distribution between processors enables spatial mapping and time scheduling the individually mapped processes ensure time multiplexing and resource sharing. Additionally, reconfiguration of resource connectivity enables adapting the resource availability on a per-task basis. Thus, the process of application mapping in this case is a multi-dimensional optimisation problem, with time and space as the contrasting parameters. This scheme of application mapping addresses the following mutually opposing design considerations:

- Priority of space over time or vice-versa viz., task scheduling versus task distribution.

- Overhead involved in reusing the same resource (time) versus introducing additional resources (space)
- The choice of static or dynamic reconfiguration, which consequently determines the overhead per reconfiguration and frequency of reconfiguration

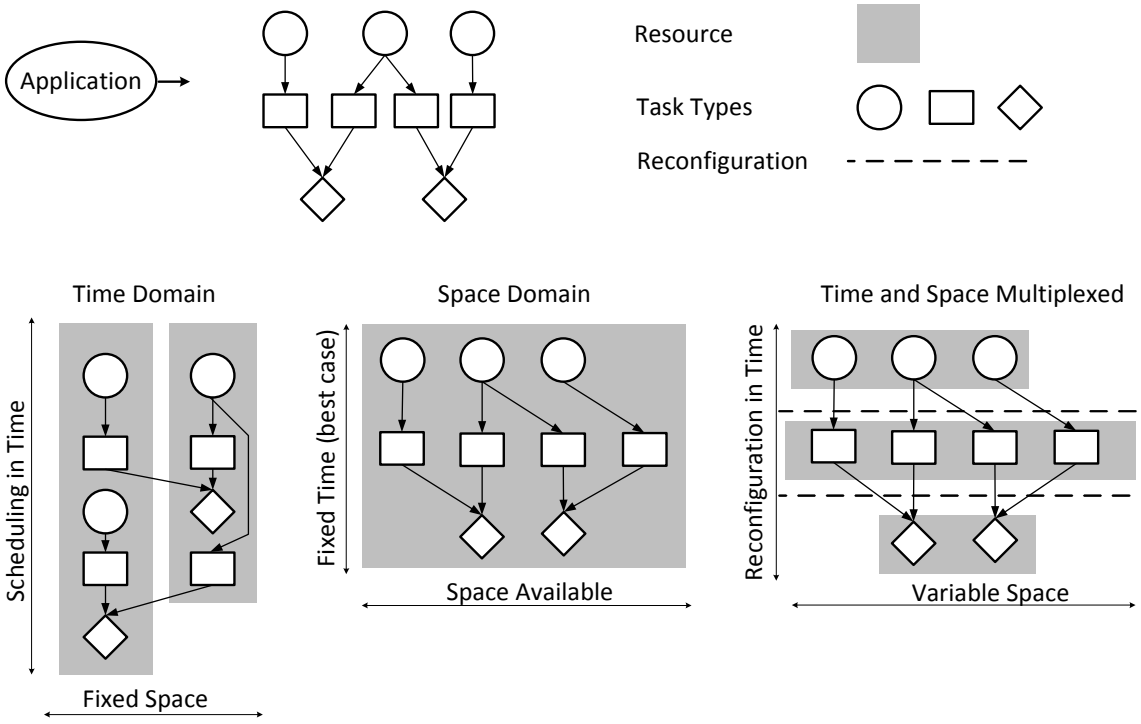


Figure 4.5: Spatial versus Temporal Scheduling

The reconfiguration mechanism necessitates both time and space scheduling, as shown in Figure 4.5. First, it ensures efficient resource utilisation by permitting resource sharing between processors. Secondly, in enables power savings by turning-off unused resources and hence saving both static and dynamic power. For such a concept to be effective, space and time scheduling have been developed. Firstly, space scheduling (or task distribution) is given a higher priority to meet resource requirement in terms of computation, which corresponds to resource allocation. Additionally, meeting the functional requirement of each of the processors is determined in this phase. Next is the stage where scheduling in time ensures that the task dependencies are satisfied. The resources allocated in space are time multiplexed to meet the application's functionality.

4.2.6 Optimisation Objectives

The process of application mapping is steered by multiple performance objectives to optimise resource efficiency. The two stages of space scheduling and time schedul-

ing address a diverse set of optimisation objectives. The performance objectives are expressed as:

$$Power = f(Register\ accesses,\ Memory\ Accesses,\ ALU\ accesses) \quad (4.1)$$
$$Time = f(Register\ cycles,\ Memory\ cycles,\ ALU\ cycles) \quad (4.2)$$

Overall, system optimisation is a combined function expressed as follows:

$$energy = f(time,\ power) \quad (4.3)$$

4.2.7 Cost Function

A domain-independent cost function is defined such that it is independent of the processor's instruction set architecture, pipeline stages, operating frequency, and cache hierarchy. It relies mainly on expressing the trade-offs observed in space and time. Thus, it enables portability for application mapping in multiprocessor architectures. Further, computational complexity and compilation time varies directly as the design space. In the formulations for the cost function, the design space is limited to establishing a trade-off between space and time, for a set of existing resource configurations. As detailed in Section 3.3 the total time and power consumption for a given application is given by:

$$\begin{aligned} T_{total} &= \sum_{i=1}^{k}(T_{comp_i} + T_{comm_i} + T_{sync_i}) \\ P_{total} &= \sum_{i=1}^{k}(P_{total-comp_i} + P_{total-comm_i} + P_{total-sync_i}) \end{aligned} \quad (4.4)$$

4.3 Adaptive Mapping in Reconfigurable Multiprocessors

As described in the previous sections, the performance of an application is optimal when the target architecture matches the application's computation, communication, and synchronisation requirements. This adaptability is achieved via reconfiguration in our multiprocessor. In order to enable flexible control and data flow between the processors in the QuadroCore multiprocessor, a set of reconfigurable capabilities have been introduced, as illustrated in Figure 4.6.

In order to ease data sharing between processors, the features introduced are classified as zones of reconfigurability, listed below. The following list enumerates the levels

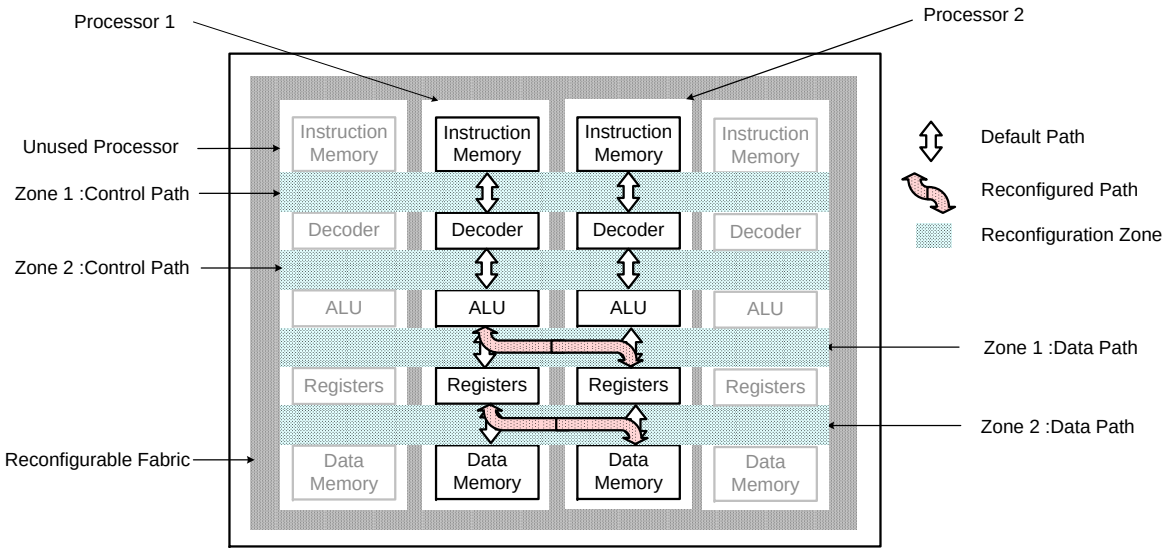


Figure 4.6: Zones of Reconfigurability

of hierarchy with the datapath of processors, which include the additional modifiable paths.

1. Local register file access (default)
2. *N*-hop neighbouring register access (Zone 1)
3. Local memory access (default)
4. *N*-hop neighbouring memory access (Zone 2)
5. External memory access (default)

In the reconfigurable multiprocessor, the first level of memory hierarchy is the register file access, as in the default case. The second level of access is made possible via the Zone 1 reconfigurable interconnect. The next stage is the local memory access, which is typically the second level of hierarchy in the absence of reconfiguration. The Zone 2 interconnect aids the access to the local memory of the neighbouring processor, and hence is the third level of memory hierarchy. Finally, the default access to external memory forms the last and the most expensive stage of data access.

4.3.1 Reconfiguration for Application Mapping

Conventionally, both in compilation and synthesis techniques, multiple optimisation passes are introduced to match the application's demands to a given static architecture. The static nature of the architecture is responsible for the performance deviation (defined in Chapter 2). Our approach in application mapping on QuadroCore relies on matching the architecture according to the diversity in application characteristics.

With changing application demands, the underlying multiprocessor architecture is reconfigured to match the computation and communication needs. The objective of application mapping is multi-dimensional. This phase relies on contrasting objectives such as − determining whether the tasks can be distributed in space (meet resource requirement) or need to be multiplexed (scheduled for time) or the focus is on energy minimisation (both power and time). Figure 4.7 shows two cases of application mapping that are made possible via reconfiguration within the multiprocessors, viz., static and dynamic mapping. The case of static mapping refers to processor customisation that are introduced on a per-application basis. Hence, this method incurs a one-time reconfiguration cost for every application. In the case of dynamic mapping, reconfiguration is used match the processor's resources for every task in an application, i.e., on a per-task basis. Hence, this case may require frequent reconfiguration to adapt to application with diverse characteristics. Figure 4.7 shows the control and data flow graph for a sample application with diverse characteristics (shown by changes in colours and shapes). In the case of static mapping, the processor configuration is determined for the application, and remains unchanged during the entire duration of application execution. In the case of dynamic mapping, the processor's configuration is continually modified with changes in application characteristics. A reconfiguration time is incurred for every change in configuration between tasks.

Static Mapping

In conventional compilation process, a match between the application-specific resource requirements is made with the available resources in the processor architecture using the information obtained during application profiling. In contrast to the conventional approach, flexibility between processor resources allows altering the resources available in the processors. The resource availability can be altered by including additional processors or borrowing resources from the neighbouring processors. Access to the additional resources is enabled via special reconfigurable interconnects among the processors in a multiprocessor organisation. A static mapping configures processors based on the application's computational demands before execution of an application. Since the processors are homogeneous and share the same base instruction set architecture, tasks are interchangeably moved or migrated onto any of the processors. Depending on the availability of processors and the achieved load distribution among processors, tasks are mapped statically onto processors. The case of static mapping configures the processors only once per application. Hence, once configured the processors retain their resource characteristics during the entire duration of the application execution. This case of static mapping resembles the case design space exploration in configurable processors like Xtensa [11], but introduced during run-time. The advantage of this approach is in the distinct division between the application execution time and

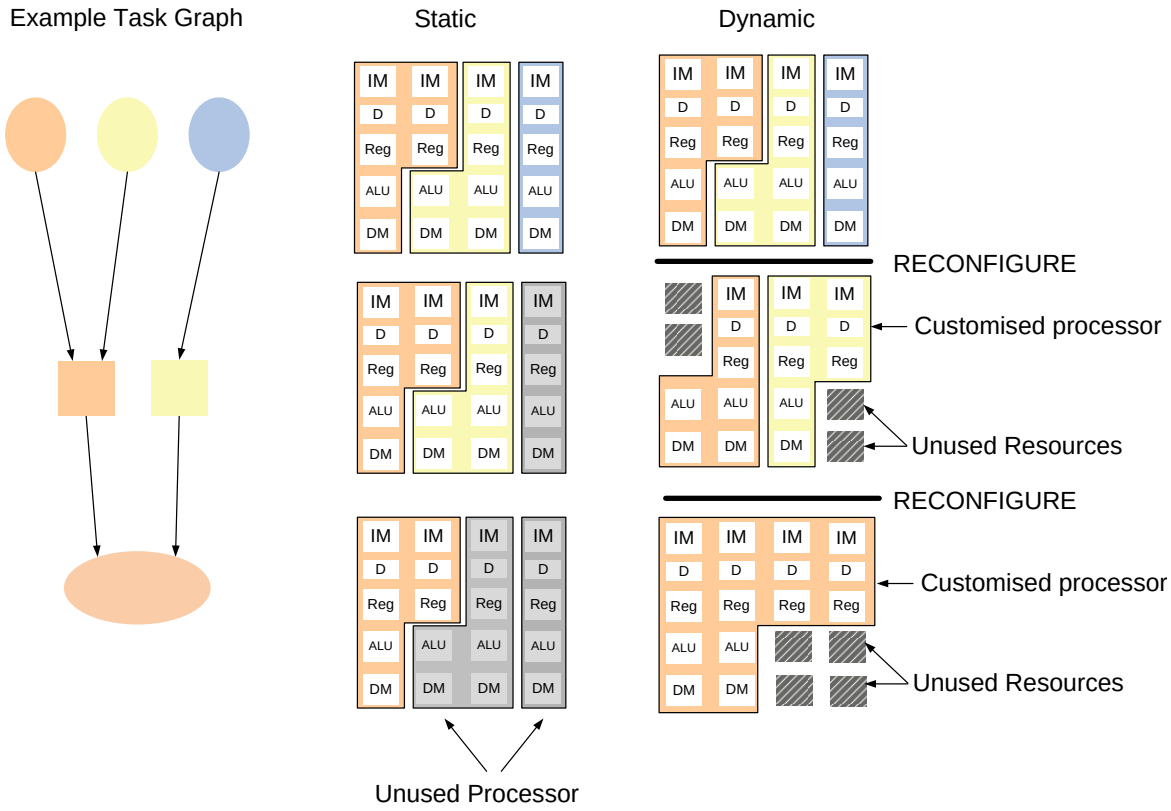


Figure 4.7: Application-driven Static and Dynamic Mapping

the reconfiguration time (once per application). However, as shown in the figure, an entire processor may be rendered unused even when resources may be required by a task, on account of the static mapping strategy.

Using the cost functions defined in Section 4.2.7, an algorithm has been formulated to steer application mapping using the static method. Reconfiguration in this context provides additional design space to optimise application-to-architecture demand. Thus, for every application that needs to be mapped, it may be characterised as the additional resources that can be accessed via reconfiguration, only limited by resource availability. To ensure optimal application to architecture mapping, a time budget (T_{budget}), power budget (P_{budget}) and energy budget (E_{budget}) is predefined, following which processor configuration is determined. The case of static application mapping is listed below in Algorithm 4.1. The algorithm takes into account the synchronisation overhead (T_{sync}), which is a trade-off between the amount of communication (T_{comm}) and the granularity of parallelism (T_{comp}). As described in the algorithm, it is an iterative procedure, where once the timing budget is met; the mapping is altered to meet the power budget. It has to be noted that in the case of static mapping, the reconfiguration time (T_{recon}) is not a part of the total execution time (T_{total}) or the total power consumption (P_{total}). Hence, static mapping is appropriate as a mechanism for

adapting the reconfigurable template to diverse applications, where reconfiguration is infrequent and is introduced only between application boundaries.

Algorithm 4.1 Static Task to Processor Mapping

Require: Number of Tasks $\leqslant$ Number of Processors
Ensure: $E_{total} \leqslant E_{budget}$ and $T_{total} \leqslant T_{budget}$

```
1: Assign one base processor for each task
2: while E_budget < E_total do
3:    while T_budget < T_total do
4:       for Each Processor, once per application do
5:          Profile Time
6:          if T_comp > T_comm then
7:             reconfigure to add ALUs
8:          else if T_comm > T_comp then
9:             reconfigure to add registers
10:         else if T_sync > T_comm then
11:            reconfigure change synchronisation mode
12:         else if P_total > P_budget then
13:            reconfigure to reduce ALUs, registers, synchronisation
14:         end if
15:      end for
16:   end while
17: end while
```

Impact of Static Mapping : The case of static mapping is advantageous for applications where the variations in the task-level characteristics within an application is minimal. Thus, a processor configuration can be chosen and retained during the entire duration of application execution, without having to reconfigure the processor. Configuring the processor to a particular application domain is an advantage for static mapping. Further, the processor conforms to the application-specific resource requirement more closely than a general-purpose processor, which typically has a fixed configuration. This results in a lower *performance deviation* and an increased optimality in terms of time, area, and power consumption in comparison to a fixed processor.

Dynamic Mapping

In the static case, individual processor building blocks may be rendered unused, depending on the diversity within an application. Hence, instead of fixing the resource definition for each of the processors statically, unused resources such as ALUs and registers are borrowed from the neighbouring processors dynamically during run-time.

Thus, a single processor is enhanced with additional resources to suit the diversity within an application. Each time, based on resource requirement, resources are acquired or relinquished via run-time reconfiguration. Each time there is a task that has to be mapped, the processors are reconfigured to match the characteristics. In contrast to the case of static mapping, the dynamic mapping encounters frequent reconfigurations within an application. This overhead can only be estimated on a per-application basis, since the characteristics and its diversity are application dependent. However, the conformity between application and architecture characteristics minimises *performance deviation* and ensures optimal resource utilisation. As shown in the figure, parts of processors may remain unused to ensure optimal resource utilisation.

The procedure adapted for dynamic mapping is listed below in Algorithm 4.2. In the dynamic case, reconfiguration is introduced on a per task basis to ensure optimality. Hence an important consideration in this case is the reconfiguration overhead (T_{recon}), which is now included in the total execution time (T_{total}).

Algorithm 4.2 Dynamic Task to Processor Mapping

Require: Number of Tasks $\leqslant$ Number of Processors
Require: Total Number of Resources per Task $\leqslant$ Total Number of Resources
Ensure: $E_{total} \leqslant E_{budget}$ and $T_{total} \leqslant T_{budget}$ and $T_{recon} \ll T_{total}$

```
For the first task, assign one base processor for each task
while E_budget < E_total do
  while T_budget < T_total do
    for Each processor, for every task do
      Profile Time
      if T_comp > T_comm then
        reconfigure to add ALUs
      else if T_comm > T_comp then
        reconfigure to add registers
      else if T_sync > T_comm then
        reconfigure change synchronisation mode
      else if P_total > P_budget then
        reconfigure to reduce ALUs, registers, synchronisation
      end if
    end for
  end while
end while
```

Impact of Dynamic Mapping : Dynamic mapping enables adapting the architecture to the incoming task requirement. This adaptability is introduced on a per-task basis. In comparison to the static case, the architecture is further adapted to suit the application. Hence, if the impact of reconfiguration is not significant in comparison the

total computation time, it results in a increased optimality with respect to resource efficiency in comparison to the static case. The task-to-processor adaptability results in a lower *performance deviation* in comparison to the static case.

4.3.2 Advantages of the Multi-dimensional Mapping Approach

The new multi-dimensional mapping of applications addresses application targeted to reconfigurable multiprocessors, agnostic of the processor's instruction set architecture. The basic principle is to rate instructions with respect to time, power, and code size. Thus, diverse properties of instructions are user to steer energy characteristics during application mapping. The formulated cost function stems from the instruction set architecture of the processor being used, thus can be modified to other instruction set architectures.

The case of time and space scheduling is addressed as a two-stage process. The space-dimension is considered first, with the resource allocation that addresses the task-to-processor mapping. This stage is either static, with fixed resource assignment or dynamic with reconfigurable resource connectivity between resources. The second stage addresses the time-domain, where the instructions are scheduled sequentially to execute the given task. On account of fixed resources and programmable interconnects, the reduction in the design space makes the multi-dimensional approach faster for design space exploration, in comparison to fine-grained FPGAs. In addition, reconfiguration acts as a tool for fine-tuning application mapping. In the static case, reconfiguration introduces adaptability on a per-application basis. To adapt to diversities within applications, dynamic reconfiguration introduces run-time modifications to match application-to-architectural characteristics.

4.4 Summary

The objective of application-to-architectural mapping is to meet diverse functional and performance requirements. In this chapter, a classification is made that matches applications and architectures, based on the nature of architectural adaptability and the variations in application characteristics. In this context, two existing methods of application-to-architectural mapping techniques — compilation in processors and the synthesis in FPGA design flows, are analysed. They are representative of two diverse paradigms, viz., sequential and parallel programming models. In both the design flows, a feedback mechanism is identified as an essential requirement to ensure optimality in the mapping process. An automated early feedback mechanism is predominantly absent in present day tools and design flows.

Our approach is to merge to the two design flows and introduce a feedback-driven mechanism to steer application mapping. Additionally, the objective is to enhance flexibility and ease programmability when mapping applications with diverse characteristics to architectures during run-time. To this effect, an adaptive mapping strategy has been introduced via reconfiguration of the control and data path of the processors. Reconfiguration in this context enables adapting the architecture to meet application-specific requirements. These concepts merge spatial and temporal programming paradigms to explore diverse performance objectives such as time, power, area, and energy. Two strategies of application mapping — static and dynamic mapping are presented that address application mapping in our reconfigurable multiprocessors. The case of static mapping aims at application-specific processor customisation, with infrequent reconfigurations. In the case of dynamic mapping, reconfiguration is introduced frequently to adapt to changes within an application. Overall, mapping is initiated in conjunction with architectural alterations using the application-level statistics and reconfiguration as a method of run-time adaptability.

Chapter 5

QuadroCore: Architecture

To ascertain the feasibility of the architectural features presented in the previous chapters, QuadroCore is the prototype implementation of the architectural concepts of the reconfigurable multiprocessor template. The focus of this chapter is to present detailed quantitative results to help assess the benefits of the concepts presented in the previous chapters and propose directions that may lead to future research. QuadroCore is a four-processor realisation of the reconfigurable multiprocessor template introduced in Chapter 2. Our systematic design methodology builds upon the architectural variations and performance impact of application-specific characteristics, discussed in Chapter 3. Application mapping is steered with time and power as the performance objectives, as discussed in detail in Chapter 4.

This chapter presents the architectural details of the QuadroCore multiprocessor and performance measurements for its implementations. In Section 5.1, the reconfiguration capabilities that have been built into the architecture are presented. Also in this section, our unique concept of a fast, low overhead run-time reconfiguration has been detailed. The reconfiguration design space that can be explored as a combination of the QuadroCore multiprocessor architecture and the reconfiguration mechanism are analysed with time and power as the performance objectives. An instruction-level power model is presented that can be incorporated in the timing and power aware application-mapping design flow, as presented in Chapter 4. In Section 5.4, the impact of the standard compilation techniques employed to map an application with considerations to the mutually opposing time and power characteristics are analysed. Finally, a performance analysis for the QuadroCore multiprocessor when realised using standard cells and FPGAs, and the *performance deviation* are analysed.

5.1 Reconfiguration Design Space

QuadroCore is our reconfigurable multiprocessor architecture composed of four 32-bit RISC-based processors, called NCore described in [88]. The instruction set architecture of NCore is based on Motorola's MCore processor. The instruction set architecture of the NCore processor provides about 11% free opcode space to allow architectural enhancements. This free opcode space has been utilised to add instruction set extensions that permit run-time modifications to the architecture and support for co-operative operation of multiple instances of the same processor. Using a network-on-chip this collection of processors can be scaled to include multiple such processors. The QuadroCore has been enhanced to include all the features described in the reconfigurable multiprocessor template. Thus, it can switch between a set of predefined modes to adapt to application-specific requirements. The template in Figure 5.1 shows four loosely coupled processors in QuadroCore. The reconfigurable connectivity between the existing resources is added as an architectural feature.

