Obtaining Performance and Programmability Using Reconfigurable Hardware for Media Processing

By
Ling-Pei Kung

SM Materials Science and Engineering
Massachusetts Institute of Technology
June 1989

SM Electrical Engineering
University of Southern California
August 1990

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in Partial Fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY
at the
Massachusetts Institute of Technology
February 2002

Signature of Author…….
Program in Media Arts and Sciences
January 11, 2002

Certified by……...
V. Michael Bove, Jr.
Principal Research Scientist
MIT Media Laboratory

Accepted by……...
Andrew B. Lippman
Chairperson
Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Obtaining Performance and Programmability Using Reconfigurable Hardware for Media Processing

By

Ling-Pei Kung

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on January 11, 2002
in Partial Fulfillment of the requirements of the degree of
DOCTOR OF PHILOSOPHY

ABSTRACT

An imperative requirement in the design of a reconfigurable computing system or in the development of a new application on such a system is performance gains. However, such developments suffer from long-and-difficult programming process, hard-to-predict performance gains, and limited scope of applications.

To address these problems, we need to understand reconfigurable hardware's capabilities and limitations, its performance advantages and disadvantages, re-think reconfigurable system architectures, and develop new tools to explore its utility.

We begin by examining performance contributors at the system level. We identify those from general-purpose and those from dedicated components. We propose an architecture by integrating reconfigurable hardware within the general-purpose framework. This is to avoid and minimize dedicated hardware and organization for programmability.

We analyze reconfigurable logic architectures and their performance limitations. This analysis leads to a theory that reconfigurable logic can never be clocked faster than a fixed-logic design based on the same fabrication technology. Though highly unpredictable, we can obtain a quick upper bound estimate on the clock speed based on a few parameters.

We also analyze microprocessor architectures and establish an analytical performance model. We use this model to estimate performance bounds using very little information on task properties. These bounds help us to detect potential memory-bound tasks. For a compute-bound task, we compare its performance upper bound with the upper bound on reconfigurable clock speed to further rule out unlikely speedup candidates. These performance estimates require very few parameters, and can be quickly obtained without writing software or hardware codes. They can be integrated with design tools as front end tools to explore speedup opportunities without costly trials. We believe this will broaden the applicability of reconfigurable computing.

Thesis Supervisor: V. Michael Bove, Jr.

Title: Principal Research Scientist

Doctoral Dissertation Committee

Thesis Advisor:

V. Michael Bove, Jr.
MIT Media Laboratory

Thesis Reader:

Thomas F. Knight, Jr.
Senior Research Scientist
Electrical Engineering and Computer Science

Thesis Reader:

Peter M. Athanas
Associate Professor
Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University

Acknowledgement

I owe many thanks to a lot of people, without whom I could not have survived these toughest years in my life. In particular,

My advisor, Mike Bove, for his great insights, perspectives and sense of humor. I could not have asked for a more understanding and patient advisor.

Tom Knight provided very helpful discussions on the subject of FPGA routing delays and semiconductor technology.

Peter Athanas, whom I had less frequent contacts, but his work in this area provided valuable background information. I look forward to more future interaction.

My colleagues, Stefan Agamanolis, for his inspiration and an always available helping hand. Bill Butera, for his great conversations and cheerful personality. Shawn Becker, for his help on software and understanding. John Wadlington, for his help and knowledge in hardware. Thanks to Cheops people, Mark Lee, Jeff Wong, Ross Yu, Brett Granger, Andrew Huang… Thanks you all, it's been a pleasure to know you.

Special thanks to my old officemates, Vadim Gerasimov, for his help on many windows problems. Nuno Vasconcelos, for discussions on image / video coding.

My friends here deserve a lot of appreciation. Thanks to Han Chen , the Wongs, Jen-Ju Huang, Mike Chou, Ralph Lin, the "ROCSA" volley ball family, and the many friends I came across over the years.

Thanks to Wan-Ping Lee and Chen-Ju Chen, who have given me very happy memories.

I owe most thanks to my sister, Ling-Ling, and her family. My brother-in-law, my lovely niece, Cheryl, and nephews, Brian and Kevin. Thank my brother, Ling-Yao, who has been taking care of our ailing father.

Finally, thank my parents for allowing me to pursue my goal. I know your sacrifices and constant worries. In particular, thank dad to gather enough strength and hang on to see this day. Your unconditional love is the source of my strength. I love you all.

TABLE OF CONTENT

1 Motivation 17

2 This Thesis 23
2.1 Issues 23
2.2 Thesis Outline 25

3 Evaluating Performance 27
3.1 Issues 27
3.2 Codes-Architecture-Organization-Compiler 28
3.3 Approach 31

4 Media Processing 38
4.1 Characteristics 38
4.2 Performance Metrics 41
4.3 Implementation Perspective 42
4.3.1 Instruction Rate 43
4.3.2 Reconfigurable Logic 46
4.3.3 Coding Efforts 48

5 Reconfigurable Architectures 50
5.1 Topology of Logic Cells and Routing Structures 50
5.1.1 Symmetrical Array 50
5.1.2 Hierarchical 52
5.2 Minimum Clock Period Estimation 54
5.2.1 Combinational Delay 56
5.2.2 Routing Delay 57
5.2.3 Clocking: Reconfigurable logic vs. Fixed Logic 62

6 Performance of Microprocessors 64
6.1 Performance Analysis 64
6.1.1 Instruction Rate 65
6.1.2 Factors 67
6.1.3 Speedup 67
6.1.4 SIMD 68
6.2 Datapath Performance Analysis 70
6.2.1 Datapath Properties 71
6.2.2 Data Processing Execution Model 73
6.2.3 Performance Properties 75

7 Architecture of Reconfigurable System 77
7.1 Technology Backdrop 77
7.2 System Architecture and Organization 78
7.2.1 Core Pairs 78
7.2.2 Reconfigurable Subsystem 82

7.3 Other Architectures 90

8 Performance Analysis 93
8.1 Performance Estimates 93
8.2 Application Scope 104
8.3 Front End Tool 105

9 Recap, Contributions 110

A Reconfigurable Computing Systems 112
A.1 Custom Computing Machines 112
A.2 Multimedia Reconfigurable Computing 112
A.3 Myth of Performance Claims 113
A.3.1 System Organization 114
A.3.2 Fabrication Technology 116
A.4 Discussions 117

B FPGA Design Issues 121
B.1 Design Issues 121
B.2 Tool Issues 124

List of Figure

Figure 1-1: Envisioned applications of reconfigurable computing and roadmap.....20

Figure 1-2: System and application development scenarios for embedded and general-purpose reconfigurable systems.....21

Figure 2-1: The organization of this thesis. 25

Figure 3-1: Performance is determined by the codes, the compiler, the architecture of the system and its components, and the organization of the components.....28

Figure 3-2: A task is either compute, memory, or I/O bound. 29

Figure 3-3: Determining performance bottleneck by emulating computer, memory, or I/O workloads.....30

Figure 3-4: Four codes-architecture-organization-compiler structures: (a) general-purpose computer host (b) with add-on application-specific subsystem (c) with add-on reconfigurable subsystem (d) future reconfigurable system.....31

Figure 3-5: Non-reconfigurable computer system design and application development paradigm. 32

Figure 3-6: Reconfigurable computer application development process.....34

Figure 3-7: Searching for new architectures (a) an ad-hoc organization based on existing organization (b) an integrated organization 35

Figure 4-1: Media processing covers a broad range of applications in digital signal processing, communications, pattern/object recognition, etc.....39

Figure 4-2: Typical multimedia processing tasks involves a data stream at the input and/or output.....40

Figure 4-3: Four media processing application scenarios: (a) both input and output require constant data rate (b) only input data must be consumed at a minimum rate (c) output must be produced at a minimum rate (d) data processed from one form and stored back. 41

Figure 4-4: A color transformation and alpha blending example (shown only the luminance). 43