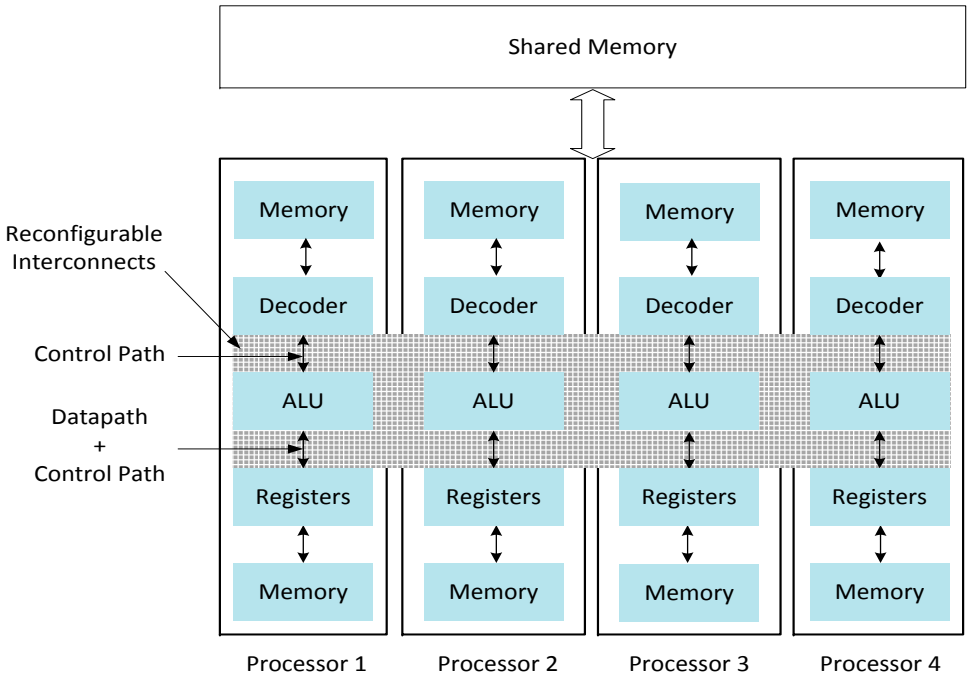


Figure 5.1: QuadroCore Reconfigurable Multiprocessor

Table 5.1 shows the possible reconfiguration options implemented our QuadroCore reconfigurable multiprocessor. As noticed from the column on the right, application characteristics define the mode of operation. These characteristics can be different within stages of an application or between applications in a single application domain. In order to adapt to these variations, this unique scheme or reconfiguration allows fast switching between the reconfigurable modes of operation during run-time. In

the following sections, each of these operating modes and the reasoning behind this approach is described. In [PP08a], a summary of all the modes that can co-exist in QuadroCore is depicted.

Table 5.1: Reconfigurable Operating Modes

Operating Mode	Application Characteristics
Asynchronous MIMD	coarse-grained, task-level parallelism
Synchronous MIMD	fine-grained, instruction-level parallelism
SIMD	data-level parallelism
Fast memory access	data-level parallelism, large amount of data exchange
Using shared register file	fine-grained, few, frequent register exchange
Wide-word ALU	data-level parallelism
Sharing registers	applications with high register pressure

5.1.1 Instruction to Control Reconfiguration

The decision of altering the existing structure is driven by the instruction executed. Hence, the choice of resources and their variations are determined during application compilation and requested during run-time. A quick, single cycle run-time reconfiguration ensures low overhead in terms of time required to reconfigure this resource connectivity. The execution of a special reconfiguration instruction connects the existing resources. Thus, depending upon the resource demand of the application these instructions are executed at boundaries between regions where a change in resource requirement is observed during program analysis. This reconfiguration instruction acts as the configuration information to determine the functionality of the reconfigurable interconnects between the intermediate stages of the instruction pipeline.

The interconnect introduced between the decode & execute stages, and execute & register read/write stages allows run-time selection and co-operative resource sharing between the multiple control paths (and/or datapaths) among the four processors. Figure 5.2 shows the location of the reconfigurable interconnect.

Reconfiguration Instruction Format

The QuadroCore reconfiguration information comprises the operating mode information. This is a single instruction, where the choice of the mode is decoded from the instruction. This approach is contrast to FPGA-based designs, where a configuration file includes both control and data information required to configure the individual configurable logic blocks, memories, and define their interconnections.

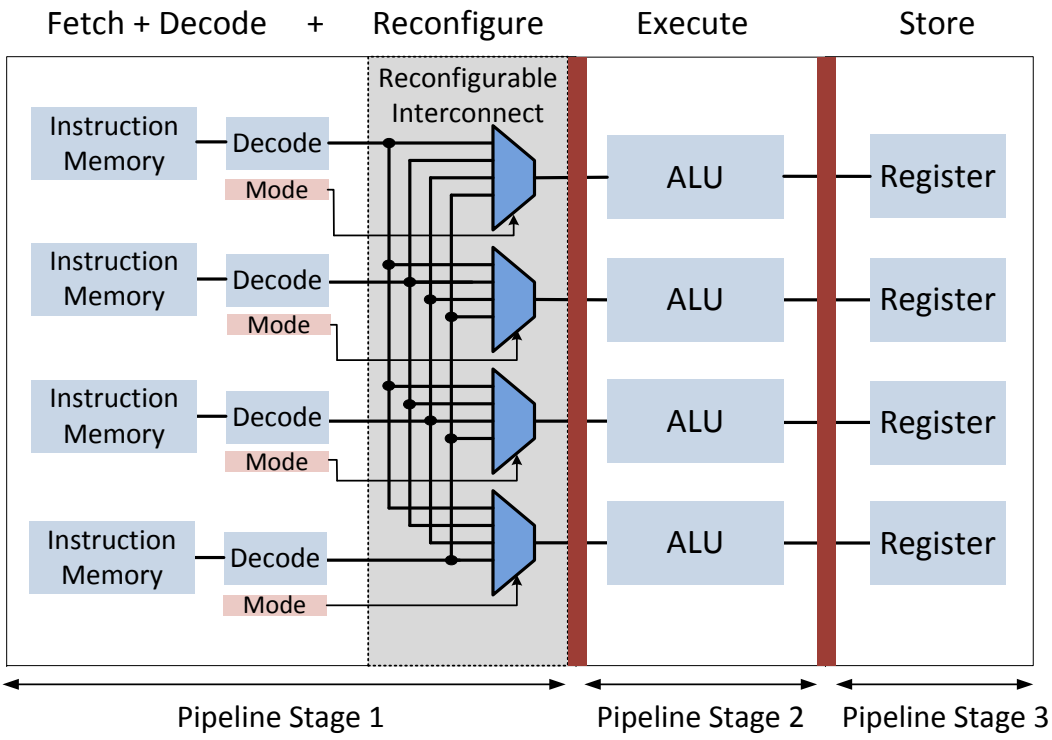


Figure 5.2: Reconfiguration Mechanism

Reconfiguration Stream

Since the configuration stream is a part of the instruction stream (shown in Figure 5.3), a separate configuration stream and configuration memory is not required. Reconfiguration information is part of the instruction stream and resides along with the program code.

Figure 5.3 illustrates mode changes encountered during program execution achieved via instruction streams in the QuadroCore multiprocessor. Initially, all the processors operate in the default asynchronous mode. Due to application demands, Processors 1-3 are required to be switched to the synchronous mode. In this case the mode switch is a two stage process; firstly Processors 1-3 are synchronised using barrier synchronisation. Next, a reconfiguration switch changes the operating mode to the synchronous mode. During this time, Processor 4 continues to operate in the default asynchronous mode. Next, all the four processors are reconfigured to operate in the SIMD mode. In all these cases, it has to be noted that the instruction stream itself behaves as the configuration stream to allow switching between the fixed set of modes. The mode changes are determined during compile-time and the architectural changes (reconfiguration) are inferred during run-time. The compilation flow and details of the compiler can be found in [PPR+].

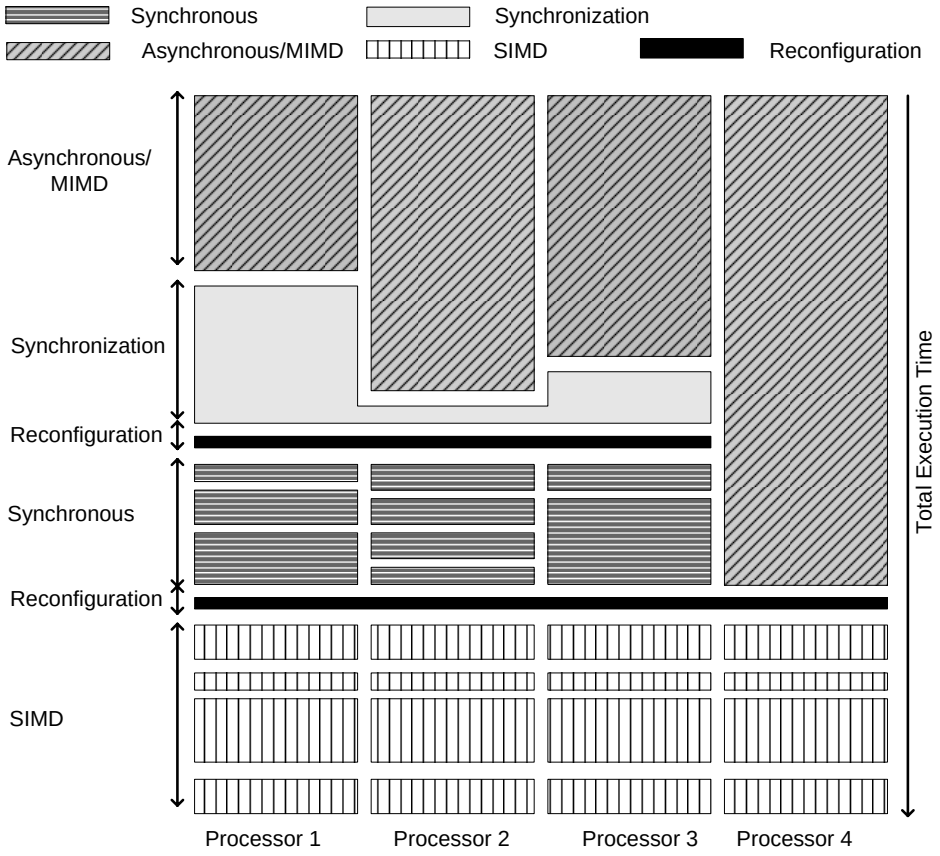


Figure 5.3: Instruction Stream as the Configuration Stream

5.1.2 Synchronisation

Synchronisation enables one or more processors to initiate communication at a common time instance. In multiprocessors, synchronisation is essential to co-ordinate processors to operate in unison. Applications mapped onto multiprocessors exhibit varying degrees of parallelism. A variable granularity of parallelism demands suitable synchronisation schemes, such that the clock-cycle overhead of synchronisation is minimal. Here, two modes of synchronisation have been introduced. For infrequent data exchange or coarse-grained parallelism, synchronisation between processors is achieved via a single-cycle barrier synchronisation scheme. This mode is termed as asynchronous MIMD, since the processors operate independently until they encounter a barrier synchronisation instruction. As shown in Figure 5.4, in the asynchronous mode of operation, the definition and use of a variable (a in figure) requires insertion of barrier synchronisation in order to avoid a read before write violation. This ensures that the *use* operation is executed only after a $define$ has been executed on Processor 1.

For frequent or fine-grained synchronisation, the processors can be configured during run-time to operate in a lock-step fashion. In this mode, all the processors operate synchronously according to the schedule pre-determined by the compiler, which avoids explicit synchronisation. This mode is termed as synchronous MIMD and is suited for instruction-level parallelism. As shown in Figure 5.4, it resembles the default operating mode in VLIW processors, where all functional units operate synchronously. Hence, the read before write violations are managed during scheduling stage in the compilation process. However, a disadvantage of this mode of operation is the need for operating each instruction synchronously, irrespective of inter-processor data exchange. In VLIW processors, the presence of a single decoding unit ensures synchronous operation at all times. Here, the synchronous MIMD mode is a case of pseudo VLIW operation, as synchronisation at instruction boundaries is explicitly introduced by the hardware implementation. The hardware architecture ensures that a new instruction is fetched only when all the processors have completed executing the current instruction. In other works, all the participating processors perform an instruction fetch simultaneously. Another method of ensuring deterministic scheduling and execution is by forcing the instruction length for every instruction. In this way, all instructions have a predetermined execution length and there are no uncertainties in instruction execution time encountered during execution.

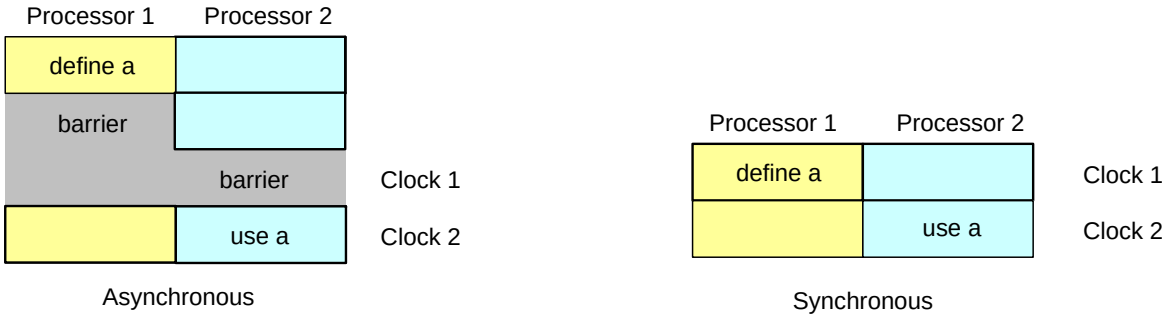


Figure 5.4: Types of Synchronisation

By default, the multiprocessors operate asynchronously and exchange of data is made possible via explicit barrier synchronisation, which has the following construct, where *mask* is a 4-bit values that represents the processors (all or a subset) that need to be explicitly synchronised.

```
barrier(mask);   /* Initiates Barrier Synchronisation */
```

The execution time for this instruction is one clock cycle and is dependent on the arrival time of all the processors required to synchronise using this instruction.

The following instruction switches processors listed by *mask* to a synchronous mode of operation, where *mask* represents the processors (all or a subset) that operate synchronously.

```
synchronise(mask); /* Synchronises Processors defined in mask */
```

In this mode, the execution times for all the instructions need to be fixed during compilation.

Hardware Support for Synchronisation

In the asynchronous mode of operation, barrier instructions synchronise between independently operating instruction streams. Since the task of barrier placement is optimised during compilation, the hardware architecture has to ensure a very low cost instruction execution time without affecting the system's operating frequency. In our architecture, this synchronisation is achieved in a single clock cycle, where each processor accesses a barrier status register asynchronously. Depending on when each of the processor encounters a barrier instruction, the barrier status register is set accordingly. When the required subset of barriers has been reached, the register is reset and the status is provided simultaneously to all the processors. Hence, constant polling of an external memory address, employed in classical synchronisation methods is avoided. The single cycle restriction introduces a minimal variation in the system's operating frequency, discussed later in Section 5.5.1. This method of synchronisation is faster than techniques implemented via software barriers in recent implementations such as in [89].

In the synchronous mode, the instruction streams operate in lock-step, synchronous fashion at every instruction. This ensures a predictable behaviour to allow the compiler to schedule the instruction to explore the maximum degree of instruction-level parallelism. The instructions are restricted to fixed cycles per instruction, explicitly fixing the execution time for all instructions. For example, instructions with data-dependent execution lengths, such as early exits in multiplications, are disabled. Although the execution time of each instruction is forced to a worst-case value, there is no additional delay involved in synchronising between instruction streams. Here, the maximum operating frequency of the system remains unaltered, but the clock cycles required for execution changes.

5.1.3 Communication

Originally, exchange of register contents between processors was solely permitted via a shared external memory. This involves a significant clock cycle overhead, since access to the external shared memory is managed by a round robin arbitration mechanism. Using this scheme each processor has to request for access and depending on the number of simultaneous requests and time of arrival of each request, the access to the

external shared memory is assigned. To avoid this large access time and to enable quick exchange of register values between processors a multi-port register file consisting of 32 registers was introduced. This multi-port shared register file is accessible to all the processors simultaneously. This allows sharing of register values without having to alter the instruction set architecture.

In addition to dedicated access to a shared register file, fast access to external shared memory was also incorporated. Since all the processors access a common external memory via a shared bus, a bus contention is inevitable. To circumvent this bottleneck, fast-memory-access mode was added. In this mode a single processor takes over the external memory access mechanism for all the four processors. It fetches data for all the processors in a single wide-word fetch (128-bit). This fetched data is internally distributed among the processing elements (32-bits). This mode of operation bypasses the bus arbitration and contention incurred during shared data access. Additionally, it also introduces determinism in the memory access mechanism, since the waiting time in arbitration is entirely avoided.

Additionally, register sharing among processors is possible on account of the reconfigurable interconnect introduced between the ALUs and the register files. For applications with high register pressure, registers from the neighbouring processors are borrowed. Figure 5.5 shows the reconfigured register write (control and datapath), which helps to avoid the register read operation for the next operation and minimises the data-exchange overhead between the two processors.

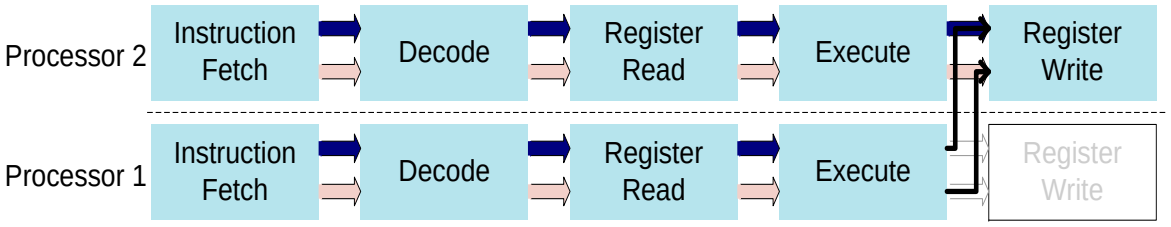


Figure 5.5: Mechanism for Sharing Registers Contents

Table 5.2 shows the *share* construct that moves the result computed in the execute stage of Processor 1 to the local register file of Processor 2 (X[2]) by reconfiguring the register write stage of Processor 1 to be directed to the local register file of Processor 2. This mode saves data transfer time (via shared register file or shared memories), hence resulting in a reduction in the number of instructions and execution time. This example assumes that the Register 2 in Processor 2 is available for use by Processor 1 during the transfer. For this operation both the control path and the datapath between the processor's execute and register write stages are reconfigured. Communication can also be chosen a reconfigurable operating mode depending on the amount and frequency of data communication between processors.

Table 5.2: Inter-processor Communication

With Reconfiguration	Without Reconfiguration
`share(X[2],2);`	`ld(X[2],M1))`
	`st(M1,2)`
or	
`send(X[2],2);`	
`receive(X[2],2);`	
3 or 4 clock cycles	10-24 clock cycles

This construct by itself translates into a reconfiguration instruction and a register write operation, which totally requires 3 clock cycles. In the absence of this mode, the total time required to transfer one single register content between two processors requires 4 clock cycles using the shared register file (send/receive operations). Using the shared memory, the duration ranges between 5 to 12 clock cycles for each operation. In the above example, the total round time results in 10 to 24 clock cycles when using external memory.

Shared Register File for Data Communication

A shared register file has been introduced to ease the data exchange mechanism between the processors. This shared register file consists of 32 registers, accessible by all the processors via dedicated ports at all times. This set of registers is in addition to the 16-entry local register file that exists for each processor. Since there are independent read and write ports for each processor, no hardware arbitration mechanism is required for registers access, since the valid read-write sequences are scheduled during compilation. This ensures a two clock cycle access time for read or writes operations, enabled via special load and store instructions. As access to the external memory for data exchange takes 6 to 15 clock cycles, it is not used for communication. Hence, the round-trip time (write and read) is 4 clock-cycles for the shared register file in comparison to 16 to 30 clock cycles using the shared memory. Further, the compiler manages data dependencies and read-write sequencing. A similar mechanism is added via instruction set extensions to allow sharing (or broadcasting) the condition flag of one of the processors for collective branch operations. The shared register file is only used for inter-processor communication, because its access time is longer than accessing the local registers of a processor. In order to utilise the shared registers for all instructions the encoding of register operands would have to be extended to store the

additional register numbers. This would result in larger instructions, and hence an increase in code size.

5.1.4 MIMD and SIMD operation

The MIMD mode of operation is the default mode of operation in the QuadroCore multiprocessor. The SIMD mode of operation is used when all the processors execute the same instruction stream. For this mode of operation, one of the decoder is responsible for forwarding the control and data signals to all the four ALUs, register files, and local data memories. Thus, in this mode of operation a single instruction is sufficient for operation of all the four processors. One of the decoder switches to a master mode and forwards the decoded instructions to all the participating processors. The unused decoding units of all the other processors are switched to an idle mode, for energy savings. Thus, instruction fetch and decode operation is performed by one of the processors (called 'master'). Consequently the instruction fetch and decode unit of the other processors are switched-off for energy savings.

Additionally, for faster data access, the fast memory access mechanism is used in conjunction with this mode (described in the next subsection). Depending on the application, one (or more than one) processor(s) could operate in the 'master' mode. A special reconfiguration instruction enables switching between the default MIMD mode (where each processor operates on its own instruction memory) to the SIMD mode. This instruction executed during run-time provides reconfigurability in hardware. Thus, both the modes of operation co-exist and are invoked based on application demands.

Figure 5.6 shows the operation for Processor 1 and Processor 2 in SIMD mode, where the control path between the decode and execute stages are reconfigured.

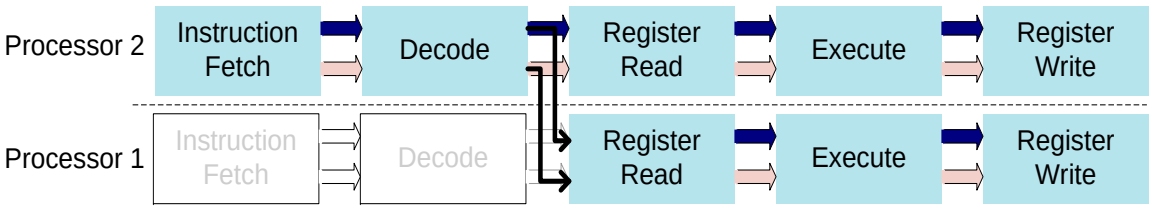


Figure 5.6: Single Instruction Stream, Multiple Data Stream

The MIMD mode is the default mode of operation and a SIMD mode of operation can be activated via a reconfiguration instruction. Following is an example of loops, as shown below is a *for* loop:

```
for-simd(i=0; i<4; i++)    /* switches to SIMD mode */
{SIMD operations}
```

where, *for-simd* results in instruction fetch and decode managed by one of the processors. Thus making a corresponding reduction in code size, instruction-fetch & decode. Additionally, the fast memory access mechanism circumvents the communication delay in accessing the shared memory for all the participating processors by simultaneously fetching multiple memory locations followed by internal re-distribution.

Fast Memory Access

The external shared memory is used for sharing data streams and large amounts of common data. This memory is accessible by all the processors via a round robin arbitration mechanism. When multiple processors access this external memory, the round robin arbitration mechanism provides sequential access. This architecture is well suited for random, asynchronous accesses initiated by the processors sharing memory. However, a substantial bottleneck is introduced during simultaneous access to memory, which is incurred in the SIMD mode. To circumvent this bottleneck, a fast memory access mechanism is added to accelerate access to adjacent memory locations using additional instructions. Using this instruction, a single transaction accesses multiple adjacent memory locations, which may represent consecutive locations of an array. Thus, the multiple data locations read are distributed (or collected) internally among the four processors. A similar procedure is also applicable for storing data arriving from the four individual processors via a single write operation to external memory. These special instructions avoid the delay involved during arbitration and reduce the total access time from a worst case of 15 clock cycles to exactly 7 clock cycles.

The following code in Table 5.3 shows the method of invoking the fast memory access mode during application programming. The *copy-simd* construct copies the variable X to all the participating processors, followed by an addition operation performed simultaneously, with only the master processor performing the instruction fetch and decode. This mode of operation is used to speedup data distribution, as in the case of the SIMD mode.