Figure 4-5: Code segment of color space transform represented as a sequence of three-register instructions on (a) MMX architecture (b) non-MMX architecture.....45

Figure 4-6: Code segments in Figure 4-4 translated into hardware macro functions.....47

Figure 4-7: Pipeline retiming (or minimum period retiming) technique can be used to achieve high clock speed.....48

Figure 4-8: Instructions executed in sequence in the temporal domain vs. concurrent hardware execution in spatial domain.....49

Figure 5-1: Topology and organization of a symmetrical FPGA.....51

Figure 5-2: Xilinx's XC4000 single- and double length lines, with programmable switch matrices (PSMs).....52

Figure 5-3: Altera Flex 10K architecture. 53

Figure 5-4: Altera FLEX 10K family's LAB connections to row and column interconnect.....54

Figure 5-5: Assuming there exists a node with fanout of two in a real design. 56

Figure 5-6: Estimation of combinational delay. Example uses Xilinx XC4000 family FPGAs. 57
Figure 5-7: Altera FLEX 10K family FPGA combinational path delay.....58
Figure 5-8: Estimation of a lower bound on routing delay (e.g. switch based routing structure)..58
Figure 5-9: Estimate of minimum routing delay in Altera's FastTrack routing architecture.....59
Figure 5-10: Delays in Altera FLEX 10k family devices (speed rating -3) 60
Figure 5-11: Percentage of Routing Delay in Minimum Clock Period increase with the Size of the FPGA. 60
Figure 5-12: Routing Delay in FPGA Grow with Process Technology Upgrades.....61
Figure 5-13: VLSI design hierarchy (a) register transfer level (b) gate level (c) circuit level (d) physical layout 63
Figure 5-14: Register packing density (a) every register in a microprocessor is active (b) not every logic cell is used for every configuration, some are used for its combinational part only. 63
Figure 6-1: An instruction can be separated into data processing stream and control and data moving stream.....67
Figure 6-2: (a) Segmented (subword) ALUs for saturated arithmetic, (b) Segmented (subword) ALUs for arithmetic, logical, shifting, and any other operations.....69
Figure 6-3: A single pipelined functional unit connected to a register file. 71
Figure 6-4: Multiple functional units sharing the same register file.....72
Figure 6-5: Temporal expansion of *N* data processing instructions. 74
Figure 6-6: A block of *N* instructions execute on a pipeline functional unit.....74
Figure 6-7: Temporal expansion of multiple FUDP. *N* instructions may be distributed among *M* functional units.....75
Figure 7-1: A reconfigurable subsystem attached to the I/O bus of a general-purpose system.....80
Figure 7-2: A reconfigurable subsystem directly sitting on the processor bus. 81
Figure 7-3: Bus interface and FIFO are an integral part of the reconfigurable subsystem, but should be implemented with fixed logic (not reconfigurable).....82
Figure 7-4: A FIFO subsystem buffering data from system to reconfigurable substrate and vice versa.....84
Figure 7-5: Reconfigurable subsystem showing constituent fixed functions and a reconfigurable logic core.....86
Figure 7-6: Memory address for common registers is fixed, while application specific registers addresses can change with applications.....87
Figure 7-7: A special register for reconfigurable logic (SRRL) in an microprocessor's register file. 87
Figure 7-8: Reconfigurable system model showing the special register interface.....88
Figure 7-9: An integrate general-purpose reconfigurable architecture. The microprocessor core and reconfigurable logic core run asynchronously.....89
Figure 7-10: A datapath model of a general-purpose reconfigurable core.....90
Figure 7-11: The PRISC architecture. A programmable functional units is sitting on the register

bus. 91
Figure 7-12: Problems with PFU as a functional unit attached to a register file 91
Figure 7-13: Garp architecture [11]. 92
Figure 8-1: Examples for bound estimates (a) color space transform (b) a 19-tap filter 97
Figure 8-2: Finding upper bound visualized as minimizing the highest bucket 98
Figure 8-3: Finding lower bound visualized bucket filling. The heights of the buckets all increase by the same amount. 99
Figure 8-4: Speedup candidate decision diagram and properties that move the dividends 103
Figure 8-5: Front end for new reconfigurable system design 106
Figure 8-6: Front design flow for reconfigurable application development on existing systems. 107
Figure 8-7: Performance bound estimates as a front end tool integrated into hardware design flow. 108
Figure 9-1: A typical functional datapath block diagram of a reconfigurable subsystem on early custom computing machines 115
Figure 9-2: Performance comparison is more fairly judged using the same manufacturing technology. 117
Figure 9-3: Drop-in replacement for performance comparison 118

List of Tables

Table 6-1: Nomenclature of “packed” data in SIMD context. Numbers represent the number of bytes in the data. 69

Table 6-2: Reported speedups using MMX instructions for some multimedia applications. 70

Table 6-3: Performance properties of three popular microprocessors. 76

Table 8-1: Performance upper bound and lower bounds on color transform example using three representative microprocessors (32-bit datapath). 100

Table 9-1: System information on several reconfigurable subsystems. 120

Terminology and Concepts

Embedded Reconfigurable System (ERS): A system based on an embedded processor core (DSP, micro-controller, or a RISC core) and reconfigurable hardware.

Field Programmable Gate Arrays (FPGA) : In this thesis, we use FPGA for SRAM-based FPGA interchangeably, unless otherwise stated.

General-Purpose Reconfigurable Architecture (GPRA) : An uniprocessor or a subsystem architecture consisting of a general-purpose processor core, reconfigurable logic array, and configuration hardware. The processor core may contain from minimum hardware for any computable tasks to full-fledged superscalar processor with cache, memory interface and bus interface support.

General-Purpose Reconfigurable System (GPRS) : A computer containing GPRA, memory, and I/Os.

Instruction Set Processor (ISP) : Any processor which executes a set of instructions, together with memory, it can compute anything computable. This term includes general-purpose processors, DSP, microcontrollers and any other variations of instruction-based processors.

Reconfigurable Computing (RC) : Computation employing reconfigurable hardware as a co-processor in an instruction set processor-based system.

Reconfigurable Computing System (RCS) : An ISP-based reconfigurable system. That is the system consists of some kind of instruction processor core and reconfigurable logic array. Both GPRS and ERS are subsets of RCS.

Reconfigurable Hardware (RH): A piece of conceptual hardware consisting of RLAs and internal configuration hardware as part of a chip, the whole chip, or an aggregate of several chips.

Reconfigurable Logic Array (RLA): A conceptual piece of hardware consisting of a finite two-dimensional array of RLL and routing resources of varying granularities without embedded memory, I/O, internal configuration, and any other peripheral circuitry. We deliberately leave out the peculiarities of embedded memory, input and output ports, and other special-purpose peripheral circuitry typically associated with today's typical commercial FPGAs. This conceptualization helps us separate out the "non-general-purpose" elements and their properties in reconfigurable devices to focus on "general-purpose" elements and their properties such as the logic and routing structures. We later propose a general-purpose reconfigurable architecture on which reconfigurable hardware is integrated with general-purpose cores on the same chip. In that case, the boundary between input/output ports and reconfigurable logic is being merged or pushed out to the system boundary. Thus, our conceptualization lays a realistic groundwork for future general-purpose reconfigurable architectures. This conceptualization does not undermine our evaluation of reconfigurable architectures. Our models of reconfigurable systems can incorporate them later.

Reconfigurable Logic Array Core (RLC) : An alias for RLA.

Reconfigurable Logic Cell (RLL) : The smallest reconfigurable logic unit whose logic function can be independently configured.

Reconfigurable Logic Device (RLD) : Reconfigurable logic arrays embodied in a

physical chip with I/O pins and configuration support circuitry. A superset of RLD is a typical commercial FPGA device, where embedded memory and peripheral circuitry may exist. A typical example of reconfigurable logic device is a FPGA device. Two most popular FPGA vendors, Xilinx and Altera, have a broad range of FPGA devices with some forms of architectural specialization and features targeted for different market segments.