5.1.5 Word-length Configurability

For a given architecture, the effective resource utilisation depends on the effectiveness of the application mapping mechanism. A variation in the valid word-length required in the application directly influences resource utilisation and consequently the power

Table 5.3: SIMD core with Fast Memory Access

With Reconfiguration	Without Reconfiguration

`copy-simd(X[i]);`
`copy-simd(Y[i]);`
`for-simd(i=0; i<4; i++)`
`{`
`Z[i] = X[i] + Y[i];`
`}`
	`for(i=0; i<4; i++)`
	`{`
	`copy(X[i]);`
	`copy(Y[i]);`
	`Z[i] = X[i] + Y[i];`
	`}`

17 clock cycles	24 clock cycles

dissipation. Here, a variable word-length property of the processor allows using only the required word length as required by the application. In addition, multiple 32-bit ALUs from the neighbouring processors are merged to expand the word-length of a processor.

Figure 5.7 shows the representation, where two 32-bit datapaths are merged to form a single 64-bit datapath, also avoiding redundant instruction fetch and decode operations.

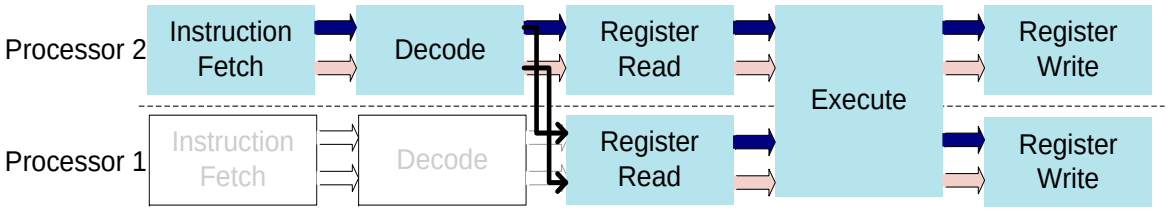


Figure 5.7: Varying the ALU Word-length

```
Y = add64(X[0], X[1]); /* 64-bit addition */
```

Two ALUs are merged by using the instruction *add64*, which ensures that the addition operations includes the carry over from the lower order bytes to the higher order ALU in the neighbouring processor. Reconfiguring the adders permits carry-over logic, and saves instruction fetches for word-width operations, provided all the participating ALUs execute the same instruction. The same is applicable to the other arithmetic units and ALU operations, viz. subtraction, division, multiplication, etc. The difference in the clock cycles required for execution with and without this reconfigurable mode is shown in Table 5.4.

Table 5.4: Variable Word-length ALUs

With Reconfiguration	Without Reconfiguration
`Y = add64(X1[63:0], X2[63:0]);`	`Y1 = add(X1[31:0],X2[31:0])` `Y2 = add(X1[63:32],X2[63:32],C0)`
2 clock cycles	4 clock cycles

To summarise, Figure 5.8 shows the modes of operation that have been introduced via reconfiguration in the QuadroCore multiprocessor. These modes of operation enable customisations according to application requirements.

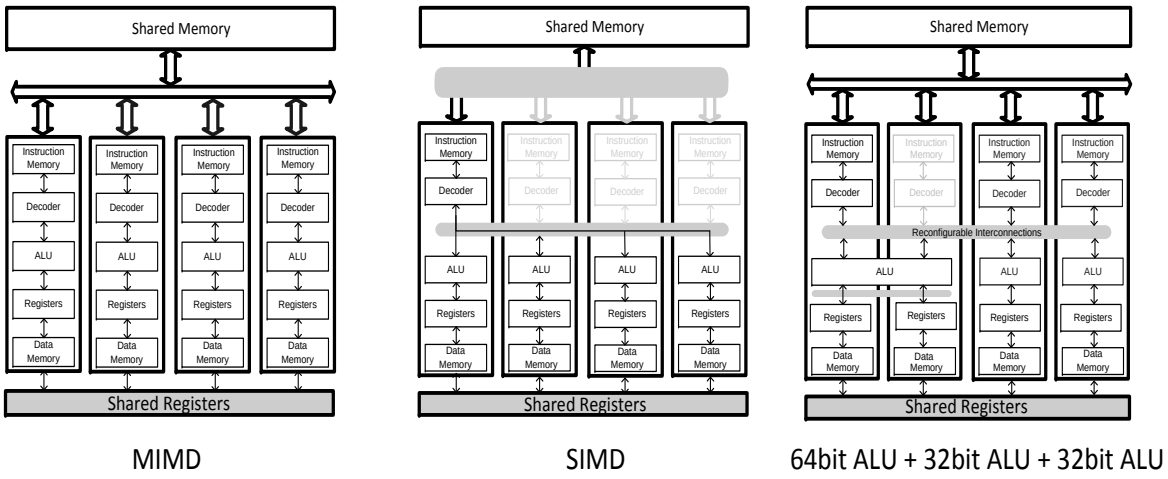


Figure 5.8: Reconfigurable Modes in QuadroCore

5.1.6 Additional Instructions for Co-operative Multiprocessing

In order to enable the above mentioned operating modes and additional co-operative processing among the four processors, a set of instruction set extensions were added to ease the control flow. Table 5.5 lists the instructions and their respective functionalities. As an example, a combination of `cstw` and `cldw` enabled data exchange among participating processors. Similarly, executing `crsync` on a master processor results in broadcasting the carry flag among the four processors.

5.1.7 Compilation Flow

To aid automatic application parallelisation onto QuadroCore, a compiler has been designed in parallel to this work on hardware architecture. The approach, called Co-

Table 5.5: Instruction Set Extensions for Co-operative Processing

Instruction	Function
`cstw`	Store contents of local register to shared register
`cldw`	Load contents of local register from shared register
`barrier`	Initiates barrier synchronisation among all (or a subset) of processors
`reconfig`	Reconfigures between modes
`mbt`	Modified branch flag condition: Branch if true
`mbf`	Modified branch flag condition: Branch if false
`mjmpi`	Modified branch flag condition: Branch if false
`mjsri`	Modified immediate jump index
`mlrw`	Modified relative word
`crsync`	Share carry flag
`load128`	Loads 128-bit data from shared memory to SIMD register
`store128`	Stores 128-bit data to shared memory from SIMD register
`writesreg`	Writes from SIMD register to local register file
`readsreg`	Reads from local register to SIMD register

BRA [1], uses compile-time analysis determines the schedule for reconfiguration during run-time. Reconfiguration alters the architecture by choosing between a fixed set of operating modes, called *reconfigurable architectural variants* at run-time. Given a program that exhibits both regular and non-regular structures, the compiler determines the best execution mode by analysing the parallelism during compilation. Further, the usage of a manageable set of variants leads to an enormous reduction in the design space, compared to fine-grained reconfiguration. The compiler then addresses this finite design space efficiently by using well-known program analysis techniques [90]. This is in contrast to research using fine-grained reconfigurable architectures, where reconfiguration typically incurs a significant overhead [91]. More details of the compiler construction and strategies used can be found in [92].

5.2 Time and Power Characteristics

Timing and power considerations have been incorporated as performance objectives in the design of the QuadroCore multiprocessor. The following sections provide a detailed analysis of these two performance components, which together contribute to the processor's energy-efficiency.

[1] Compiler Driven Dynamic Reconfiguration of Architectural Variants (merge two Ds to a B)

5.2.1 Timing Characteristics

Application partitioning distributes instructions, tasks, or data among the four processors in the QuadroCore multiprocessor. Application partitioning at instruction-level granularity is possible using the CoBRA compiler. Methods of divide and conquer algorithms aid in task and data partitioning. In this context, an application with execution time T_{total} that is partitioned onto the four processors is expressed as follows:

$$T_{QuadroCore} = \frac{T_{total}}{4} + T_{sync} + T_{comm} \tag{5.1}$$

where, $T_{QuadroCore}$ is the execution time on QuadroCore, 4 is the number of processors in QuadroCore, T_{sync}, T_{comm} correspond the inter-processor synchronisation and communication times respectively, as defined in Section 3.3. Here, each of the timing components can be steered using the reconfigurable operating modes in QuadroCore.

QuadroCore Access Times

The base RISC processor, NCore has a single cycle execution time for most ALU operations and access to the local register file. Access to the local memory for data and instruction access involves a 3 clock cycle delay. Next in the memory hierarchy, the access to the shared external memory involves an access time that ranges between 6 to 15 clock cycles. The variation in the access time is on account of the delay involved in arbitration and the number of simultaneous requests, which requires queuing of the accesses. This is the default access time in the QuadroCore multiprocessor.

The additional reconfigurable operating modes introduce modifications to the access times. The shared register file has an access time of 2 clock cycles per operation and 4 clock cycle round time for a read-write operation. The fast memory access mechanism reduces the 15 clock cycle delay in accessing the external shared memory to a 7 clock cycle access time in case of access to adjacent memory locations. To summarise, the variations in clock cycles required for access observed within the processor hierarchy is shown in Figure 5.9.

5.2.2 QuadroCore Power Distribution

To study the power characteristics of the QuadroCore multiprocessor, the entire architecture comprising four processing elements and the corresponding local instruction and data memory was synthesised onto UMC's 90 nm standard cells. The processor core accounted to a gate count of about 130K gates and the local instruction and data memory were a total of 32K bytes. Using the synthesis tools, the default power values

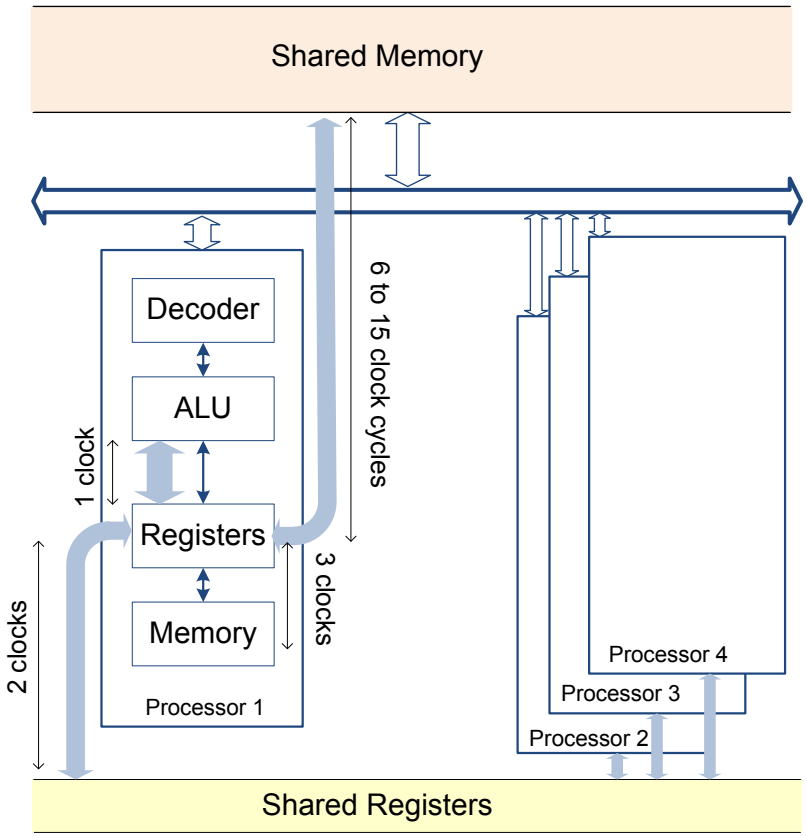


Figure 5.9: Communication Hierarchy

were measured. In Figure 5.10, the power distribution plot for the reconfigurable multiprocessor shows that 80% of the total power is dominated by memory. The remaining 20% is the contribution from the processor cores.

Power Estimation

On analysing the power distribution in the QuadroCore multiprocessor, the total power consumption in QuadroCore ($P_{QuadroCore}$) is expressed as the cumulative power of the processors' building blocks. The selection of the processors' building blocks depends on the operations executed within the instruction life cycle. The total power consumed may be represented as:

$$P_{QuadroCore} = 4 * \sum_{i=1}^{N} (P_{imem_i} + P_{dmem_i} + P_{exec_i} + P_{reg_i} + P_{recon_i}) \tag{5.2}$$

where, N is the number of instructions, 4 is representative of the number of processors in QuadroCore, P_{imem} is the power contribution by the instruction memory, P_{dmem}

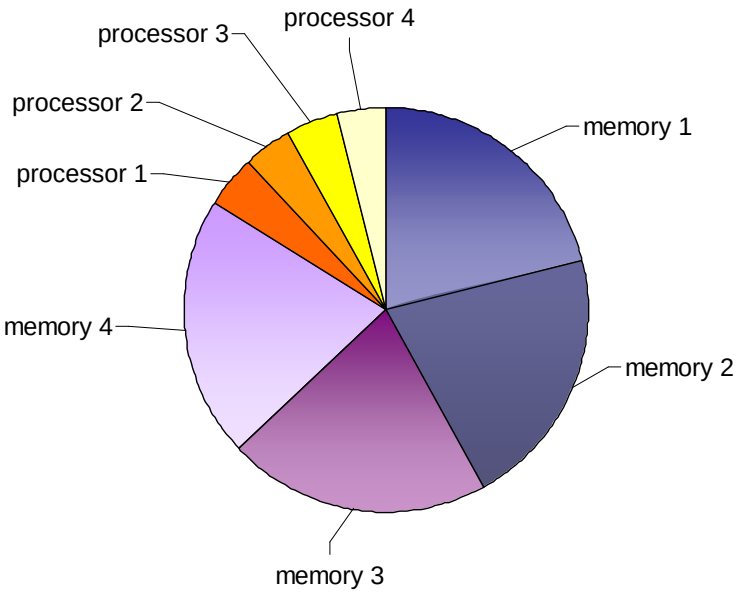


Figure 5.10: Core Power vs. Memory Power

is that of the data memory, P_{exec} is the power for the execution unit, P_{reg} is for the register file, and P_{recon} is the power consumption on account of reconfiguration.

5.2.3 Time and Power variations in the Reconfiguration Design Space

Table 5.6 lists the performance impact on the individual time and power components for each of the modes in the reconfiguration design space. Each of the modes has a distinct impact on time and power characteristics, which are determined by the application that is mapped onto the QuadroCore multiprocessor. For instance, the choice of the type of synchronisation is a trade-off between the frequency of synchronisation, which influences T_{sync}, the corresponding impact on power P_{imem}, or the change in code size, which alters T_{recon}. Similarly, in comparison to the MIMD mode, the SIMD mode has reduced instruction fetches (only for one processor), which results in savings in P_{imem} at the cost of addition time (T_{recon}). In case of register-sharing the additional power required is only for the register file borrowed from the neighbouring processor and the power required for reconfiguring the register inputs. With configurable ALUs, multiple identical operations can be executed simultaneously, affecting T_{total}, T_{recon}, and P_{imem} and power consumed by the addition ALU (P_{alu}). Thus, the choice of operating mode is based on application-specific characteristics, which can be altered during application definition.

Table 5.6: Performance Impact based on Reconfigurable Modes

Architecture	Time	Power
ASYNC/SYNC	T_{sync},T_{recon}	P_{imem}
SIMD/MIMD	T_{recon}	P_{imem}
Register Sharing	T_{recon},T_{comm}	P_{dmem},P_{imem}
Configurable ALUs	T_{total},T_{recon}	P_{imem},P_{alu}

5.3 Instruction-level Power Model

For processor-based designs, application-specific energy consumption is primarily recorded after mapping the application on the architecture. Hence, energy estimations are observed as an 'after-effect' rather than a design criterion. Although tools allow back-annotating the switching activity during synthesis, at this stage no significant reduction in power or energy can be observed since the application-to-architecture-mapping (which determines the entire dynamic activity) does not consider power as a design criteria. More particularly, the dynamic power is entirely application dependent, hence application-specific power adaptations can only be applied during application mapping. High-level model-based approaches such as [93, 94, 95], characterise applications and application-level transformations for power and energy, based on the switching activity measured for the application running on the target processor. These tools provide instruction-accurate estimates for power and energy as a measure for high-level transformations on energy. Similarly, in [83], the reduction in hamming distance is used as a method for reducing the overall switching activity. This method is explored by reducing the hamming distance between consecutive processor operations by compiler-driven register name adjustment and dead register reassignment. It accounts to an overall power savings in the range of 25-44% of the core power. In our perspective, application-level transformations influence the processor-to-memory interaction and make an impact on system power (processor + memory). These transformations mainly influence the choice of instructions, which in turn alters the processor-to-memory interactions, influencing the total system power. When comparing the total power of the system consisting of the processor power and the memory power, the impact of strategies applied to the processor power alone makes a very small impact. In line with our investigations, [96] also emphasises that the impact of memory power on the total power is significant. In [96], the authors present the impact of loop transformations such as loop tiling, loop unrolling and loop fusion, used to reduce memory accesses as a method of reducing power. Similar to this approach is our objective — to correlate memory transactions to the basis of instruction selection, which ultimately determines the power contribution of the memory subsystem. Hence, high-level transformations, in our perspective, relate to variations in instruc-

tion selection. Instruction scheduling described, e.g., in [84, 85] aims at reducing the switching activity based on the instructions selected and the operands used. The focus is to vary the schedule of instructions selected to influence the inter-instruction or the circuit-state effect and not the choice of instructions. Consequently, the only criteria for instruction selection has been time, hence the concept of code-generation for low power is the focus of our investigation.

Typically, instructions are classified on the basis of the number and type of operations, type of operands, and addressing modes. Based on these classifications, an instruction is weighed on the basis of the time (in terms of clock cycles) required to execute the instruction. Hence, during compilation instruction selection mainly relies on the number of clock cycles as a criterion to choose the sequence of instruction to match the control and data flow graphs of the application. In some cases, the number of instructions required to represent a basic block is also a criterion to reduce the program code size. Therefore, the stage where application-to-architectural transformation is performed entirely on these selected instructions. The focus here is to characterise instructions in terms of time and power. Thus, instruction selection relies on timing and power perspective for overall energy minimisation. This methodology characterises instruction-level power using transactions on the basic building blocks of the processor (such as register file, ALU and memories, which can be quantified in terms of power). This method enables instruction-specific attributes to be customised to suit the instruction set architecture. Hence, the methodology and the model are both extensible. Additionally, this method is independent of the application-specific input data, inter-instruction effects, input data, and methods of instruction scheduling. A simple trade-off between time and power has been identified using the access counts to the processor's building blocks.

For each instruction P_{instr}, the total system power is the sum of the dynamic power and the static power. But, for the implementation in today's standard cell technologies (90 nm, 130 nm etc.), the dynamic power exceeds the static power in orders of magnitude, hence we only consider the dynamic power. The total dynamic power consumed directly relates to the average switching activity or S_{rate}. Thus instruction-level power is given by :

$$\Rightarrow P_{instr} \quad \propto \quad S_{rate} \tag{5.3}$$

Where S_{rate} is the switching rate (or toggle rate), obtained by measuring the switching activity within a given instruction duration, given by:

$$S_{rate} \propto \frac{N_{toggles}}{T_{instr}} \tag{5.4}$$

where, $N_{toggles}$ is the total toggle count for a given instruction and T_{instr} is the instruction length.

5.3.1 Instruction Life Cycle

In this section, the time and power consumption for each instruction is modelled based on the intermediate operations in the instruction life cycle. The total life cycle of an instruction is composed of the instruction fetch, decode, register read, execute and finally the write to the register file. In the following sections based on these processor transactions, instructions are characterised for both for power.

The instruction-level power (P_{instr}) may be given by:

$$P_{instr} = P_{imem} + P_{dec} + P_{exec} + P_{reg} \tag{5.5}$$

where, P_{imem} is composed of the power consumed when fetching the instructions and directly relates to the number of instructions executed, P_{dec} is the power consumed by the decoder for each instruction execution, P_{exec} is that of the execution stage, and P_{reg} corresponds to the power consumed during register read and write stages.

P_{exec} depends on the type of instruction executed. Hence, we classify types of instruction as follows:

$$P_{exec} = \begin{cases} P_{alu} & \text{for arithmetic and logical operations,} \\ P_{reg} & \text{for mov instructions,} \\ P_{dmem} & \text{for load or store operations,} \\ P_{sync} & \text{for synchronisation instructions.} \end{cases} \tag{5.6}$$

where, P_{alu} is the power consumed by the ALU, P_{reg} is the power consumed by the register file, P_{dmem} is the power consumed by the data memory, P_{sync} is the power consumed during synchronisation instruction.

Also, P_{reg} depends on the number of registers specified as operands in the instruction that need to be read (or written) from (or to) the register file. Hence, for QuadroCore's instruction set,

$$P_{reg} = \begin{cases} P_{reg-read} & \text{for store operations,} \\ P_{reg-write} & \text{for load operations,} \\ n * P_{reg-read} + m * P_{reg-write} & \text{for ALU operations.} \end{cases} \tag{5.7}$$

where, $P_{reg-read}$ is the power consumption for register read during for store operations, $P_{reg-write}$ is the power consumption for register write during for load operations and n is the number of source register operands and m destination operands. Hence, using Equation 5.5,5.4 the total energy consumption (E_{total}) that is expressed independent of the input data and inter-instruction effects is given by:

$$\begin{aligned} E_{total} &= \sum_{i=1}^{n} P_{instr} * T_{instr} \\ &= \sum_{i=1}^{n} (P_{imem} + P_{dec} + P_{exec} + P_{reg}) * T_{instr} \\ &\propto \sum_{i=1}^{n} (S_{mem_i} + S_{dec_i} + S_{exec_i} + S_{reg_i}) * T_{instr} \end{aligned} \tag{5.8}$$

where, n is the total number of instructions and S is the switching activity recorded for each of the resources, viz., instruction memory (S_{imem}), decoder (S_{dec}), execution unit (S_{exec}), and the register file (S_{reg}).

5.3.2 Memory Accesses

The time characteristic of an instruction is given by the execution time or the instruction length. In contrast, the power consumption relates to the average switching activity recorded during the entire instruction cycle. Using Equation 5.9, the switching activity recorded for the processor's memory blocks relates directly to the number of memory accesses, given by:

$$\begin{aligned} S_{mem} &= S_{imem} + S_{dmem} \\ S_{mem} &\propto (N_{imem} + N_{dmem}) \end{aligned} \tag{5.9}$$

where, S_{mem} is the total switching activity of the memories, which is the sum of instruction memory S_{imem} and data memory S_{dmem}. N_{imem} and N_{dmem} are the access counts at the instruction and data memory respectively.

Hence, a reduction in the number of instructions or the choice of instructions with reduced memory accesses (both data and instruction) corresponds to lower switching activity and hence lower power dissipation. In addition it may also be observed that a single long instruction corresponds to lower power consumption in comparison to multiple short instructions to achieve the same functionality (due to reduced average activity incurred on account reduced instruction memory accesses).

As seen in Figure 5.10, the local memories dominate the power plot, which further emphasises the need for reducing processor initiated memory transaction to reduce the total system power. Since the total switching activity is inversely proportional to the instruction length (see Equation 5.4, instructions with larger execution length correspond to a reduced frequency in memory accesses. This reduction in memory accesses and hence switching activity results in lower power consumption compared to instruction with shorter execution lengths.

5.3.3 Register Accesses

Analysing the core power of our reconfigurable multiprocessor QuadroCore, it was seen that the register file contributes to 30% of the core power, which suggests register accesses as the next level of application-specific power contribution. Thus, switching activity associated with register file is directly related to the number of register accesses (N_{reg}) made in a single instruction cycle.

$$S_{reg} \propto N_{reg} \tag{5.10}$$

5.3.4 ALU Accesses

The next dominant component contributing to power, is the ALU which accounts to 20% of the core power. Switching activity at the ALU-level is data-dependent, which implies that variations in the input data patterns alter the switching activity. These variations can be observed with variations in the word-length of the input data and the type of ALU operation. Furthermore, as only 20% of the total processor power (core + memory) is influenced by the core power contribution, a reduction in the ALU power only makes a very small impact in the total system power optimisation.