Reconfigurable Processor (RP) : A fully configured reconfigurable logic device which can perform some processing on data according to its resident configuration.

Reconfigurable Subsystem (RSS) : A system containing reconfigurable hardware and external configuration hardware (for example, a microprocessor interface that are necessary for microprocessor-controlled configuration, or a hardwired configuration controller and configuration memory). The subsystem may contain memory, glue logic, dedicated ICs, etc.

Semiconductor Processing Technology : The process of taking a wafer through various physical, chemical, and mechanical processing steps to make integrated circuits. In this thesis, we characterize a semiconductor process generation by its minimum feature size, metalization process, and operating voltage.

Subword Parallelism : Packing of multiple independent data of subword sizes into one big word so that operations on a word is equivalent to operations on all subwords within the big word.

Subword: one part of an equally-partitioned memory word.

1 Motivation

The Big Picture – Applications and Interfaces

Early computing devices were designed to meet the needs of scientists, engineers, and later merchants[1]. The early applications conceived and implemented were numerical routines to solve scientific or engineering problems, followed by transaction processing for commercial use. Personal computers were conceived for personal use, yet its first applications still evolved around problem solving for scientists and engineers. The most popular application for personal use was word processing. However, early word processing applications were not user-friendly as manifested by the notorious what-you-see-is-not-what-you-get problem. Introduced later, graphical user interfaces made application input and output interfaces look more graphical (though not necessarily intuitive), though they still wrapped around the same suite of applications. The claim that use of personal computers increased productivity was moot. Many studies found the contrary. In short, the early applications for (personal) computers were conceived, created, and used by scientists, engineers, programmers, and computer literates. As such, these applications had very little use for the average person. Let alone poorly designed user interfaces exacerbated the burden of learning to use them.

These two issues - mismatch between the application space and the mass population, and disharmony between applications and users, were natural products of an evolutionary process. The applications and interfaces were conceived and thus confined by the minds of their inventors and the available technology at their disposal. Scientists and engineers begot computing devices, contrived their applications, and molded their problem solving process into applications and interfaces. Limited computing power at the time dictated the kind of applications that could be accepted by users.

New Class of Applications - Multimedia

Over 40 years experience of using computers, a consensus has gradually taken form among the computing society at large. A computer should do what a user wants, not what it or its inventor wants[2]. And that user isn't necessarily a computer expert but an average person. Computers should help us gather, process, manage, and distribute information the way a human being does and only better. It should present the information to us the natural way - through our visual, aural, and other sensory forms. It should also help us communicate and interact with other fellow human beings the same way as if we were face-to-face. It should provide entertainment, education, and training contents the way we want and anywhere we want. The application suite must expand beyond scientific and engineering endeavors. They must deal with the kind of

---

[1] Dating back to the earliest device – difference engine (1823) and analytic engine (1833) designed by Charles Babbage to the first commercial computer IBM introduced in 1953.

[2] We use "computer" here to denote a broader class of computing devices, PDA, cellular phones, intelligent appliances, etc. That is, any system based on an instruction-set processor.

information we come into contact everyday. We call that kind of information "multimedia".

Changing Application Requirements

What we want means we want computing devices to be whatever we want them to be. It must be versatile, can do many things if not all things. The way we want means that they must be have natural interfaces. They must be intuitive to use. Anywhere we want suggests that they must be small and portable, and perhaps "they" are one single device. Be many things, portable, natural human interfaces seem to be conflicting requirements. However, we don't need "them" to be all things at the same time. It will require a sophisticated interface, contrary to our requirement, to manage the complexity. We just need "it" to be different things at different times. We can time-multiplex different functionalities, thus simplifying the interface to the user, onto the same pieces of hardware real estate. It must be "reconfigurable".