5.3.5 Multiprocessor Synchronisation

When considering multiprocessors, co-operative operations such as data sharing and data exchange necessitates inter-processor synchronisation, based on data-dependency determined during compilation. Typically, in synchronous VLIW processors NOPs are inserted to assert wait states or stalls. In asynchronous multiprocessors, barrier synchronisation is a well-known technique in inter-processor synchronisation. Our reconfigurable multiprocessor supports both these methods; hence the analysis for the right choice of synchronisation mechanism was evaluated both with respect to time and power. Each NOP instruction corresponds to an instruction fetch and results in a single stall cycle. Whereas, a single barrier instruction could result in multiple stall

cycles, making it power-friendly on account of reduced processor-to-memory interactions. Thus, the power consumed during synchronisation is directly proportional to the number of memory accesses per stall cycle.

5.3.6 Instruction Set Characterisation

As a first step to analyse power, the power distribution of the processor was analysed for our reconfigurable multiprocessor architecture. To analyse the impact of instruction-level attributes on the variation in total power, a diverse set of instructions were chosen. Further, to make an accurate estimate of instruction-level attributes on power, the toggle rate was recorded exactly for the instruction duration. An accurate power measurement was made using the value change dump (VCD) file obtained for the exact instruction duration based on gate-level simulations at 200 MHz on the synthesised netlist generated using UMC's 90 nm standard cells. Power measurements were made using PrimeTime PX from Synopsys [97]. Figure 5.11 shows the variations observed in the total power and in the core power for the set of chosen instructions.

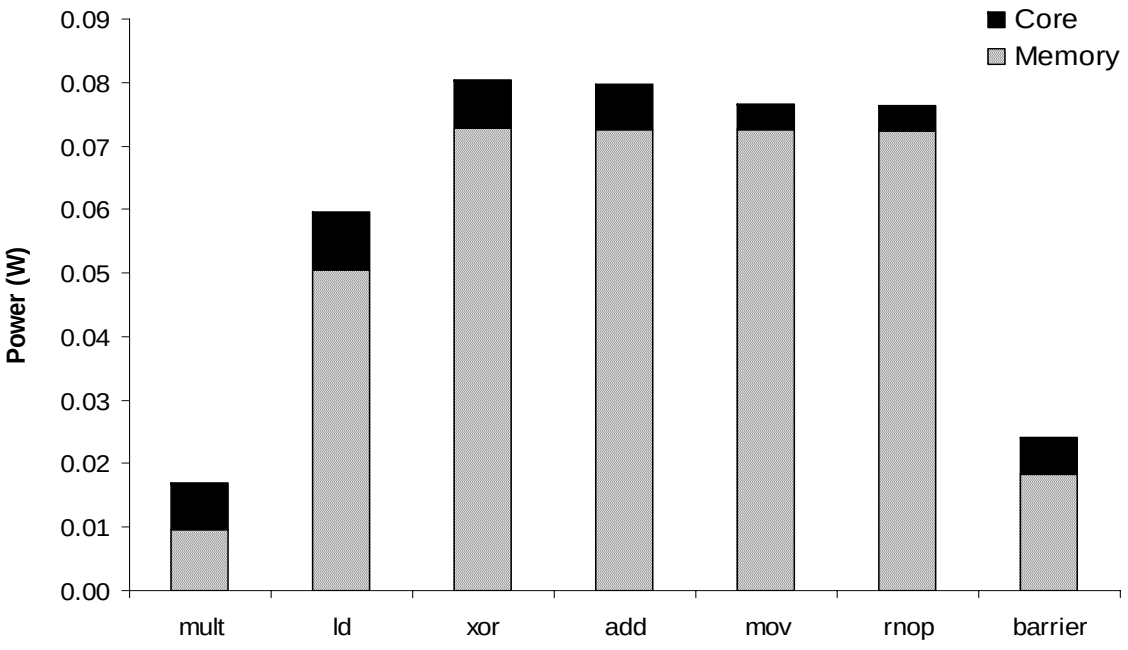


Figure 5.11: Comparing Instructions in terms of Core Power and Memory Power

Since gate-level simulation for every instruction is time consuming, a subset of instructions was chosen as representative instructions. The choice of instructions was made such that it could be representative for most of the instructions in the QuadroCore instruction set architecture. Hence, a load (*ld*) instruction is a representation of all instructions with a register access and a memory transaction (which also includes

store instructions). An *xor* (or *add*) instruction is a representation of all single cycle, ALU-based arithmetic instructions with two register reads, and a register write (which includes instructions such as *nor*, *and*, *add*, *sub*, *shift*, etc). The *mov* instruction represents a single-cycle instruction with a register read and a register write. Special instructions that were included for synchronisation are *barrier* and *rnop*. *barrier* initiates a barrier synchronisation mechanism and *rnop* adds stalls to the unused processors. As can be seen, the core power accounts for only about 10% of the total power consumption in all the cases.

Comparing Instruction Length, Power, and Energy

To analyse the variations in time and power for each of the instructions, Table 5.7 lists the instruction length in clock cycles, the total power consumption during the execution of the instruction, and the total energy consumption (product of execution time and power consumption for the instruction duration).

Table 5.7: Comparing Instruction Length, Power, and Energy

Instruction	Length	Total Power	Total Energy
mult	18	0.0139 W	1.251 nJ
ld	3	0.0596 W	0.894 nJ
xor	1	0.0855 W	0.428 nJ
add	1	0.0799 W	0.399 nJ
mov	1	0.0767 W	0.384 nJ
rnop	1	0.0763 W	0.382 nJ
barrier	1 to N	0.0241 W	0.121 nJ

The instruction *mult* has the maximum execution time of 18 clock cycles and the lowest power consumption. However, the execution time overrides the power consumption, resulting in higher energy consumption. The *ld* instruction has a 3 cycle execution time. However, the power consumption for this instruction is significantly higher than the *mult* instruction. As a consequence of lowered instruction length, the energy consumption of *ld* is lower than that of *mult*. Instructions *xor*, *mov*, *rnop* have the same instruction length and almost the same power and energy consumption. These three instructions represent ALU operations. The barrier instruction has a variable instruction execution time, since it depends on the instruction executed on other processors as well. The above example shows that the instruction length is larger than that of a *rnop* instruction and has a power consumption that is lower than the other instructions. Consequently the *barrier* instruction has the lowest power consumption, since the processor stalls all operations and waits for the other processors to encounter

a barrier instruction. A detailed analysis of the variations in power in each of the instruction follows in the next section.

Impact of Accesses on Core Power and Memory Power

Table 5.8 lists the variations in number of access to the processor's resources and the impact on the core power and memory power consumption for each of the above mentioned instructions.

Table 5.8: Comparing Accesses, Core Power, and Memory Power

Instruction	Memory Accesses	Register File Access	Core Power	Memory Power
mult	1	3	7.63 mW	0.063 W
ld	2	2	9.21 mW	0.050 W
xor	1	3	12.5 mW	0.073 W
add	1	3	7.33 mW	0.073 W
mov	1	2	4.17 mW	0.073 W
rnop	1	0	3.85 mW	0.073 W
barrier	1	0	5.83 mW	0.018 W

Firstly, the frequency of memory accesses per instruction directly influences the memory power and hence the total energy. The multiplication operation has a instruction memory access frequency of $\frac{1}{18}$, which is the longest with respect to time, least in terms of average memory power. Similarly, for a *load* instruction the frequency is $\frac{2}{3}$ as compared to *xor* $\frac{1}{1}$. This hypothesis can be extended to instructions that require multiple registers to be loaded (or stored) to (or from) memory based on a single instruction. It exemplifies the case where a single instruction fetch and decode leads to multiple loads (or stores), which may be advantageous as compared to multiple load (or store) instructions. Similarly, another analogy for this case is with the load (or store) instruction, where the data to be loaded resides at an address location identified via a relative addressing with respect to the program counter or from PC-relative address. Such a load requires multiple memory accesses for a single load, which corresponds to higher power consumption for every load (or store).

Further, for instructions with the same number of memory accesses, the number of register accesses directly influences the core power. These variations can be observed in the instruction *add* (3 accesses) in comparison to *mov* (2 accesses), where the core-power is directly proportional. Finally, addressing the type of synchronisation, a *nop* (1 instruction for every stall cycle) is expensive compared to a *barrier* instruction (one instruction for the entire duration of the stall).

The significant difference in core power and memory power suggests using methods such as using multiple voltage domains and frequency domains for core and memory organisation.

Impact of Operand Values

As a second step, the variations in power with changes in the input operands were measured and are listed in Table 5.8. Two instructions that are identical in the number of memory accesses and the number of register accesses (*add* and *xor*), were chosen to observe the variations in power, which correspond to energy since the time for executing these instructions is the same. These two instruction show a minimal variation in core power, in accordance to Equation 5.6. Table 5.9 summarises the variations based on the operands on two different instructions of the same length. As can be seen, a variation of up to 38% in core power has been noticed for the test case, but the overall impact on total power is a minimal 5%.

Table 5.9: Variations based on Operands Values

Instr.	Core Power	Change in Core Power	Total Power	Change in Total Power
xor	0.013 W	38%	0.0805 W	5%
add	0.007 W	2.7%	0.0799 W	0.12%

5.4 Impact of Compilation Techniques

To investigate the mutually opposing costs and benefits of time and power characteristics during instruction selection and code generation, the following standard compiler optimisation strategies were considered. The classical approach is to apply these strategies with the objective of optimising time; however the contrasting effects when measuring power has been highlighted below.

Data-flow Optimisations

Optimisations such as common sub-expression elimination, constant folding and constant propagation are methods of target independent optimisations, which involve eliminating unnecessary computations, resulting in reduced code. Reduction in code size corresponds to reduction in memory transactions on account of reduced instruction fetches. Hence, these transformations make a positive impact on both time and power.

Loop Transformations

Methods such as loop-unrolling and loop fusion have been frequently applied for code optimisation. The purpose of introducing loop unrolling is to avoid using loop-specific overhead in the control statements and branch operations. Similarly, loop fusion is used to increase the amount of work done in a loop and to reduce the number of iterations. Both these techniques amount to reduced control instructions, which implies reduced instruction fetches and hence memory accesses. This makes a positive impact on overall energy reductions, as long as the code resides in the local cache. Consider a simple for loop, with the following bounds:

```
for(int i=0;i< loop-size;i++)
    a[i] = b[i] + c[i];
```

Table 5.10 compares the assembly code for the above code with and without loop unrolling. Using the instruction-level power model, the difference in the memory, register, and ALU accesses are computed for the two variations. These variations relate to variation in power, which has been obtained using gate-level simulations and are listed in Table 5.11.

The assembly code in Table 5.11 compares the advantages of loop-unrolling with respect to power for a loop-size of 10 recursions. The savings in power correlate to the formulations presented in Section 5.3 expressed as transactions on processor building blocks, which align with the suggestions in [96].

Strength Reduction

Typically, instructions are characterised with respect to time, hence strength reduction directly involves replacing time-expensive operations with combinations of one or more less expensive instructions. This may result in increased code size or number of instruction, but results in lower execution time. However, an increase in number of instruction results in an increase in the number of instruction fetches. This corresponds to an increase in number of memory accesses and therefore an increase in power. Although strength reduction is beneficial in terms of time, it may result in increase power consumption. A sample case of replacing time-expensive multiplication with a combination of shift and addition operations is shown in Table 5.12. Multiplication operation requires 18 clock cycles (90 ns) and is replaced by a combination of shift & add instructions. This replacement results in a 50% reduction time required for execution. However, the difference in power consumed is 85%. This results in an overall difference of 70% in energy consumption, justifying the mutually opposing nature of this optimisation strategy for time and power.

Table 5.10: Comparing Code with and without Loop-unrolling

```
** Without loop unrolling **
      movi r2, 0 // init. loop counter
      lrw r3, mA //address of A
      lrw r4, mA //address of B
      lrw r5, mA //address of C
loop: ld r6,(r3)
      ld r7,(r2)
      add r6,r7
      st r6, (r5)
      incrmt r3

      incrmt r4
      incrmt r5
      incrmt r2
      cmplti r2, loop-size
      bt loop
      jmp main



Memory Accesses
= 7 + (12 * loop-size) + 3
Register Accesses
= 4 + (17 * loop-size) + 1
ALU Accesses
=      (2 * loop-size) + 2
```

```
** With loop unrolling **
lrw r3, mA //address of A
lrw r4, mA //address of B
lrw r5, mA //address of C
ld r6,(r3)
ld r7,(r2)
add r6,r7
st r6, (r5)
incrmt r3
incrmt r4
incrmt r5
//second recursion
ld r6,(r3)
ld r7,(r2)
add r6,r7
st r6, (r5)
incrmt r3
incrmt r4
incrmt r5
//third recursion
...
..
Memory Accesses
= 3 + (10 * loop-size)
Register Accesses
= 3 + (14 * loop-size)
ALU Accesses
=       4 * loop-size
```

Table 5.11: Loop Transformations: Variations in Time, Power, and Energy

Strategy	Time (ns)	Total Power (W)	Core Power (W)	Mem. Power (W)	Tot. Energy (nJ)
For loop	1600	0.067	0.013	0.054	107
Unrolled	1000(-37%)	0.058(-13%)	0.012(-8%)	0.046(-15%)	58(-46%)

Table 5.12: Strength Reduction: Variations based on Instruction Selected

Strategy	Time (ns)	Total Power (W)	Core Power (W)	Mem. Power (W)	Tot. Energy (nJ)
Combined	45	0.084	0.012	0.072	3.78
Multiply	90 (+100%)	0.013 (-85%)	0.007 (-42%)	0.006 (-92%)	1.17 (-70%)

Re-materialisation

In order to optimise time, reloading a register from memory is substituted by re-computing the memory value. Although, it eliminates time expensive memory access with less expensive ALU and register access, it may result in an increased number of instructions (which is entirely data/address dependent). This results in an increase in the instruction memory accesses and as a consequence an increase in the overall system power. Although, if re-materialisation is achieved by replacing a spill by a fewer computable instructions, it may be advantageous in terms of power, as depicted in Table 5.13.

Table 5.13: Comparing Code with and without Re-materialisation

```
** Register spilling**                        ** Recompute register value**

addi sp, offset  //location to spill          ldi r2, 0xa     //compute reg.
st r2,(sp)       //copy to stack
..                                            or
..
subi sp, offset //recomp. location            ldi r2, 0xFFFFFFF   // multiple instr.
ld r2, (sp)     //restore reg.                sar r2, 10          // to compute reg.

Mem. Accesses  = 6 + offset comp.             Mem. Accesses = 2
Reg. Accesses  = 6                            Reg. Accesses = 2
ALU Accesses   = 2                            ALU Accesses  = 0
Clock Cycles   = 6 + offset comp.             Clock Cycles  = 2-3
```

Based on these examples, it is observed that instructions characterised in terms of time do not always imply power and energy optimisation. Hence, true characterisation of instructions need to include both time and power as parameters for optimal code-generation and steer application mapping for energy optimisation.

5.5 Implementation and Performance Measurements

This section analyses the performance reports for the QuadroCore multiprocessor realised using standard cell technology and Xilinx's FPGA. Based on the implementation reports, prominent performance differences and deviations are discussed.

5.5.1 Standard Cell Implementation

The design composed of the four processors and their respective instruction and data memory, was synthesised with UMC's 90 nm standard cell technology under typical operating conditions, where voltages was 1.0V and temperature of 25C. Each of the processors has a 16×32 registers file, 32-bit ALU and 32K local instruction and data memory. For UMC's 90nm standard technology, a 2-input NAND gate has a gate-count of 4 and a gate density of $400/mm^2$. With these inputs for the base cell, area reports are obtained after synthesis and place route using Synopsys's Design Compiler and Cadence's SoC Encounter.

Table 5.14: Performance Comparison: Impact of Reconfigurable Modes

Architecture	Clock Frequency	Area	Total Power
Original Multiprocessor	476 MHz	0.27 mm^2	11.53 mW
QuadroCore	454 MHz	0.34 mm^2	11.36 mW

On adding reconfigurable modes to the original architecture, a change in area and operating frequency is shown in Table 5.14. A change of 9% in area and operating frequency of 3% was observed as compared to the original architecture [98]. The layer of interconnects are controlled via instruction set extensions to alter the control and dataflow between the decode, execute, and register access stages. These enhancements ensure that the base instruction set architecture is reused and reconfiguration is managed at a high-level of abstraction, as suggested using the above-mentioned programming model. All these configurations and the programming constructs result in a reduction in the number of instructions, hence reduced memory transactions, and reduced power consumption. The leakage power using UMC's 90nm standard cell technology is in the range of 0.8 mW. The benefits in terms of clock cycles are at a cost of additional multiplexers and additional routing interconnects between the processing stages, which result in reduced overall frequency. Therefore, the presented methodology is a trade-off between achieved clock frequency and the cycles required for application execution.

Area Reports

Table 5.15 compares area reports for synthesis at 222 MHz. In terms of area, it is may be noted that a single NCore contributes to an insignificant amount of 3% of the total core area. The processing engine or the ALU contributes to 20% of the processor core, and the register file to 40% of the processor area. The compute intensive unit of the core corresponds to 20% of total, where as the storage unit corresponds to 40%, and

the rest is the control logic (40%). The memories itself correspond to nearly 88% of the total core area.

Table 5.15: Comparing Area : Compute and Communication Overhead

Architecture	**Area** (mm^2)	% **Area**
ALU	0.02	0.6
Decoder	0.005	0.1
Register File	0.04	1.2
Address Generation	0.003	0.1
Other Control Logic	0.03	1.0
NCore	0.08	3
4 × NCore	0.32	12
4 × Memory	2.30	88
QuadroCore	2.62	100

Core versus Memories

Table 5.16 shows the difference in the performance of the QuadroCore with and without its local instruction and data memories. As can be seen, nearly 85% of power consumption is on account of the local memories. Additionally, maximum clock frequency of the processor with memories is lower than the cluster of four processors, on account of the slow memory interface. A single bank of instruction and data memory nearly corresponds to four processing elements in terms of power and eight processor core in terms of area.

Table 5.16: Impact of Introducing Memories

Architecture	**Clock Frequency**	**Area**	**Power**
QuadroCore (Core + Memory)	261 MHz	2.63 mm^2	39.10 mW
QuadroCore (Core)	454 MHz	0.34 mm^2	11.36 mW

From the detailed power reports for the QuadroCore multiprocessor operating at 1.0 V supply, it was seen that the total dynamic power is nearly 90% (37.05 mW) of the total power consumption (39.13 mW). It has to be noted that the dominant dynamic power is entirely application dependent. Hence any further power savings requires modifying the application definition and application mapping strategies for low power. Further, it was noted that about 86% of the total power was contributed by the on-chip memory. Among the rest (14%), the register file itself had a contribution of about 20% of power.

NCore, DualCore to QuadroCore

Table 5.17 shows the impact on time, area, and power for increasing number of cores. The clock frequency of operation remains marginally affected and is around 2%. However, the area and power change are in a linear relationship to the number of cores. The change in area from a 2-core cluster to a 3-core cluster is 32% and from a 3-core cluster to a 4-core cluster is 26%. Similarly in case of power, a change of 37% from a 2-core to 3-core and a change of 28% from a 2-core to 4-core cluster is noticed.

Table 5.17: Impact of Additional Processors

Architecture	Clock Frequency	Area	Power	mW/MHz
1-Core	280 MHz	0.67 mm^2	7.2 mW	0.03
2-Core	273 MHz	1.32 mm^2	18.1 mW	0.07
3-Core	270 MHz	1.97 mm^2	28.7 mW	0.11
QuadroCore	261 MHz	2.63 mm^2	39.1 mW	0.15

Impact to Technology Scaling

A comparative study for the implementation of the QuadroCore multiprocessor on UMC's 90 nm and 130 nm technologies was carried out. The 90 nm technology has a density of 400K gates per mm^2 and an operating voltage of 1.0V. The 130 nm technology has a density of 200K gates per mm^2 and a 1.2V operating voltage. With reduced process technology, the area reduces and the maximum achievable clock frequency is significantly higher. The side-effect of higher operating frequencies and lower area is the increase power consumption, as can be seen in Table 5.18.

Table 5.18: Impact of Technology Scaling

Architecture	Clock Frequency	Area	Power
130 nm	285 MHz	0.75 mm^2	9.6 mW
90 nm	454 MHz	0.34 mm^2	11.36 mW

Impact of Clock Gating

Table 5.19 shows the performance impact of introducing clock gating as a power saving technique in the QuadroCore multiprocessor. It results in a minor reduction in the maximum frequency of operation (about 0.8%) and increase in area (about 1%) and a significant amount of power savings. The power savings are nearly 31% of the total

power after introducing clock gating. The majority of power savings are due to the reduction in power introduced in the register files. With clock gating, the power contribution of register files reduces from 25% to 2% of the total power. The addition of 298 clock-gating elements accounts to a clock gating of 96% the total 8033 registers.

Table 5.19: Impact of Clock Gating

Architecture	Clock Frequency	Area	Power
QuadroCore without clock gating	270 MHz	2.67 mm^2	57 mW
QuadroCore with clock gating	261 MHz	2.63 mm^2	39.1 mW

QuadroCore Reconfigurable Modes : Costs

Table 5.20 lists the individual performance reports for QuadroCore's reconfigurable modes and the corresponding performance impact on the achieved maximum clock frequency, area, and total power (with the default switching activity).

Table 5.20: Standard Cell Synthesis Reports - Typical Operating Conditions

Architecture	Clock Freq.	Area	Power
QuadroCore: Base	276 MHz	2.50 mm^2	41.8 mW
QuadroCore: Sync	276 MHz	2.51 mm^2	42.0 mW
QuadroCore: Comm	276 MHz	2.61 mm^2	41.7 mW
QuadroCore: SIMD/MIMD	270 MHz	2.59 mm^2	40.7 mW
QuadroCore: ISEs	273 MHz	2.59 mm^2	41.0 mW
QuadroCore: FMA	273 MHz	2.59 mm^2	41.9 mW
QuadroCore: All Modes	261 MHz	2.63 mm^2	39.1 mW

In the table, *Base* refers to the original four processor cluster in the absence of any reconfiguration modes. *Sync* refers to the enhancements to the *Base* to support the synchronisation mode of reconfiguration described in Section 5.1.2. *Comm* refers to the enhancements to the *Base* to support the communication mode of reconfiguration described in Section 5.1.3. *SIMD/MIMD* refers to the enhancements to the *Base* to support the SIMD/MIMD modes described in Section 5.1.4. *ISE* refers to the enhancements to the *Base* to support the additional instruction set extensions that were included to support co-operative processing, discussed in Section 5.1.6. *FMA* refers to the enhancements to the *Base* to support the fast memory access mode described in Section 5.1.4. Finally, *QuadroCore: All Modes* includes all the above mentioned modes.

As can be seen, the overall impact on introducing the reconfigurable modes is about 5% in terms of achieved maximum clock frequency, 5% in terms of area, and 6% in terms of dynamic power. The maximum change in clock frequency is noticed in the *SIMD/MIMD* mode of operation, where the control path is significantly altered to broadcast control signals to all the processors. Similarly, the maximum change in area is noticed in the *Communication* mode, where the shared register file is responsible for the additional area. Figure 5.12 summarises the performance reports in terms of frequency of operation, area, and power consumption for each of the reconfigurable operating modes.

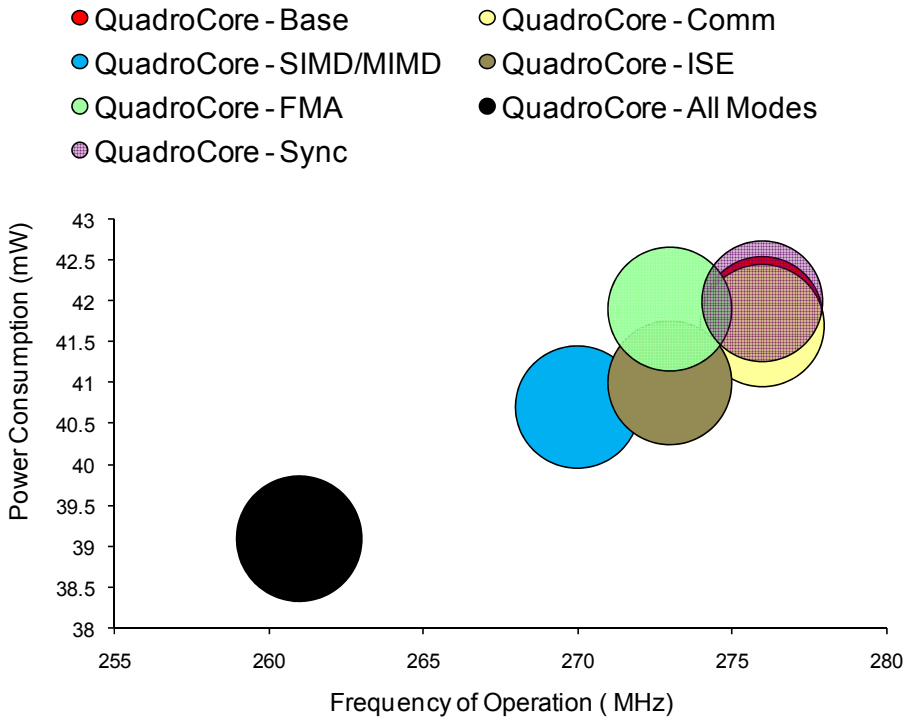


Figure 5.12: QuadroCore : Performance Reports

5.5.2 Post-layout Implementation Reports

Table 5.21 compares the implementations at 285 MHz, for the post-synthesis and post place and route (P&R) implementations. The results are compared first only for the core (without memories). The post-place and route reports indicate the area and power reports for a utilisation density of 85%. It has to be noted that the post-synthesis reports are estimations with an approximation for the interconnects and wires using a wire-load model. Whereas, the post-place and route reports include the actual report as obtained after the design has been routed.

Table 5.21: Post place and Route evaluations: Core Only

Architecture	Area	Power	Utilisation Density
QuadroCore (post-synthesis)	0.26 mm^2	6.62 mW	
QuadroCore (post P&R)	0.40 mm^2	19.82 mW	85%

Table 5.22 compares the implementation reports for the entire QuadroCore multiprocessor (core and memories) at 222 MHz. The reports are indicative of the impact of introducing actual wire-length estimations after the place and route phase.

Table 5.22: Post place and route evaluations - Core + Memory

Architecture	Area	Power	Utilisation Density
QuadroCore (post-synthesis)	2.63 mm^2	39.1 mW	
QuadroCore (post P&R)	3.33 mm^2	56.72 mW	68.9%

Figure 5.13 shows the floor plan for the major building blocks in the QuadroCore multiprocessor. The layout is a 'core-only' (without any I/O interface) implementation to analyse the impact of physical characteristics in the modifications introduced in the QuadroCore multiprocessor.

5.5.3 FPGA Reports

The entire architecture has been mapped to our scalable rapid prototyping system, RAPTOR2000 [99]. Using this environment, accelerated architectural prototyping for performance analysis of applications (described in C), is then possible. This experimental setup facilitates convenient cycle accurate performance estimations for large benchmarks. The FPGA implementation of the QuadroCore has an operating frequency of 12.5 MHz and occupies 58% of the slices and 53% of the Block RAMs on a Virtex-II 4000. In Table 5.23 time, area, power reports for the FPGA implementation that was made for functional validation.

Table 5.23: FPGA Performance Reports

Architecture	Clock Frequency	Slices	Block RAM	Power
FPGA XC2v4000	12.5 MHz	58%	53%	2 W

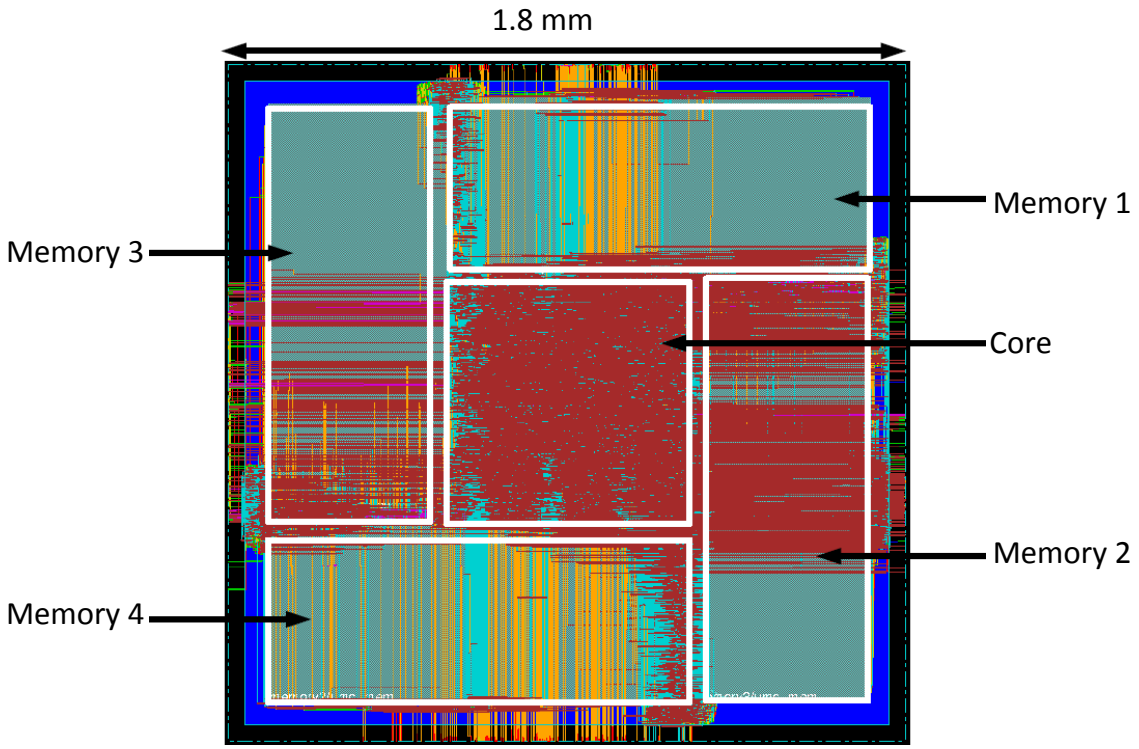


Figure 5.13: Post Place and Route Layout

Performance Deviation

Table 5.24 shows the performance deviation between standard cell and FPGA implementation. The area comparison is an approximation to FPGA equivalent gate count, included only for simplistic comparison.

Table 5.24: Performance Deviation

Architecture	Clock Frequency	Area	Power
QuadroCore- All Modes	261 MHz	783505	39.1 mW
FPGA	12.5 MHz	4455240	2 W
Performance Deviation	20.88	6	51

5.6 Summary

A reconfigurable multiprocessor template that provides fast adaptability to application-specific computation, communication, and synchronisation demands has been developed. Using this template, a reconfigurable multiprocessor — QuadroCore that has

four instances of our RISC processors (called NCore) has been realised. The focus of this approach is to use multiple instance of an existing processor core, introduce minimal modifications to the base instruction set architecture, in order to enable application-specific customisation via run-time reconfiguration. The most important advantage of this approach is the reduction design and verification time, and hence the time-to-market. Design space exploration is made possible via a set of reconfigurable modes. Reconfigurability in this context enables adapting the multiprocessor architecture to variations the degree of task-level parallelism, processor-to-processor communication, and the inter-processor synchronisation. The hardware overhead itself has a very minimal impact on account of the reconfiguration enhancements. In order to ascertain the performance benefits of the proposed approach, the QuadroCore multiprocessor has been realised on on UMC's 90 nm standard cell technology and Xilinx Virtex2 FPGA. The standard cell realisation has been used for performance measurements and the FPGA prototype aids in accelerated functional validation with large benchmarks. Analysing the performance changes compared to a fixed architecture, a change in operating frequency of 3%, area of about 9%, and power of 1% has been noticed. On the other hand, introducing these reconfigurable capabilities extends using the multiprocessor architecture over a diverse range of applications. Although the addition of flexibility in the architecture results in a reduction in the maximum operating frequency, and increase in area, is compensated by the reduction in the number of clock cycles and power required to perform a given task.

Chapter 6

QuadroCore: Applications

The QuadroCore reconfigurable multiprocessor adapts according the granularity of computation, communication demands, and synchronisation paradigms of the application. In order to analyse the impact of application-specific characteristics on performance, a diverse set of applications have been mapped to the QuadroCore multiprocessor. Firstly, to confirm the impact on the timing evaluation, a set of micro-benchmarks were mapped. Next, the costs and benefits of this approach with respect to execution time and power consumption for three different application domains viz., DSP applications, multiplier used in Elliptic Curve Cryptography, and Neural Networks based Self-organising Maps are analysed. Additionally, the built-in features in QuadroCore have been used for processor customisation that can be enabled via profile-driven resource estimation. Typically, methods of processor customisations are mainly driven by introducing instruction set extensions or adding application-specific dedicated hardware accelerators. Here, we limit our discussion to exploring the design space within processor architectures, with customisations that can be introduced during run-time. The application-specific functionalities and resource requirements are identified during compile time and the processor architecture is modified during run-time. A generic multiprocessor template is developed to aid such a mechanism, which is flexible enough to adapt to a range of processor-specific variations. For simplification, a standard processor limited with single-issue and in-order execution is chosen.

In this chapter, Section 6.1 shows the design flow employed for mapping applications with diverse characteristics to the QuadroCore multiprocessor. Section 6.2.1 shows the performance improvement in terms of time for a set of micro-benchmarks and shows the advantage of a reconfigurable, mixed-mode approach. As a proof of concept for the time-power optimisation, applications from three diverse domains have been mapped and their performance reports have been analysed in Section 6.2. Resource efficiency is the primary focus of mapping for these applications. Next, in order to extend the

usability of in-built flexibility, methods of processor customisations are presented in Section 6.3. Also in this section, a discussion on using QuadroCore multiprocessor as a platform for application validation, particularly with respect to parallel programs is included. In this case, the combined hardware-software validation is a distinct advantage.

6.1 Design Flow for Resource Efficiency

Performance and optimal resource utilisation have been the primary focus of designing our reconfigurable multiprocessor — QuadroCore. It provides a generic architectural template, where applications with diverse characteristics can be mapped. Its flexible nature provides adaptability in terms of granularity of computation, amount communication, and frequency of synchronisation to suit the application. The implementation details to support this scenario are discussed in the following sections.

Figure 6.1 shows the design flow used to adapt the QuadroCore multiprocessor to the application demands. The QuadroCore multiprocessor can be categorised as a clock-cycle programmable architecture, as described in Section 3.1. Thus, the maximum frequency of operation is fixed, and the programmable parameter is the clock-cycles required for execution. In this design flow, the objective is to optimise both time in terms of clock cycles required for computation and the power consumed at the instruction-level. This method of mapping is a two-stage process. In the first level, choice of the right reconfigurable mode is made according to the application's characteristics. This translates to a choice of reconfigurable modes incorporated in the hardware during application execution. The second stage is a process of application-specific instruction-level power and time optimisation, in software. For each operating mode, the choice of the appropriate set of instructions is made for the performance objective. The choice of the reconfigurable mode has a greater freedom for design space exploration. The choice of the mode can influence parameters that affect both time and power reports, as listed in Table 5.6. The next level of fine-tuning is achieved based on the instruction-level timing and power modes, described in Section 5.3. The design-flow is feedback directed and provides and estimation of the run-time statistics of the impact of application-level transformations on the target reconfigurable multiprocessor. Transformations can be introduced during the application specification (in software) and application mapping (in hardware). This new approach is generic and the performance objectives can be customised to suit the application-to-architecture mapping.

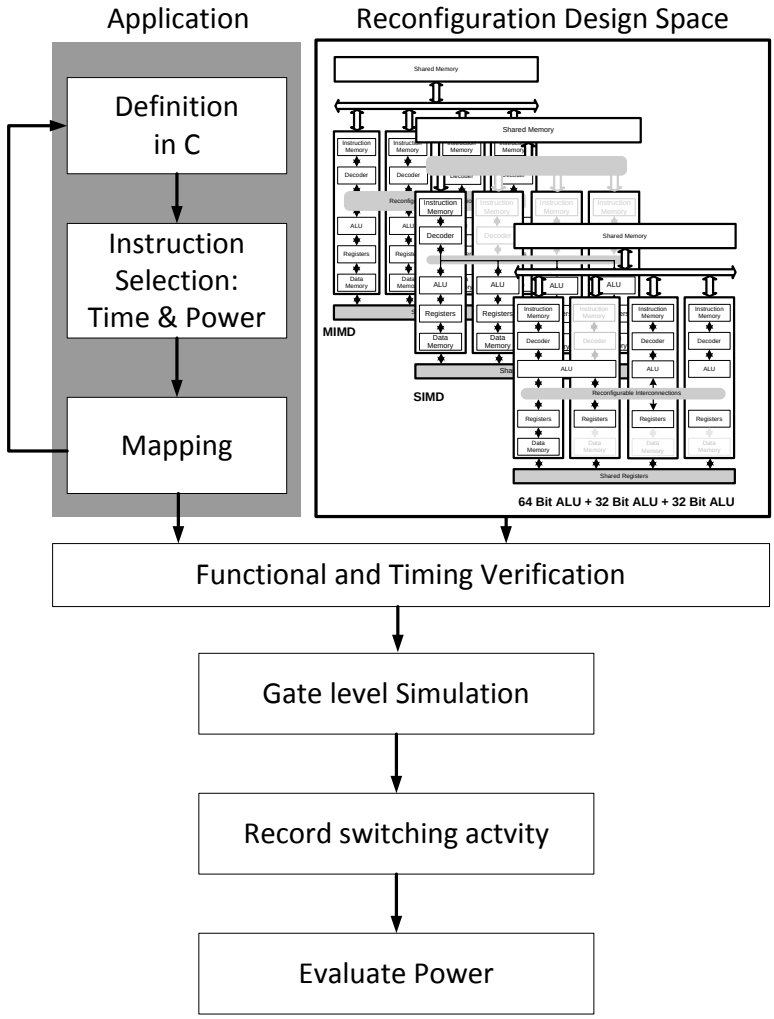


Figure 6.1: Design Flow for Application Mapping on QuadroCore

6.2 Applications Mapped to QuadroCore

In this section, the performance impact of mapping a set of applications onto the QuadroCore multiprocessor has been analysed. In Section 6.2.2, the variations in time, power, and energy for two DSP applications are reported. The focus here is to analyse the impact of application partitioning in terms of time, power, and consequently energy. In Section 6.2.3 a multiplier used in Elliptic Curve Cryptography, which is representative of applications with function-level or task-level parallelism is analysed. The variations in time and power, with reconfiguration used for power savings are analysed. In Section 6.2.4, a data-parallel application called Self-organising Maps has been analysed for energy savings achieved via reconfiguration and parallel data processing. The same application has also been mapped onto a state-of-the-art configurable processor to compare the difference in performance.

6.2.1 Timing Advantage of Reconfiguration

Experiments were performed using the parallelising compiler on the hardware implementation of the proposed reconfigurable multiprocessor architecture. The same was simulated with a cycle accurate simulator. The current prototype implementation of the CoBRA compiler performs a fine-grained parallelization on basic block level. This section presents evaluations of performance comparison of the reconfigurable operating modes. For the initial evaluation, small excerpts from practical audio and video applications were selected. These computational blocks constitute typical transcoding algorithms for aggregation network access nodes.

convolution: Computes the discrete convolution of a 50 element array with a 16 element array.

fft: Represents the variable access pattern of a Fast Fourier Transformation with two arrays of 16 elements each.

mm: Multiplies two 4x4 matrices.

sharpening: Sharpening algorithm for an image dimension of 10x10 pixels.

vectormuladd: Multiply-accumulate on vectors of 10 elements.

The compiler selects between asynchronous, synchronous or SIMD modes for each basic block or just use a single processor. Run-time mode change is enabled via a single cycle reconfiguration. We restrict our comparisons to a single processor and a cluster of four processors, although evaluations of the architecture with two, three of four processors may be performed. In Table 6.1, *ASYNC* represents the asynchronous mode, *SYNC* is the synchronous mode and *SIMD* is the SIMD mode. A combination of one or more modes is achieved via reconfiguration, as suggested by the compiler. It has to be noted that no single mode of operation is a true winner for all the applications. This further emphasises the point that a fixed hardware architecture may not be suitable for application, even within the same application domain. The results imply that the performance improvements depend on the type of the application and the corresponding mode of operation. For `convolution`, `mm` and `fft` the CoBRA compiler achieves a significant improvement in performance by partitioning the algorithm into four processors. Parallelising `fft` yields a speedup of 10, because the well-balanced register need avoids a significant amount of spill code in contrast to a single processor. Even when multiple processors access the external memory with a significant overhead, a performance increase can be observed compared to a single processor. However, in case of `vectormuladd` it may be seen that an increase in the number of processors does not have a positive effect on the performance (as in the case of synchronous and asynchronous mode). These results demonstrate that using the selected reconfigurable modes in our multiprocessor is beneficial, since the architecture allows switching between these modes and the compiler selects the optimal implementation for each piece

Table 6.1: Reconfigurable Modes: Performance Reports

Benchmark	Modes	Clock Cycles	Speedup
mm	Single Processor	681	
	ASYNC	**207**	3.28
	ASYNC + SYNC	226	3.01
	ASYNC + SYNC + SIMD	236	2.88
fft	Single Processor	34431	
	ASYNC	4105	8.38
	ASYNC + SYNC	**3165**	10.87
	ASYNC + SYNC + SIMD	3662	9.40
convolution	Single Processor	16871	
	ASYNC	**12330**	1.36
	ASYNC + SYNC	12428	1.35
	ASYNC + SYNC + SIMD	16231	1.03
sharpening	Single Processor	40069	
	ASYNC	38486	1.04
	ASYNC + SYNC	35602	1.12
	ASYNC + SYNC + SIMD	**29026**	1.38
vectormuladd	Single Processor	883	
	ASYNC	1482	*0.56*
	ASYNC + SYNC	1438	*0.61*
	ASYNC + SYNC + SIMD	**755**	1.17

of code. Additional analysis with time as the optimisation objective can be found in [HTK+07].

6.2.2 DSP Algorithms

Two algorithms frequently used in DSP applications, viz., Matrix Multiplier and FIR filter are mapped onto the 4-processor QuadroCore using the new reconfiguration design space.

Matrix Multiplication

Consider two $m-$dimensional square matrices, A and B, the resulting elements of the product matrix C is computed as given by:

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} * B_{k,j} \tag{6.1}$$

The computation can be divided into the following steps:

1. Distribute A and B among the four processors (corresponds to T_{comm})
2. $\sum_{k=1}^{m}(A_{i,k} * B_{k,j})$(corresponds to T_{comp})
3. Final result C is available in the individual processors

where, each element of the resulting matrix C requires m product computations and $m-1$ summations. The algorithm computes using $m \times m$ data elements. Step 1 corresponds to the communication overhead on account of partitioning, T_{comm}. Step 2 is the core computation time, T_{comp} it can be distributed across on all the processors. Since the operation is identical and parallelised it can be operated in SIMD mode, which influences T_{recon} and P_{im}.

FIR filter

The equation for realising an FIR filter is given as follows:

$$y(n) = \sum_{k=1}^{N} a(k) * x(n-k) \tag{6.2}$$

where *a(k)* is the coefficient of the FIR filter at tap *k*, *x(n)* is the input, *y(n)* is the output at time n and N is the length. For partitioning an FIR filter onto the 4-processor QuadroCore using the new reconfiguration design space, the algorithm is divided into the following steps:

1. Distribute *a(k)* and *x(n-k)* among the four processors (corresponds to T_{comm})
2. *y(k) = a(k) * x(n-k)*(corresponds to T_{comp})
3. Wait for end of computation, for all k elements (corresponds to T_{sync})
4. Collect *y(n)* from all the other three processors (corresponds to T_{comm})
5. $\sum_{k=1}^{N} y(k)$ (corresponds to T_{comp})

The choice of the mode is a compromise between the number of cycles required for execution (optimising for time) and the corresponding impact on power (optimising for power). The clock cycles depict the variation in execution time for each chosen mode, and the benefits reflect the corresponding savings in clock cycles or power that is achieved in each of the modes. Since the choice of the modes are entirely application dependent, accordingly are the costs and benefits evaluated. E.g., in Step 2, the MIMD mode of operation is beneficial in time, but results in higher power consumption in comparison to SIMD. Similarly, in Step 3, the SYNC mode results in lower synchronisation time, but ASYNC is power-efficient on account of fewer instruction memory

accesses. A detailed timing analysis of the variations in timing for each of the above mentioned steps when the available reconfigurable modes is discussed in [PP08b].

Impact on Power

The power characteristics of the QuadroCore depends on the number of active processors and the choice of the reconfigurable modes. Power scales linearly with the number of processors. The difference in power is the largest between the SIMD and the MIMD mode of operation. Thus, to ensure low-power operation, the QuadroCore is switched to SIMD mode, as much as possible. In the presence of data-dependent control flow or in the absence of data-level parallelism, all the processors need to operate in the default MIMD mode. For operation of N identical processors in MIMD mode, the total power is given by:

$$P_{mimd} = P_1 * N \tag{6.3}$$

where, P_1 is the power dissipated by a single processor. To save power, the QuadroCore multiprocessor is operated in the MIMD mode only in the presence of data-dependencies. Thus, in the best-case scenario, the QuadroCore operates entirely in the SIMD mode resulting the least power dissipation (P_{simd}), and in the worst case entirely in the MIMD mode (P_{mimd}) resulting in the highest power consumption. Thus, for QuadroCore:

$$P_{QuadroCore} = x * P_{simd} + y * P_{mimd} \tag{6.4}$$

where, x is the percentage of data-parallel SIMD component in the application, and y is the default MIMD component, such that $x + y = 1$, Figure 6.2 shows the change in power with increasing number of processing elements, for two application - viz., matrix multiplication and an N-tap FIR filter. The inherent application characteristics restrict operating in the low-power SIMD mode, thus making an overall impact on the total power consumption. Power consumption reduces with the introduction of the SIMD mode. This is proportional to the number of processors. The FIR filter switches back and forth between SIMD and MIMD modes of operation, where as the matrix multiplication has SIMD as the dominant mode of operation. Thus, for four processors, the application characteristics results in operation between the two extremes (SIMD and MIMD), as shown in the figure.

Impact on Energy

Energy has a combined influence on time and power characteristics. Energy consumption for the N processors is given by:

$$E_N = P_N * T_{total} \tag{6.5}$$

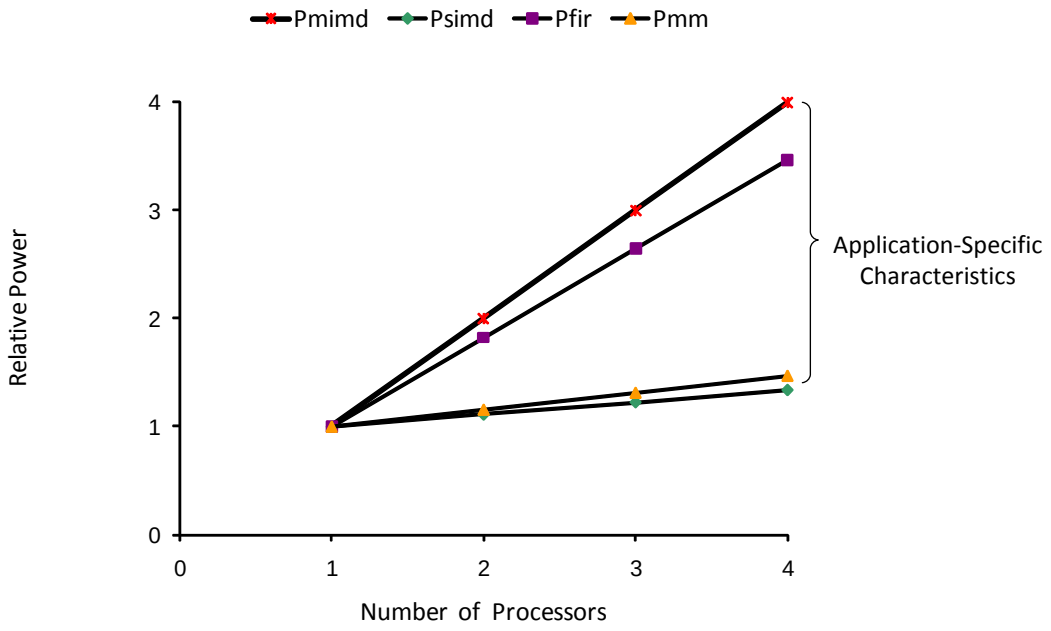


Figure 6.2: Application-Specific Power Characteristics in QuadroCore

Figure 6.3 shows the impact on energy, with increase in the number of processors and the impact of application-specific characteristics. Matrix multiplier represented by E_{mm} is a case of minimal impact on account of application partitioning, whereas E_{fir} is the case with increasing overhead in terms of communication and synchronisation overhead on account of partitioning. As can be seen, the SIMD mode of operation represents the best-case solution for time and power. The application mapping of an FIR filter on two processors is beneficial in terms of energy, than the case of additional processors. For matrix multiplication, the low overhead of partitioning provides a best-case solution with three processors. However, the change in the overhead for partitioning increases with increase in number of processors. To confirm the estimations, the two applications were mapped on the QuadroCore multiprocessor. The point denoted by $E\text{-}Q\text{-}FIR$ and $E\text{-}Q\text{-}MM$ represent the energy values for the FIR filter and the matrix multiplier respectively.

6.2.3 Multiplier used in Elliptic Curve Cryptography

The finite field multiplication in $GF(2^{233})$ used in Elliptic Curve Cryptography was modified using the Karatsuba's method [100], and was mapped onto the QuadroCore

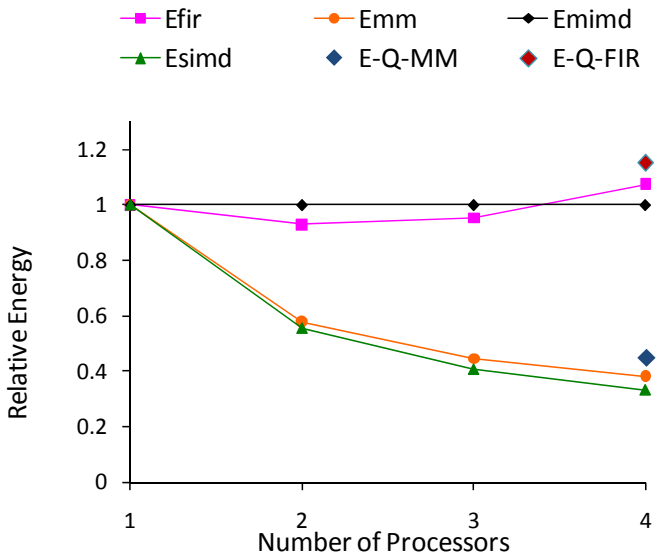


Figure 6.3: Impact of Application Partitioning on Energy

multiprocessor. To match the word-length of the individual processors ($w = 32$ bit), the input polynomials $a(x)$ and $b(x)$ are divided into eight 32 bit words:

$$\begin{aligned} A(x) &= \sum_{j=0}^{n-1} A_j \cdot X^{jw} \\ B(x) &= \sum_{j=0}^{n-1} B_j \cdot X^{jw} \\ C &= A \cdot B \end{aligned} \tag{6.6}$$

with $n = 8$,$w = 32$. The coefficients of higher degree than the considered binary field $(233 \cdots 255)$ are padded with zeros. By applying the Karatsuba method iteratively, the multiplication of binary polynomials of degree 232 can be calculated with 27 finite field multiplications at word-level. The application's characteristics were discussed in Section 3.4.2. The word-level multiplications are distributed to the four processors *PE1* $\cdots$ *PE4* of the QuadroCore multiprocessor. In this way, always four partial products are processed in parallel. Here the task executed is the same for all the processors. For each partial word-level multiplication the processors calculate the sum of the words j of the input polynomials $A(x)$ and $B(x)$. In binary fields, the sum of the input coefficients is easily calculated by an *xor* operation. The multiplication at word-level itself is performed using *shift-and-xor* instructions [101]. The product C is a polynomial of double word length, which is stored in two registers containing the high *(H)* and low *(L)* word of the product, respectively. Finally, the partial products are added, i. e. *xor'ed*, to the corresponding word segments c_i of the result.

Table 6.2 shows the variation in time and power for a case study on a multiplier in Elliptic Curve Cryptography, with an operating frequency of 200 MHz, UMC's 90 nm standard cell technology. A savings of 10% were observed when used in the SIMD-MIMD mode. In the SIMD-MIMD mode, the processors execute blocks of code in SIMD mode whenever possible. In case of data dependent variations to the control path, the mode is switched back to MIMD. The processors (*PE2*, *PE3* and *PE4*) are reconfigured to operate based on the instruction stream of the 'master' (*PE1*), when executing the same function, e. g. the word-level multiplication. Each time, reconfiguring between the modes requires one clock cycle. Since all the processors need to execute the same instruction stream, all the processors need to be synchronised before entering the SIMD mode. Hence, there is a difference in the execution times between SIMD and MIMD modes. The reconfiguration functionality was hand-coded in assembly as a first proof of concept. A power saving of 23.4% was noted when used in SIMD mode, as compared to the MIMD mode. The resulting energy saving was 15.54% as discussed in [PPP08]. An important observation is the energy savings observed even in the absence of an N-fold speedup.

Table 6.2: ECC: Performance Variations with Operating Mode

Mode	**Execution Cycles**	**Speedup**	**Power**(W)	**Energy** (μJ)
Single Processor	9402	1	0.017	0.799
MIMD	**3193**	**2.94**	0.063	1.012
SIMD−MIMD	3337	2.81	**0.057**	**0.957**

Scalability for Function-level Parallelism

The multiplier used in Elliptic Curve Cryptography, represents an application with function-level parallelism. The algorithm has 27 identical functions, which can be executed in parallel, provided the inter-function dependencies are satisfied. The inter-function dependencies permit a maximum parallelism of three functions to be executed in parallel. Hence, for equal load balancing a 3-processor QuadroCore is optimal. As also observed from the speedup reports in Table 6.2, it is in the range of 3, even for a four processor QuadroCore. This example proves the usability of the QuadroCore multiprocessor to applications that exhibit function-level parallelism. In the presence of data-level dependency a single cycle mode switch to the MIMD mode or the use of an instruction helps in sharing the branch/jump condition (as discussed in Section 5.1.6).

Comparison to CoreVA

For comparison, CoreVA [62] - a VLIW processor that was developed in the research group is listed in Table 6.3. CoreVA has been extended to include instruction set extensions to accelerate the application (listed as CoreVA-Extended). The performance measurements for the two processors were made at 200 MHz.

The difference in the clock cycles for execution is accounted to the fact that the C_{comp}, C_{comm}, and C_{sync} for the two architectures are different, as listed in Table 3.6. The inter-task communication and memory access mechanisms that account for C_{comm} in a VLIW processor and a chip multiprocessor like QuadroCore are diverse. A VLIW processor like CoreVA relies on a shared memory access mechanism with a shared register file and shared instruction and data memory. However, the QuadroCore multiprocessor uses the shared register file only for exchange of data and stores all the shared data in the shared external memory. This difference in access mechanism makes a difference of 6-15 clock cycles for each data accessed.

Table 6.3: Comparing QuadroCore with CoreVA

Mode	**Execution Cycles**	**Power**(W)	**Energy** (μJ)
Single Processor	9402	0.017	0.799
MIMD	**3193**	0.063	1.012
SIMD−MIMD	3337	**0.057**	**0.957**
CoreVA	1839	0.207	1.90
CoreVA-Extended	1636	0.210	1.71

As can be seen, implementing the algorithm using the QuadroCore processor is beneficial in terms of power consumption and total energy requirement. Whereas, the CoreVA processor is beneficial in terms of clock cycles required for implementation. However, it has to be noted that the CoreVA processor has been synthesised on ST's 90nm standard cell technology, whereas QuadroCore has been implemented on UMC's 90nm standard cells. Additionally, the power measurements for the QuadroCore multiprocessor include the power consumption of the core and memory, obtained after post-synthesis simulation. Whereas, the power reports obtained for CoreVA processor are high-level estimates and exclude the power contribution of the local memories. As discussed in Section 5.2.2, the core power corresponds to only 20% of the total power. Furthermore, power saving strategies such as clock gating that has been introduced in QuadroCore result in significant power savings. Hence, the difference in power estimations between the two processors is a significant advantage for QuadroCore.

6.2.4 Self-organising Maps

Self-organising maps (SOM) are neural network based machine learning algorithms that are extremely data parallel in nature. This application was chosen as an example to observe the performance impact of mapping data parallel algorithms on QuadroCore. Details of the application-level characteristics and its characteristics in terms of the computation, communication, and amount of inter-task synchronisation have been presented in Section 3.4.3. As described, the algorithm is composed of two functions, viz., a best match search and an adapt-map function. Since all the data elements need to execute these two functions, the algorithm is representative of data-parallel applications. The functional representation of the algorithm as mapped onto the QuadroCore multiprocessor is shown in Figure 6.4. On partitioning data onto the four processors in the QuadroCore, the search operation is sub-divided into two stages. In the first stage, a local best match is found locally in each of the processors. Next, these local best match values are shared and a global best match is identified. After the best match has been located, the adapt-map function is performed using this global best match. Additional details of the algorithm may be found in [PPPR09]. For the algorithm itself, the total execution time (T_{total}) is given by:

$$T_{total} \propto |N||X| \dim(\overrightarrow{x}) \cdot epochs \tag{6.7}$$

where, $|N|$ is the number of Neurons, $|X|$ is the number of vectors, $\dim(\overrightarrow{x})$ is the dimension of the weight vectors, and epochs is the number of iterations.

For the implementation on QuadroCore, using Neuron parallel approach, the execution time $T_{QuadroCore}$ is given by the following equation:

$$T_{QuadroCore} \propto \left(\left\lceil \frac{|N|}{|PE|} \right\rceil \dim(\overrightarrow{x}) + t_{gBM} \right) |X| \cdot epochs, \tag{6.8}$$

where, PE is 4 for our QuadroCore processor and t_{gBM} is the additional time (in clock cycles) during which the four processors synchronise, share their local best matches, and find a global best match.

SOM on QuadroCore

As shown in Figure 6.4, the algorithm is subdivided into functions *find local best match*, *find global best match*, and *adapt Map*. Since *find local best match* and *adapt Map* perform exactly the same operations on a large amount of data, data is partitioned onto the four processors and these functions are executed in parallel on all the processors. Since the operations are identical on all the processors, they can be executed in SIMD

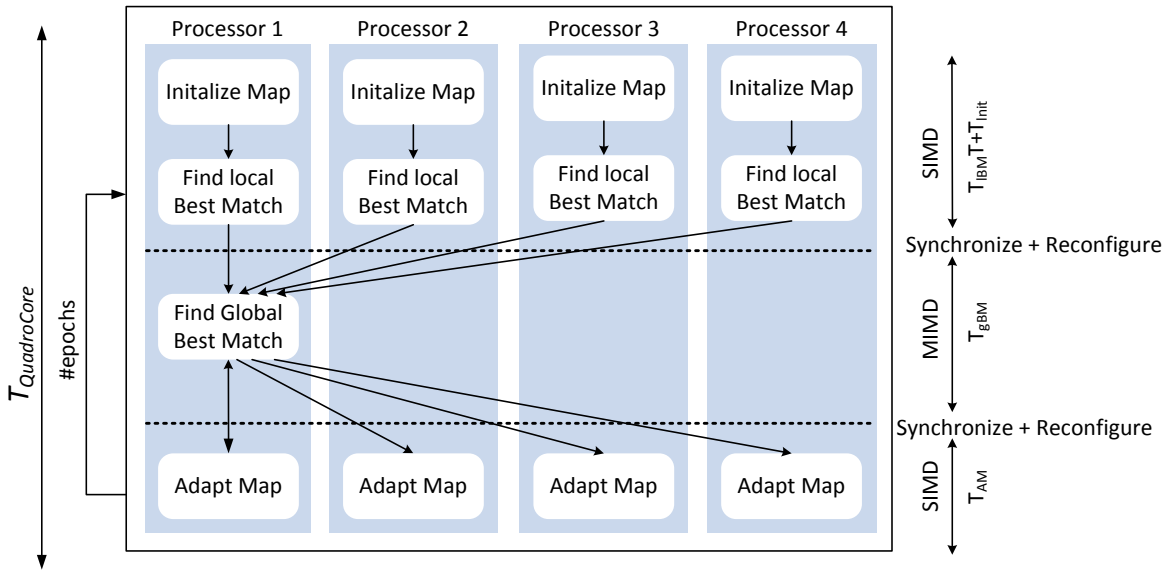


Figure 6.4: SOM Mapped to QuadroCore

mode. A switch to the SIMD mode from the default MIMD mode is done by inserting a reconfiguration instruction (function called `reconfig(mode)`). After the local best match function is executed, each processor sends its local best match data to the shared register file (function called `send(data, location)`). The steps between *find local best match* and *adapt Map* are data-dependent (a search operation), thus performed by one processor only (MIMD). Therefore, another reconfiguration instruction is inserted and the local best match data from all the processors is received from the shared register file `receive(data, location)` to calculate a global best match. For executing the adapt map function all processors again enter SIMD mode, as they share the same instruction stream. In general, finding the local best matches and adapting the map are the dominating functions as they are executed more often. They are executed on all the processors simultaneously in SIMD mode, resulting in significant power savings and an increase in code density.

Tables 6.4, 6.5, 6.6 compare the performance of the SOM application when mapped to a single processor, the QuadroCore operating in MIMD mode, and when operating in the low power SIMD mode (which includes reconfiguration overhead to/from MIMD mode). The speedup denotes the improvement in execution time on the reconfigurable multiprocessor as compared to the execution time on a single processor. Energy is calculated for the processor when operating at 200 MHz and is the product of the total time required (product of execution cycle and clock period) and the power consumed in the respective modes. The power measurement for a single processor was made for the entire QuadroCore, with one active processor. Hence, a difference in energy measurements for the QuadroCore with one active processor and the QuadroCore operating in MIMD mode are noticed on account of the power contribution of the

unused resources. This additional power includes the leakage power and internal power for the shared register file, the wishbone bus, and other resources which are unused when only one processor is active. The leakage power is in the range of 1.5 mW. In the absence of power contribution from the unused resources, the energy measurements for a single processor would be lower than that of the MIMD mode. Adding power gating capabilities to the architecture would result in zero leakage power contribution from the unused resources.

In the following experimental results, it is clear that the MIMD mode of operation is optimal in terms of time and the SIMD mode of operation ensures low power consumption. However, the power savings in the SIMD mode in comparison to the MIMD mode has an additional advantage. The significant power savings in the SIMD mode also results in energy savings, which is in the range of 28.44% to 30.9% in the following tables. It has to be noted that energy savings are observed even in the absence of a 4-fold speedup in our four processor QuadroCore operating in SIMD mode, which is a distinct advantage. As discussed in Section 3.5, this new scheme of reconfiguration ensures power savings and performance speedup for every additional processor. These experiments are a proof of concept for our scheme of architectural modifications to ensure energy savings for applications mapped on to the QuadroCore multiprocessor.

Table 6.4: Parameters: Vectors = 25 , Epoch = 1, Components = 5, Neurons = 16

Operating Mode	**Execution Cycles**	**Speedup**	**Power** (W)	**Energy** (μJ)
Single Processor	268,327	1	0.015	20.90
MIMD	**77,267**	**3.47**	0.052	20.20
SIMD−MIMD	78,250	3.43	**0.043**	**17.04**

Table 6.5: Parameters: Vectors = 5 , Epoch = 1, Components = 10, Neurons = 16

Operating Mode	**Execution Cycles**	**Speedup**	**Power** (W)	**Energy** (μJ)
Single Processor	100,719	1	0.016	8.11
MIMD	**28,581**	**3.52**	0.055	7.83
SIMD−MIMD	28,744	3.51	**0.045**	**6.47**

Table 6.6: Parameters: Vectors = 5, Epoch = 1, Components = 5, Neurons = 16

Operating Mode	**Execution Cycles**	**Speedup**	**Power** (W)	**Energy** (μJ)
Single Processor	53,870	1	0.015	4.14
MIMD	**15,608**	**3.45**	0.052	4.05
SIMD−MIMD	15,790	3.41	**0.044**	**3.50**

Variation in Application Parameters

The impact of variations in parameters such as the number of vectors, components and number of epochs was measured. For the test cases, the achieved speedup was in the range of 3.2 to 3.5. For further analysis, they can be easily scaled to larger scenarios. An increase in the number of vector components is beneficial in terms of speedups and energy savings, since the ratio of synchronisation overhead to parallel execution reduces with the number of vector components. Similarly, an increase in the number of *neurons* will result in speedup closer to 4 and make a corresponding impact on energy savings. As seen from the tabulated results, power savings in the range of 14% to 18% have been observed in the SIMD mode, when compared to the MIMD mode. The corresponding energy savings were in the range of 30% in comparison to a single processor. As a consequence of switching between SIMD−MIMD modes, the reduction in speed for SIMD operation is in the range of 0.2 to 2%. Apart from power and energy savings, the code size in SIMD mode was also reduced by about 8%, due to reduction in code size for the slave processors.

Scalability of Data Parallel Application

For the SOM algorithm, the parallelisable portion corresponds to the algorithm for the computation of the best match and the adapt map functions. However, the overhead of partitioning includes a serial portion, which is encountered on account of the division of the search process into local best match and global best match functions. Hence, the execution time obtained on the QuadroCore on account of the application partitioning can be expressed as:

$$T_{QuadroCore} = T_{init} + (T_{gBM} + \frac{T_{lBM} + T_{AM}}{PE}) \cdot epochs \tag{6.9}$$

where, T_{init} is the initialisation time for the maps, T_{lBM} is the time required to the local best match (function call *find best match*) among the neurons in the local memories on each of the processors. T_{gBM} is the time required to compute the global best match on one of the processors, after the individual local best matches have been located. And, T_{AM} is the time required to adapt the map (function call *adapt Map*) on each of the processors, based on the global best match shared among all the processors. PE is the number of processors that are simultaneously operating on individual maps. By increasing the number of processors PE, the time required to compute the local best match (T_{lBM}) and the adapt map (T_{AM}) reduces proportionally ($T_{lBM} \propto \frac{1}{PE}$ and $T_{AM} \propto \frac{1}{PE}$). However, T_{gBM} is the additional overhead involved in partitioning. This time depends on the synchronisation time between the participating processors, which is proportional to the number of participating processors ($T_{gBM} \propto PE$). However,

the computation time involved in locating the local best time and the adapt map functions is much larger than the time required to find the global best match ($T_{lBM} \gg T_{gBM}$). Thus, the overhead of partitioning is minimal compared to the advantage of partitioning the application onto the four processors on QuadroCore. In addition, the reduction in the memory accesses on account of operation in the SIMD mode leads to reduction in power. Consequently, it leads to energy savings due to application partitioning.

6.2.5 Comparison: Parallelism, Speedup, Energy

Table 6.7 summarises the variations in costs and benefits for the above discussed applications on QuadroCore. DSP benchmarks (such as convolution, FFT) showed a performance speedup of up to 11 on account of the high register need. This corresponds to energy savings of 60% compared to the multiprocessor without reconfiguration. A machine learning application (self-organising maps) with task and data-level parallelism, showed a speedup of 3.5 compared to a single processor. Additionally, energy savings of up to 31% are observed on account of reconfiguration. A multiplier used in Elliptic Curve Cryptography with task-level parallelism exhibits a similar speedup of 3 in comparison to a single processors and energy savings of 20% on account of reconfiguration in the QuadroCore multiprocessor. Figure 6.5 summarises the perfor-

Table 6.7: Application Performance on QuadroCore

Application	Parallelism	Max. Speedup	Energy Savings
DSP	Instruction-level	11	60%
SOM	Data-level	3.5	17-31%
ECC	Task/Function-level	3	20%

mance reports of the three applications in terms of speedup, power consumption and energy consumption. Additionally, it compares the achieved reports to the theoretical estimations (Ideal Scenario) according to Amdahl's law as discussed in Section 3.5. As can be seen, the speedups obtained are higher than the ideal case for DSP applications, where a superlinear speedup is seen. The power consumption is equal to or less than the ideal case, on account on the reconfiguration mechanism employed in the QuadroCore processor. Finally, energy consumption is equal to or less than the ideal case for the three diverse applications.

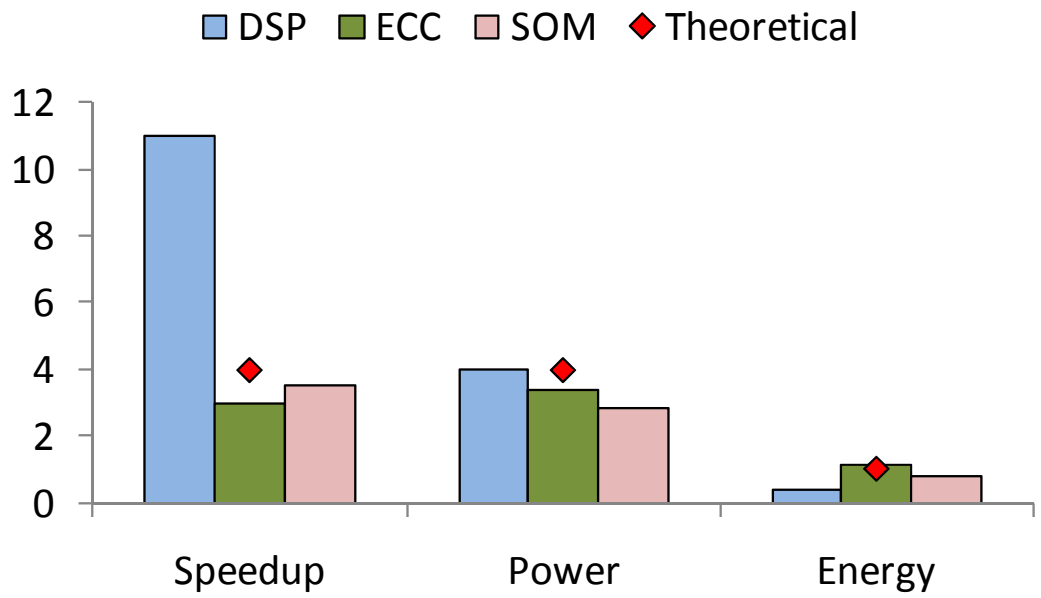


Figure 6.5: Comparing Performance with Theoretical Estimations

6.2.6 Comparable Architectures

Figure 6.6 compares the performance in cycles, area, and energy for SOM algorithm mapped onto a set of architectures. The plot shows the variations for a single NCore processor, the QuadroCore operating in MIMD mode, SIMD mode, Tensilica's base processor (1K Instruction cache and 1K Data cache), and the same Xtensa base processor enhanced with instruction set extensions. For comparison, all the cores were analysed at 90nm target technology with an operating frequency of 200 MHz. The size of the points corresponds to a third dimension, which is the gate-count for the processor core (excluding the memory), in each of the cases.

As seen in the plot, the energy consumption of the QuadroCore multiprocessor on account of the parallelism is lower in comparison to a single Xtensa processor. Although QuadroCore outperforms the others in terms of clock cycles, it has been noted that a single QuadroCore has four independent datapaths as compared to a single control and datapath in Xtensa processors. Multiple control and data-paths in QuadroCore provide an advantage in terms of speed for the data-parallel SOM algorithm. Further, the SIMD mode of operation also results in additional energy savings. It may be noted that increasing the number of Xtensa processors will also result in an improved performance (cycles). However, increasing the number of processor cores will result in higher power consumption, and require much larger area. The presented method can be incorporated along with instruction set extensions as a solution to address performance trade-offs such as area, power, and time. For comparison, a custom realisation

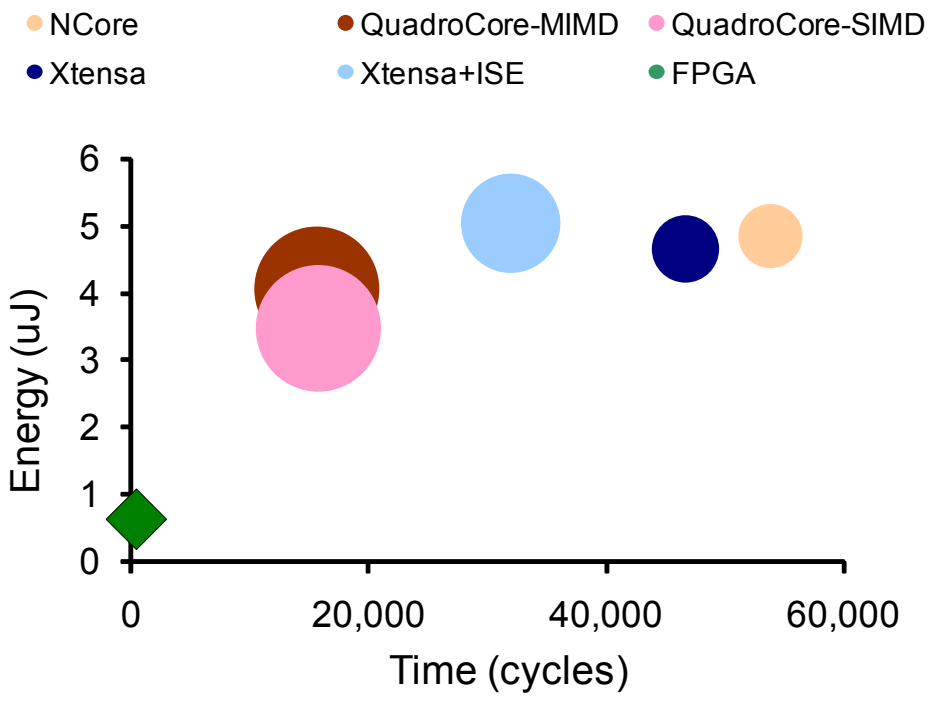


Figure 6.6: Comparing Area, Energy, and Execution time

of the SOM algorithm based on [102], was implemented on Xilinx XCV4LX100, operating at 150 MHz (indicated as a green quadrilateral). This implementation achieves a significant improvement in time and energy. However, the inherent fine-grained configurability in FPGAs results in a high area requirement, which is excluded in Figure 6.6.

Figure 6.7 compares the same set of architectures in terms of area, clock cycles of execution and power consumption. As can be seen, the performance of the QuadroCore multiprocessor in both the reconfigurable operating modes is comparable in power consumption to other similar state-of-the-art RISC processors. However, the power consumption of FPGAs exceeds the power consumption of the remaining architecture by orders of magnitude. An important point of comparison between the reconfigurable QuadroCore and the configurable Xtensa processors is the capability of run-time adaptability in QuadroCore in comparison to design time configurability (via additional instructions) in Tensilica's processors. In addition, the design time for QuadroCore is much shorter, since the design entry and reconfiguration management is made possible using C-based application description. The presented architecture and methodology is a proof of concept for reusing existing processors (NCore processors here) to build clusters of multiprocessors that can be customised to application requirements during run-time. The changes introduced ensure that the pipeline stages and the original processor ISA are unaltered.

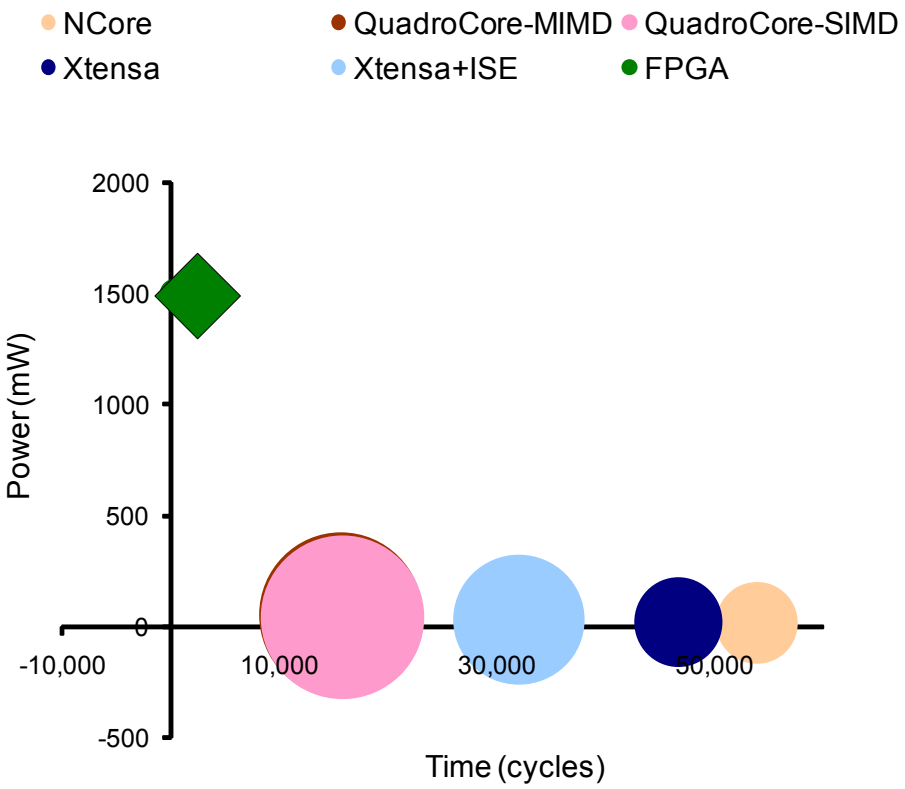


Figure 6.7: Comparing Area, Power, and Execution time

6.3 Extending the QuadroCore Multiprocessor

Architectural adaptability in the QuadroCore is a significant advantage that extends the usability of the multiprocessor, beyond application acceleration. To explore these features, the QuadroCore multiprocessor was mapped onto our FPGA-based prototyping environment. The benefit of mapping this multiprocessor environment onto our rapid prototyping environment is two-fold. Firstly, it provides and environment to validate parallel programs and manage application partitioning. Next, the flexible nature of the QuadroCore multiprocessor aids in introducing architectural changes based on the run-time statistics obtained using the prototyping framework. Together, the combined hardware-software design environment provides a platform for an accelerated application and architectural validation.

6.3.1 Platform for Validating Parallel Programs

The QuadroCore multiprocessor organisation is adaptable and scalable to suit application needs. With an operating frequency of about 200 MHz, it provides a framework for validating parallel programs, where the design entry is based on widely used se-

quential programming models, such as C. FPGA-based prototyping environments such as FAST [103, 1] have been deployed to accelerate simulation speeds. The programming model FPGAs is a significant drawback to this approach, since it only relies on HDLs design entry for application mapping. Our approach in contrast, expresses parallelism application definition using legacy sequential programming languages such as C, as described in Chapter 5. The necessary directives to support parallel programming initiatives are methods of expressing synchronisation, communication, and task-level parallelism. These paradigms are supported in the compilation framework used for application mapping in the QuadroCore multiprocessor. Typically, the main concern in performance estimations for large applications is the large execution time in simulation-based environments. To circumvent this drawback, As shown in Figure 6.8, the entire architecture has been mapped to our scalable rapid prototyping system, RAPTOR2000 [104]. This prototyping environment is supported with a user-interface and monitoring framework to obtain run-time statistics [PPP09]. This accelerated feedback obtained in real-time, enables faster validation, and enables modification that can be incorporated in the design phase. Thus, partitioning applications and load balancing based on input data are now possible via feedback parameters obtained via the prototyping environment. Based on the instruction-level power model described in Section 5.3 a feedback on the application-level power characteristics is also generated. As the application definition is made using high-level programming languages such as C, mapping existing benchmarks and applications is straightforward. An additional advantage of the FPGA-based prototyping environment is the nature of its scalability. The entire multiprocessor organisation is scalable to include many such processing clusters. The number of processors and the inter-connectivity can be scaled as in the GigaNetIC architecture [98], which connects multiple such clusters via a network-on-chip. In addition, the prototyping environment is scalable with multiple add-on daughter boards and fast inter-board communication. Thus, this platform for accelerated prototyping facilitates convenient cycle accurate timing estimation and power estimation for large computationally intensive applications.

6.3.2 Environment for Run-time Processor Customisation

Introducing processor customisation is a standard procedure for of application-specific adaptability. Configurable processors such as Xtensa [11] as examples of design time customisable processors. The QuadroCore reconfigurable multiprocessor provides capabilities to alter its control and data-flow using high-level constructs as discussed in Section 5.1. These changes to the processor architecture are introduced during run-time. With these features, the QuadroCore multiprocessor can be customised to suit applications with instruction set extensions introduced during run-time. For example, the need for an additional ALU or a register file is achieved by reconfiguring the control

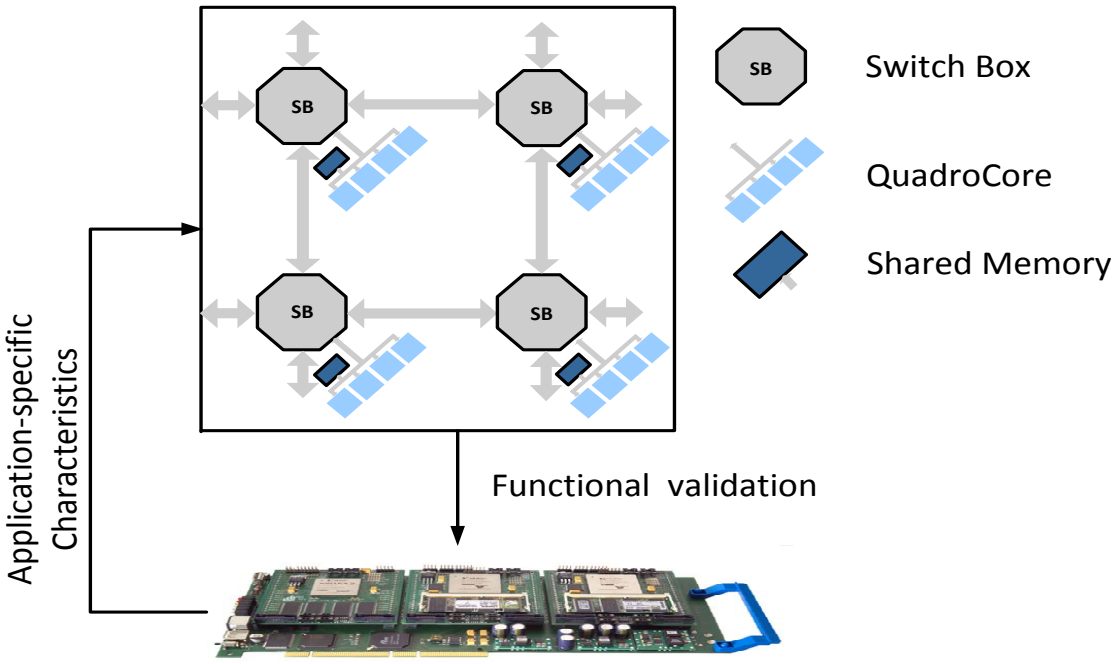


Figure 6.8: Scalable Architecture and Prototyping Environment

and datapath between the execute and register read/write stages. Therefore, the configurable features typically introduced during design time are now introduced during run-time via performance statistics obtained in conjunction with rapid prototyping environment. The design framework now has configurable features as functional units per processor, processors per cluster, and clusters per multi-core architecture. Each of these features is customisable during run-time to suit variations in applications, architecture, and the collective influence of the two domains.

6.4 Summary

This chapter focuses on analysing the impact of application-specific characteristics on the performance of the QuadroCore multiprocessor. As a proof of concept, the performance impact of mapping three diverse applications onto the QuadroCore multiprocessor have been analysed. Firstly, a set of micro-benchmarks were mapped to the QuadroCore using the CoBRA compiler, where speedup of up to 11 was observed in comparison to a single processor implementation. Next, it has been established theoretically that for DSP applications, the advantage achieved in terms of power is in the range of 1.2 to 3.5 times, depending on algorithm-specific characteristics. For a function-parallel multiplier used in Elliptic Curve Cryptography, a power savings of 23% and a corresponding energy savings of 15.5% using the presented reconfiguration

mechanism have been achieved on the QuadroCore multiprocessor. The reconfiguration mechanism yields power savings in the range of 14-18% and energy savings of 28-31% for a data-parallel application called Self-organising Maps. The diversity of application characteristics and the consistent performance advantage confirms the usability of the QuadroCore multiprocessor over a wide range of applications. A comparison of the results obtained for these applications with the theoretical estimations typically employed in multiprocessors is made. It can be seen that the energy results obtained using QuadroCore are the lowest in all the cases. Finally, a comparison in terms of performance is also made with a state-of-the-art configurable processor Xtensa from Tensilica. Overall, the average power and energy consumption using the QuadroCore multiprocessor is lower compared to the other processors.

In addition to resource efficiency and application acceleration, the QuadroCore multiprocessor has been mapped onto our FPGA-based prototyping environment. This helps in using this environment for accelerated validation of parallel programs. Further, the flexibility in the QuadroCore multiprocessor has been used to introduce processor customisations during run-time. The uniqueness is that it ensures combined validation of both the application and the architecture. The scalable nature of the prototyping environment and the reconfigurable multiprocessor together provides a mechanism for validation and early estimations for time, power, and area.

Chapter 7

Conclusions and Future Work

Architectural comparisons consider diverse performance factors such as time, power, and energy. In addition to architectural merits, the application's characteristics and its computational demands define the type of architecture deployed. Thus, identifying application-level properties is necessary in steering the overall performance. Diminishing returns in spite of higher transistor densities and operating frequencies emphasise the need for other alternatives of performance enhancements. Standard approaches use single processors, which are expected to operate at high frequencies in order to meet performance demands. Alternatively, using multiple processors provides the possibilities for application partitioning and task distribution over multiple processing units each with lower operating frequencies. In this case, application-level characteristics and granularity of application partitioning are of concern for high performance. Another important objective for application redesign is the need for energy-efficiency. Present day architectures have primarily relied on high-performance, with power and energy costs as the inevitable side-effects. Apart from these performance characteristics, the design, verification, and validation time of an architecture makes a significant impact on the time-to-market, which directly contributes to the market success.

Recently significant efforts have been directed towards multi-core design, where parallel application description and portability to other existing legacy code are dominant challenges. On one hand, it is often stated that designing applications for parallel computing and multi-core architectures is a whole new research dimension. Nevertheless, methods of parallel application description have long existed in HDL-based designs, for close to 30 years. Similarly, in the case of processor architectures, newer generations of processor architectures have been developed with the primary task of being binary compatible to its previous generations. Code portability and binary compatibility have been the focus in designing new processors.

In lieu of the range of questions that have appeared in the multi-core design domain, this thesis has presented an approach to the design of a reconfigurable multiprocessor architecture. It presents a novel perspective on application description, and addresses application-to-architectural mapping with the focus on introducing power as a primary criterion in the design flow. Additionally, our objective has been on minimising hardware re-spins by introducing a run-time mechanism to fine-tune architectural features to match application characteristics. Methods of designing architectures, applications, and the process of mapping applications to architectures in embedded computing domain have been presented in this thesis.

7.1 Summary

The focus of this thesis has been to develop a reusable multiprocessor template that benefits from the advantages in multiprocessors and reconfigurable architectures. In addition, it addresses application description and mapping by applying diverse performance objectives, viz., time, power, and energy.

In Chapter 2, the architectural aspects in designing multi-core processors have been presented. This stage is governed by mutually opposing design decisions, such time versus area, or time versus power etc. These decisions necessitate design space explorations. Architectural explorations often use models to aid fast and early performance estimations. However, accuracy of these models and the time required for explorations are mutually opposing decisions. Typically, performance measurements are accurate when made towards the end of the design cycle. However, the need for a quick performance feedback to the architecture under design necessitates the introduction of early performance estimates. These opposing objectives have lead to introducing high-level models to speedup design space exploration. In the same chapter an overview of existing methods in defining architectural models and the extent to which each of the models contribute to decision making in architectural considerations are described. Architectural design decisions are also greatly influenced by the amount of flexibility introduced. To highlight the impact, the degree of flexibility and the variations observed on performance characteristics are discussed. The diversity in architectural features necessitates exploration, which is based on merits defined to identify the correct architecture. Based on these features, a classification on existing multi-core architectures and their pros & cons are discussed. A similar study is made for reconfigurable architectures. Learning from the benefits and drawbacks that exist in both these kinds of architectures, a run-time reconfigurable multiprocessor template has been developed. Our QuadroCore multiprocessor is designed and implemented based on this design philosophy. It merges two successful architectural techniques,

viz., reconfigurability and parallelism. Finally, the advantages of this scheme have been summarised.

Performance of a processor depends to a large extent on the type of application that it is used for. Typically, time is the primary focus of application mapping. Performance measurements such as power and energy have been widely addressed as hardware and circuit design issues, where power is predominantly recorded after an application is mapped. However, the application that runs on a given processor or hardware is entirely responsible for the power consumption by the device. With these diverse design issues, Chapter 3 classifies applications with a focus on methods of description and characteristics to support parallel programming. Independent of the type of application description is the cost of application partitioning and its impact on computation, communication, and synchronisation demands. Three diverse applications, viz., DSP algorithms, multiplier used in Elliptic Curve Cryptography, and a data-parallel machine learning algorithm — Self-organising Maps, have been characterised based on these attributes. Increasingly, parallelism has been used as an approach for performance improvement. Hence, we extend the classical approach described in Amdahl's law to express parallelism as a function of time, to include parameters such power and energy. This approach provides a feedback on the impact on the granularity of parallelism, independent of the architecture it is mapped on. This in turn, affects performance parameters, viz., time, power, area, and energy consumption.

Mapping addresses the issue of matching application-level features and architectural attributes. In Chapter 4 the diversities in application to architectural features were discussed. When the application's features entirely coincide with architectural attributes, it results in a perfect solution. However, a perfect solution for every application under consideration introduces a large time-to-market. The choice of a perfect solution versus a timely solution is made with performance characteristics such as time, area, power, which provide a direct feedback to steer the mapping algorithms. In this chapter, two mapping schemes, viz., compilation and FPGA synthesis were detailed, representing temporal and spatial mapping strategies respectively. A scheme that merges both these paradigms, in order to gain from the advantages in both the approaches, is presented. Additionally, a scheme of adaptive mapping via run-time reconfiguration is presented. For the reconfigurable multiprocessor template in Chapter 2, an adaptive design flow that uses reconfiguration as a method of customising the architecture has been presented. As power begins to emerge as an important performance factor, mapping is addressed as a multi-dimensional optimisation problem with time, power, and energy as the objectives.

In Chapter 5 the design and implementation details of the QuadroCore multiprocessor were presented. To observe the diversity in spatial and temporal computation, a proof of concept for the multiprocessor framework was developed. Using a collection of

four 32-bit processors, reconfiguration was added as a feature for application-specific adaptability. Application-specific adaptability addresses the computational complexity, communication model, and the method of synchronisation via reconfiguration. The design was implemented on UMC's 90 nm standard cell technology. Using the synthesised netlist, gate-level simulations were conducted to observe the impact of application-specific characteristics on performance characteristics such as power, and time, consequently energy. A power-model was formulated at instruction-level for the processor architecture. This generic power model is integrated in the application-mapping design flow to enable time and power aware application mapping.

With the validation of the presented technique via a prototype implementation, we were encouraged to look at another dimension where the architecture can be employed. In Chapter 6, we have presented three applications where this architecture provides a resource efficient application mapping in comparison to other existing architectures. In addition, the flexibility in QuadroCore is a feature that facilitates the use of the multiprocessor as a platform for run-time processor customisation and in a prototyping environment for validating parallel programs.

7.2 Future Work

The QuadroCore design philosophy successfully addresses the scheme of introducing reconfiguration as a mechanism for run-time application adaptability. This scheme of application description, architectural modification, and mapping can be extended as follows:

Reliability via Reconfiguration

The fast single cycle reconfiguration mechanism can be used to tackle variations in device performance and reliability. The quick reconfiguration scheme acts as a quick recovery mechanism to adapt to variations encountered during device deployment. The uniform fabric of processors aids in easy task migration and resource reuse, which together assist in recovering from faults identified during operation.

Asymmetric Data paths

This mode of operation is an extension to the existing modes of operation in the QuadroCore. It uses the two reconfigurable interconnects, viz., between the instruction fetch and decode stage, and registers access and execute stages. In this mode of operation, the non-functional parts of the processors are replaced by the corresponding

functional building blocks borrowed from the nearest neighbour, by actively reconfiguring the interconnections. This is feasible since all the processors are identical and the resources of the neighbouring processors are easily accessible. Figure 7.1 shows the reconstructed control and data path using the fault-free resources of two adjacent processors.

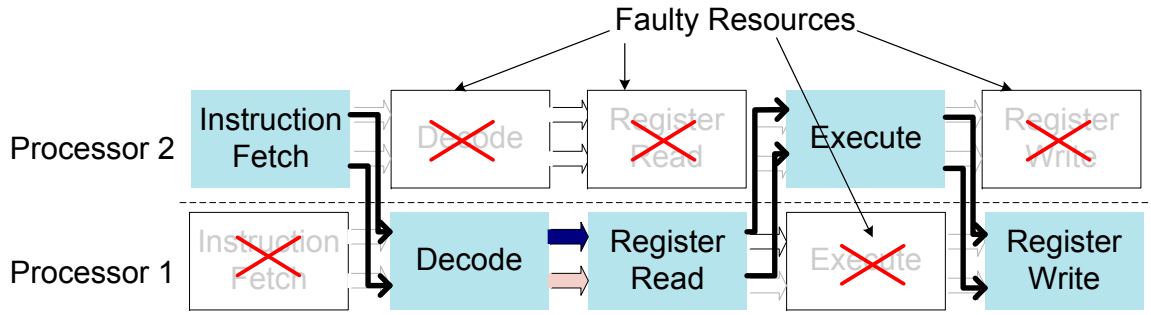


Figure 7.1: Reusing Resources to Restore Normal Operation

Dynamic Voltage and Frequency Scaling

To introduce power as a criterion for application mapping, this thesis characterises instructions in terms of power consumption, which is in addition to characterisation in terms of time. An immediate extension to this approach is to include frequency and voltage characteristics during instruction selection via DVFS. Using voltage and frequency as an application-mapping criterion provides a fine-grained control over timing and power, for application-specific modifications that can be introduced during run-time.

Field Programmable Multiprocessor Arrays

QuadroCore is a first step towards using reconfiguration for application-specific run-time adaptability and can be extended to include various flavours of reconfiguration. With next generation FPGA architecture primarily composed of a sea of processors, programming via high-level programming languages simplifies customisations. The added benefit of such architectures is the reduction in power consumption, a significant performance deviation, and the possibility to explore parallelism only by the amount of parallelisable portions as described by Amdahl's law. A field programmable multiprocessor (FPMA) array provides a platform for exploring both sequential and parallel processing paradigms with a communicating-sequential processing model.

The flexibility in FPMAs enables modifying the architecture to incorporate a parallel pipelined model to enable streaming operations. The ease of modification transforms

the 'von Neumann' model of computation to a systolic array-like interconnected architecture.

Scheduling for Peak Power Reduction

Peak power is a cumulative affect of application characteristics recorded for every clock cycle of execution. Since the choice of the instruction is power-aware, scheduling introduces the additional benefit of maintaining a peak power envelope such that an overshoot of the power margin is circumvented. During compile time, it ensures scheduling of instructions to even out power consumption. Additionally during run-time, the same principle ensures rescheduling to benefit from power savings to extend battery lifetimes.

Adaptive Load-balancing

Merging the features of DVFS and adaptive scheduling provides a platform to design real-time monitoring environment, with load balancing to adapt according to variations in power, temperature, and operating frequency. The fast reconfiguration mechanism enables adaptability, while making a small impact on the overall speed of operation. Effects such as temperature overshoots recorded during run-time can then be tackled via task migration, frequency adaptation, or adaptive scheduling mechanisms.

Glossary

C_{comm} . . . . . Clock cycles required for each communication operation

C_{comp} . . . . . Clock cycles required for each computation operation

C_{sync} . . . . . Clock cycles required for each synchronisation operation

E_1 . . . . . Energy consumption for a single processor

E_N . . . . . Total energy consumption for N identical processors

E_{budget} . . . . . Energy budget for an application

E_{total} . . . . . Total energy consumption

N_{comm} . . . . . Total number of communication operations

N_{comp} . . . . . Total number of compute operations

N_{dmem} . . . . . Number of accesses to data memory

N_{imem} . . . . . Number of accesses to instruction memory

N_{reg} . . . . . Number of accesses to register file

N_{sync} . . . . . Number of synchronisation operations

$N_{toggles}$ . . . . . Total toggle count for a given instruction

P_1 . . . . . Power consumption for a single processor

$P_{QuadroCore}$ . . . Power consumption on QuadroCore

P_{alu} . . . . . Power consumption of the ALU

P_{budget} . . . . . Power budget for an application

P_{comm} . . . . . Power consumption of the communication operations

P_{comp} . . . . . Power consumption of the computation operation

P_{dec} . . . . . Power consumption of the decoder

P_{dmem} . . . . . Power consumption of the data memory

P_{exec} . . . . . Power consumption of the execution unit

P_{imem} . . . . . Power consumption of the instruction memory

P_{instr} . . . . . Power consumed in an instruction cycle

P_{mimd} . . . . . Power consumption in MIMD mode

P_{par} . . . . . Power consumed by the parallelisable portion of the application

P_{recon} . . . . . Power consumption on account of reconfiguration

$P_{reg-read}$ . . . . Power consumption for register read operation

$P_{reg-write}$ . . . . Power consumption for register write operation

P_{reg} . . . . . Power consumption of the register file

P_{seq} . . . . . Power consumed by the sequential portion of the application

P_{simd} . . . . . Power consumption in SIMD mode

P_{sync} . . . . . Power consumption of the synchronisation operations

$P_{total-comm}$ . . . Total power consumption of the communication operations

$P_{total-comp}$ . . . Total power consumption of the computation operation

$P_{total-sync}$ . . . . Total power consumption for all the synchronisation operations

P_{total} . . . . . Total power consumption

S_{dec} . . . . . Switching activity at the decoder

S_{dmem} . . . . . Switching activity at data memory

S_{exec} . . . . . Switching activity at execution unit

S_{imem} . . . . . Switching activity at instruction memory

S_{mem} . . . . . Total switching activity at data memory and instruction memory

S_{rate} . . . . . Rate of switching activity

S_{reg} . . . . . Switching activity at register file

T_{AM} . . . . . Time required for executing the adapt map function

$T_{QuadroCore}$ . . . Execution time on QuadroCore

T_{budget} . . . . . Timing budget for an application

T_{comm} . . . . . Total time for all communication operations

T_{comp} . . . . . Total time for all compute operations

T_{gBM} . . . . . Time required for locating the global best match

T_{init} . . . . . Time required for initialising the maps

T_{instr} . . . . . Instruction length

T_{lBM} . . . . . Time required for locating the local best match

T_{par} . . . . . Parallelisable portion of the total execution time

T_{recon} . . . . . Reconfiguration time

T_{seq} . . . . . Sequential portion of the total execution time

T_{sync} . . . . . Total time for all synchronisation operations

T_{total} . . . . . Total execution time

f_{comm} . . . . . Operating frequency for each communication operations

f_{comp} . . . . . Operating frequency of the computation operation

f_{sync} . . . . . Operating frequency for each synchronisation operation

p . . . . . Percentage parallelisable portion of the total execution time

ADL . . . . . Architectural Description Language

ASIC . . . . . Application-specific Integrated Circuit

CLB . . . . . Configurable Logic Blocks

CMOS . . . . . Complementary Metal Oxide Semiconductor

CMP . . . . . Chip Multiprocessor

CoBRA . . . . . Compiler Driven Dynamic Reconfiguration of Architectural Variants

CPU . . . . . Central Processing Unit

CSP . . . . . Communicating Sequential Processes

DCA . . . . . Direct Cache Access

DLP . . . . . Data-level Parallelism

DSP . . . . . Digital Signal Processing

DVFS . . . . . Dynamic Voltage and Frequency Scaling

ECC . . . . . Elliptic Curve Cryptography

EIB . . . . . Element Interconnect Bus

FFT . . . . . Fast Fourier Transform

FPGA . . . . . Field Programmable Gate Arrays

FPMA . . . . . Field Programmable Multiprocessor Array

GPGPU . . . . General Purpose Computing on Graphic Processing Units

GPU . . . . . Graphic Processing Units

HDL . . . . . Hardware Description Language

ILP . . . . . Instruction-level Parallelism

IP . . . . . Intellectual Property

LUT . . . . . Lookup Table

MIMD . . . . . Multiple Instruction Multiple Data

MPI . . . . . Message Passing Interface

NOP . . . . . No Operation

QPI . . . . . Quick Path Interconnect

RISC . . . . . Reduced Instruction Set Computer

RTL . . . . . Register Transistor Level

SIMD . . . . . Single Instruction Multiple Data

SOM . . . . . Self-organising Maps

TLP . . . . . Task-level Parallelism

UPSLA . . . . . Universal Processor Specification Language

VCD . . . . . Value Change Dump

VLIW . . . . . Very Long Instruction Word

List of Figures

List of Tables

References

[1] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai, "A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays.* New York, NY, USA: ACM, 2008, pp. 77–86.

[2] M. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 23–34, April 2007.

[3] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara, "From vhdl register transfer level to systemc transaction level modeling: A comparative case study," in *SBCCI '03: Proceedings of the 16th symposium on Integrated circuits and systems design.* Washington, DC, USA: IEEE Computer Society, 2003, p. 355.

[4] O. Bonorden, N. Bruels, D. K. Le, U. Kastens, F. Meyer auf der Heide, J.-C. Niemann, M. Porrmann, U. Rueckert, A. Slowik, and M. Thies, "A holistic methodology for network processor design," in *Proceedings of the Workshop on High-Speed Local Networks held in conjunction with the 28th Annual IEEE Conference on Local Computer Networks (LCN2003)*, Oct. 2003, pp. 583–592.

[5] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with Lisa.* Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[6] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *European Design and Test Conference, 1995. ED and TC 1995, Proceedings.*, Mar 1995, pp. 503–507.

[7] P. Mishra and N. Dutt, "Architecture Description Languages for Programmable Embedded Systems," in *In IEE Proceedings on Computers and Digital Techniques*, 2005, pp. 285–297.

[8] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine stream processor," in *Proceedings 2002 IEEE International Conference on Computer Design*, Sep. 2002, pp. 282–288.

[9] *BCM 1480 : Product Brief*, Broadcom Corporation, May 2006, available from http://www.broadcom.com/collateral/pb/1480-PB04-R.pdf.

[10] N. R. Potlapally, S. Ravi, A. Raghunathan, R. B. Lee, and N. K. Jha, "Impact of Configurability and Extensibility on IPSec Protocol Execution on Embedded Processors," in *VLSID-06: Proceedings of the 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 299–304.

[11] R. E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[12] *ARC Configurable CPU/DSP Cores*, ARC Inc., Aug. 2008, available from http://www.arc.com.

[13] *Xtensa® LX2 Microprocessor Data Book*, Tensilica, Inc., 2008.

[14] *XLS™ Processor Family*, RMI Corporation, available from http://http://www.rmicorp.com/products/xls.htm.

[15] *Product Brief: OCTEON Plus CN58XX 4 to 16-Core MIPS64-Based SoCs*, Cavium Networks, 2008, available from http://www.caviumnetworks.com.

[16] *RM9224 Integrated Multiprocessor: Product Brief*, PMC-Sierra Inc., available from http://www.pmc-sierra.com/products/details/rm9224/.

[17] *NVIDIA Tesla C1060 Datasheet*, NVIDIA Corporation, June 2008, available from http://www.nvidia.com/object/product_tesla_c1060_us.html.

[18] *ATI Radeon™ HD 4800 Series Overview*, Advanced Micro Devices, Inc., June 2009, available from http://ati.amd.com/products/radeonhd4800/index.html.

[19] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[20] *Intel® Core™ i7-920 Processor*, Intel Corporation, June 2009, available from http://www.intel.com/support/processors/corei7/.

[21] *Key Architectural Features of AMD Phenom™ X4 Quad-Core Processors*, Advanced Micro Devices, Inc., June 2009, available from http://www.amd.com/us/products/desktop/processors/phenom/Pages/AMD-phenom-processor-X4-features.aspx.

[22] *The Industry-Changing Impact of Accelerated Computing*, Advanced Micro Devices, Inc., 2008, available from http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf.

[23] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak,

M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation CELL processor," in *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, 2005, pp. 184–592 Vol. 1.

[24] T. W. Ainsworth and T. M. Pinkston, "On Characterizing Performance of the Cell Broadband Engine Element Interconnect Bus," in *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 18–29.

[25] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: Design alternative for cache on-chip memory in embedded systems," in *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign.* New York, NY, USA: ACM, 2002, pp. 73–78.

[26] *UltraSPARC T1 Processor*, Sun Microsystems, Inc., June 2009, available from http://www.sun.com/processors/UltraSPARC-T1/index.xml.

[27] *Product Datasheet : MSC8144 Quad Core Digital Signal Processor*, Freescale Semiconductor Inc., 2009, available from http://www.freescale.com.

[28] *Product Datasheet : TMS320C6474 Multicore Digital Signal Processor*, Texas Instruments, 2009, available from http://focus.ti.com/docs/prod/folders/print/tms320c6474.html.

[29] *Datasheet : Blackfin Embedded Symmetric Multiprocessor ADSP-BF561*, Analog Devices, Inc., 2007, available from http://www.analog.com/static/imported-files/data_sheets/ADSP-BF561.pdf.

[30] G. Burns, M. Jacobs, M. Lindwer, and B. Vandewiele, *Silicon Hive's Scalable and Modular Architecture Template for High-Performance Multi-Core Systems*, 2006.

[31] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture.* Washington, DC, USA: IEEE Computer Society, 2004, p. 2.

[32] M. Butts, "Synchronization through Communication in a Massively Parallel Processor Array," *IEEE Micro*, vol. 27, no. 5, pp. 32–40, Sept.-Oct. 2007.

[33] *PC102 Product Brief*, PicoChip Inc, March 2004, available from http://www.picochip.com/.

[34] *Rapport' KC256, Technical Overview*, Rapport Inc., 2008, available from http://www.rapportincorporated.com.

[35] K. Sankaralingam *et al.*, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," *IEEE Micro*, vol. 23, no. 6, pp. 46–51, 2003.

[36] F. X. Guichang Zhong and J. Alan N Willson, "A Power-Scalable Reconfigurable FFT/IFFT IC Based on a Multi-Processor Ring," in *IEEE Journal Of Solid-State Circuits, Vol. 41, No. 2, February 2006.* Washington, DC, USA: IEEE, 2006, pp. 483–495.

[37] M. Thuresson, M. Sjalander, M. Bjork, L. Svensson, P. Larsson-Edefors, and P. Stenstrom, "FlexCore: Utilizing Exposed Datapath Control for Efficient Computing," *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pp. 18–25, July 2007.

[38] M. Epalza, P. Ienne, and D. Mlynek, "Adding Limited Reconfigurability to Superscalar Processors," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques.* Washington, DC, USA: IEEE Computer Society, 2004, pp. 53–62.

[39] S. B. Gregory W. Donohoe, K. Joseph Hass and P.-S. Yeh, "A Reconfigurable Data Path Processor for Space Applications," in *Proc. Military and Aerospace Applications of Programmable Logic Devices, Laurel,MD, September 24-28*, 2000.

[40] M. S. Schlansker and A. B. Seong, "Processing cells for use in computing systems," in *US Patent*, Hewlett-Packard Development Company, 2004.

[41] R. J. Gove, K. Balmer, N. K. Ing-Simmons, and K. M. Guttag, "Reconfigurable multi-processor operating in SIMD mode with one processor fetching instructions for use by remaining processors," in *US Patent*, Texas Instruments Inc, 1996.

[42] J. Backus, "Can programming be liberated from the von Neumann style? : a functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[43] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, 1989.

[44] *Occam Reference Manual, SGS THOMSON Microelectronics Limited*, 1995.

[45] C. Whitby-Strevens, "The transputer," in *ISCA '85: Proceedings of the 12th annual international symposium on Computer architecture.* Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, pp. 292–300.

[46] *C-to-FPGA Solutions*, Impulse Accelerated Technologies, available from http://impulsec.com.

[47] *HandelC Language Reference Manual*, Embedded Solutions Limited.

[48] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE*

'04. Proceedings. Second ACM and IEEE International Conference on, June 2004, pp. 69–70.

[49] *Cynthesizer^{TM} The most productive path to silicon*, Forte Design Systems, 2008, available from http://www.forteds.com.

[50] *Cyberworkbench: System LSI Development Is Changing!*, NEC Sytem Technologies Ltd., 2005, available from http://www.nec.co.jp/.

[51] *Catapult C*, Mentor Graphics Inc., 2006, available from http://www.mentor.com/catapult.

[52] *Developing algorithmic designs using Bluespec*, Bluespec Inc., 2007, available from http://www.bluespec.com.

[53] NVIDIA, *NVIDIA CUDA Programming Guide 2.0.* NVIDIA, 2008.

[54] S. S. Huang, A. Hormati, D. Bacon, and R. Rabbah, "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary," in *In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Paphos, Cyprus*, 2008.

[55] *A Framework for Hardware-Software Co-Design of Embedded Systems*, available from http://embedded.eecs.berkeley.edu/Respep/Research/hsc/abstract.html.

[56] S. Singh, "A Demonstration of Co-Design and Co-Verification in a Synchronous Language," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe.* Washington, DC, USA: IEEE Computer Society, 2004, p. 21394.

[57] M. Snir and S. Otto, *MPI-The Complete Reference: The MPI Core.* Cambridge, MA, USA: MIT Press, 1998.

[58] M. Butts, A. M. Jones, and P. Wasson, "A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing," in *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 55–64.

[59] A. A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics Doklady*, vol. 7, pp. 595–596, 1963.

[60] T. Kohonen, *Self-organization and associative memory.* Springer-Verlag New York, Inc. New York, NY, USA, 1989.

[61] M. Porrmann, U. Witkowski, and U. Rueckert, "A massively parallel architecture for self-organizing feature maps," *Neural Networks, IEEE Transactions on*, vol. 14, no. 5, pp. 1110–1121, Sept. 2003.

[62] R. Dreesen, T. Jungeblut, M. Thies, M. Porrmann, U. Kastens, and U. Rueckert, "A synchronization method for register traces of pipelined processors," in *International Embedded Systems Symposium*, 2009, pp. 207–217.

[63] S. Cho and R. Melhem, "Corollaries to Amdahl's Law for Energy," *IEEE Comput. Archit. Lett.*, vol. 7, no. 1, pp. 25–28, 2008.

[64] D. H. Woo and H.-H. S. Lee, "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era," *Computer*, vol. 41, no. 12, pp. 24–31, 2008.

[65] B. Kienhuis, E. Deprettere, K. Vissers, and P. Van Der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pp. 338–349, Jul 1997.

[66] D. Luebke, M. Harris, J. Krueger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn, "GPGPU: General purpose computation on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes.* New York, NY, USA: ACM, 2004, p. 33.

[67] H. Zhong, S. A. Lieberman, and S. A. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 25–36.

[68] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. Tseng, "An overview of the SUIF compiler for scalable parallel machines," in *In Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, 1995, pp. 662–667.

[69] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002, *See http:// llvm.cs.uiuc.edu.*

[70] L. N. Chakrapani, J. Gyllenhaal, W. mei W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah, *Trimaran: An Infrastructure for Research in Instruction-Level Parallelism.* Springer Berlin / Heidelberg, 2005, ch. Languages and Compilers for High Performance Computing, pp. 32–41.

[71] B. F. Romanescu and D. J. Sorin, "Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques.* New York, NY, USA: ACM, 2008, pp. 43–51.

[72] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke, "The StageNet fabric for constructing resilient multicore systems," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, Nov. 2008, pp. 141–151.

[73] B. Mei, A. Lambrechts, D. Verkest, J.-Y. Mignolet, and R. Lauwereins, "Architecture Exploration for a Reconfigurable Architecture Template," *IEEE Des. Test*, vol. 22, no. 2, pp. 90–101, 2005.

[74] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[75] T. A. Proebsting, "BURS automata generation," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 3, pp. 461–486, 1995.

[76] P. J. Hatcher and T. W. Christopher, "High-quality code generation via bottom-up tree pattern matching," in *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1986, pp. 119–130.

[77] R. Ramesh and I. V. Ramakrishnan, "Nonlinear pattern matching in trees," *J. ACM*, vol. 39, no. 2, pp. 295–316, 1992.

[78] J. Ullman, "NP-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384 – 393, 1975.

[79] *C-to-Silicon Compiler*, Cadence Inc., 2009, available from http://www.cadence.com/products/sd/silicon_compiler/.

[80] S. D. Sahasrabuddhe, H. Raja, K. Arya, and M. P. Desai, "AHIR: A hardware intermediate representation for hardware generation from high-level programs," in *VLSI Design*, 2007, pp. 245–250.

[81] G. Haiyun and X. Juyan, "Research on the routing algorithm of SRAM-based FPGA," *Solid-State and Integrated Circuits Technology, 2004. Proceedings. 7th International Conference on*, vol. 3, pp. 1964–1966 vol.3, Oct. 2004.

[82] Árpád Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, "Survey of code-size reduction methods," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 223–267, 2003.

[83] P. Petrov and A. Orailoglu, "Compiler-Based Register Name Adjustment for Low-Power Embedded Processors," in *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*. Washington, DC, USA: IEEE Computer Society, 2003, p. 523.

[84] A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Instruction Scheduling for Low Power," *The Journal of VLSI Signal Processing*, vol. 37, May 2004.

[85] M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and minimization techniques for embedded dsp software," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 5, no. 1, pp. 123–135, Mar 1997.

[86] *Stretch : Software Configurable Processors*, Stretch Inc., http://www.stretchinc.com.

[87] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[88] M. Gruenewald, U. Kastens, D. K. Le, J.-C. Niemann, M. Porrmann, U. Rueckert, M. Thies, and A. Slowik, "Network Application Driven Instruction Set Extensions for Embedded Processing Clusters," in *PARELEC 2004, International Conference on Parallel Computing in Electrical Engineering, Dresden, Germany*, 7 - 10 Sep. 2004, pp. 209–214.

[89] M. Ito, T. Hattori, Y. Yoshida, K. Hayase, T. Hayashi, O. Nishii, Y. Yasu, A. Hasegawa, M. Takada, M. Ito, H. Mizuno, K. Uchiyama, T. Odaka, J. Shirako, M. Mase, K. Kimura, and H. Kasahara, "An 8640 MIPS SoC with Independent Power-Off Control of 8 CPUs and 8 RAMs by An Automatic Parallelizing Compiler," *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 90–598, Feb. 2008.

[90] S. S. Muchnik, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[91] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.

[92] M. Hussmann, "Compiler-Driven Dynamic Reconfiguration of Architectural Variants," Ph.D. dissertation, University of Paderborn, Apr. 2008.

[93] W. Ye, N. Vijaykrishnan, M. T. Kandemir, and M. J. Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool," in *Design Automation Conference*, 2000, pp. 340–345.

[94] P. Stanley-Marbell and M. Hsiao, "Fast, flexible, cycle-accurate energy estimation," in *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*. New York, NY, USA: ACM Press, 2001, pp. 141–146.

[95] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *ISCA*, 2000, pp. 83–94. [Online]. Available: citeseer.ist.psu.edu/brooks00wattch.html

[96] M. Kandemir, N. Vijaykrishnan, M. Irwin, and W. Ye, "Influence of compiler optimizations on system power," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 6, pp. 801–804, Dec 2001.

[97] *Synopsys PrimeTime PX*, Synopsys, Inc., available from http://www.synopsys.com.

[98] J.-C. Niemann, C. Puttmann, M. Porrmann, and U. Rueckert, "GigaNetIC - A Scalable Embedded On-Chip Multiprocessor Architecture for Network Applications," in *ARCS'06 Architecture of Computing Systems*, 13 - 16 Mar. 2006.

[99] H. Kalte, M. Porrmann, and U. Rückert, "A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs," in *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*, Hamburg, Germany, 2002.

[100] National Institute of Standards and Technology (NIST), *Digital Signature Standard (DSS)*. U.S. Department Of Commerce, 27 Jan. 2000, vol. FIPS 186-2, ch. Recommended elliptic curves for federal government use, pp. 24–48.

[101] C. K. Koc and T. Acar, "Montgomery multiplication in GF(2^k)," *Des. Codes Cryptography*, vol. 14, no. 1, pp. 57–69, 1998.

[102] C. Pohl, M. Franzmeier, M. Porrmann, and U. Rueckert, "gNBX-reconfigurable hardware acceleration of self-organizing maps," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, 2004, pp. 97–104.

[103] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–261.

[104] M. Porrmann, J. Hagemeyer, J. Romoth, and M. Strugholtz, "Rapid prototyping of next-generation multiprocessor SoCs," *In Proceedings of Semiconductor Conference Dresden, SCD 2009, Dresden, Germany, April 29-30, 2009, invited paper*, April 2009.

Author's Publications

[HTK+07] Michael Hussmann, Michael Thies, Uwe Kastens, Madhura Purnaprajna, Mario Porrmann, and Ulrich Rueckert. Compiler-driven reconfiguration of multiprocessors. *in Proceedings of the Workshop on Application Specific Processors (WASP) 2007 held in conjunction with the Embedded Systems Week, 2007 (CODES+ISSS, EMSOFT, and CASES)*, pages 3–10, 2007.

[PP08a] Madhura Purnaprajna and Mario Porrmann. Run-time reconfigurable cluster of processors. In *PhD Forum, International symposium in Parallel and Distributed Processing Symposium*, 2008.

[PP08b] Madhura Purnaprajna and Mario Porrmann. Run-time reconfigurable cluster of processors. In *Workshop on Design, Architecture, and Simulation of Chip Multiprocessors (dasCMP), held in conjunction with Micro-41*, pages 123–129, 2008.

[PPP08] Madhura Purnaprajna, Christoph Puttmann, and Mario Porrmann. Power aware reconfigurable multiprocessor for elliptic curve cryptography. In *DATE '08: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1462–1467, New York, NY, USA, 2008. ACM.

[PPP09] Mario Porrmann, Madhura Purnaprajna, and Christoph Puttmann. Self-optimization of mpsocs targeting resource efficiency and fault tolerance. In *NASA/ESA Conference on Adaptive Hardware Systems 2009*, pages 467–473, 2009.

[PPPR09] Madhura Purnaprajna, Christopher Pohl, Mario Porrmann, and Ulrich Rueckert. Using run-time reconfiguration for energy savings in parallel data processing. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA'09, July 13-16, 2009, Las Vegas, Nevada, USA*, pages 119–125, 2009.

[PPR+] Madhura Purnaprajna, Mario Porrmann, Ulrich Rueckert, Michael Hussmann, Michael Thies, and Uwe Kastens. Run-time reconfiguration of multiprocessors based on compile-time analysis. *Accepted for Publication in ACM Transaction in Reconfigurable Technology (TRETS)*.

[PPR09] Madhura Purnaprajna, Mario Porrmann, and Ulrich Rueckert. Run-time reconfigurability in embedded multiprocessors. *SIGARCH Comput. Archit. News*, 37(2):30–37, 2009.