More Complex Applications

The idea of applying the interface metaphor of human-human or human-environment to human-computer interface isn't new. Curiously enough, and perhaps counter-intuitive because human beings can do this effortlessly, an application with "simple"[3], intuitive, natural human-friendly interface often requires high computing power. Hand-writing recognition, speech recognition, object recognition and tracking are such examples, just to name a few. At the dawn of personal computers, the computing power wasn't enough for any of these applications. Thanks to semiconductor industry's relentless drive to higher performance and lower cost, some multimedia applications using a natural human interface began to emerge. More advanced applications requiring more computing power always exist ahead of the technology curve. We are always looking for more "performance".

Many multimedia applications involve natural user interfaces via sensory devices. They bring computing technology closer to an average no-so-computer-literate person's daily life for the purpose of information or entertainment. We believe this is where future computing technology should make greater strides. Media processing technology should allow us greater freedom in exploring better and more natural human-machine interface. Multimedia applications, in particular perceptual signal processing, is the foundation of intelligent, cognitive processing of sensual data using computer.

Application Scenarios

We envision reconfigurable computing has many potential applications in the embedded as well as the general-purpose computing environment. A few scenarios illustrate the potential applications from an end user's point of view.

---

[3] Simplicity means hiding most of the complexity and exposing only a few parameters to a user.

Embedded

Imagine that one can add, change, or upgrade functionalities to his personal communication and entertainment equipment, without throwing away his old equipment, once new applications, new services, or new standards become available.

For example, one can upgrade his personal communication and entertainment equipment, such as a new and stricter just-in-time encryption algorithm recently deployed by the service provider, by simply downloading the new hardware configuration files into his cellular phone.

Imagine a cellular phone can be turned into a global positioning satellite (GPS) system when needed[4]. It works as a Bluetooth business card when you exchange information with your business contacts. Attaching a camera lens, it becomes a digital camera. With an ear bud, you can listen to MP3 music. When connected with a network cable, it lets you transmit data to the wired network.

In cases when you need to tell your friend about some information you keep in your notes, your cellular phone should be your organizer at the same time. Your PCES may function as a phone plus an organizer at the same time, but not a phone and a MP3 player, nor a GPS and camera at the same time. Normal usage conditions[5] make some combinations of functionalities very unlikely to be used at the same time.

An embedded reconfigurable system (RES) is ideal for our example. It can potentially offer a lower cost, lower power consumption[6], smaller weight, and smaller size solution over an all-ASIC solution.

General-Purpose

Imagine a user wants to create a high quality multimedia archive, containing life-long audio, photo, and video footages of his family members. He must be able to modify, annotate, search, play, and store this archive quickly. The archive must be indexed with many ways with different parameters or annotations for different searching and browsing conditions. The archive also must be compressed to save storage space. He wants to browse the archive quickly, yet he wants to see the footages at normal speed once he finds it.

Or a user wants to participate in an online class through video conferencing setups. He must be able to take notes, ask questions, show his projects, and see other classmate's demonstration. He must be able to see at least two video windows at the same time.

---

[4] We assume the analog front end for both standards are embedded in the system when manufactured.

[5] For example, we would not want to talk to someone on the phone while listening to music. We cannot use the LCD screen for the GPS and the camera at the same time. Whether limited by our sensual ability or device constraints, some functions are mutually exclusive.

[6] This is one very important constraint for mobile applications.

Current general-purpose system is still too slow to process high-quality image or video. For example, saving 3Kx2K 24-bit color image as a JPEG file takes around 25 seconds on a Pentium III-based system[7]. This photo quality is close to the quality of today's top of the line digital cameras. Imagine a future with digital camcorder of this or even higher quality.

Performance gains translate into increased productivity regardless whether an application has real-time hard constraints or not. Reconfigurable hardware provides potential performance gains to a general-purpose computer.

Figure 1-1 shows a vision of reconfigurable computing deployed in future systems. Some small number of fixed standard applications (as standardized by international standard bodies), in particular portable and wireless, may take advantage of reconfigurable hardware for smaller footprint and cost considerations. The actual physical embodiment of such a system will be largely determined by the application's input and output devices, and their physical form factors required for the user interface.

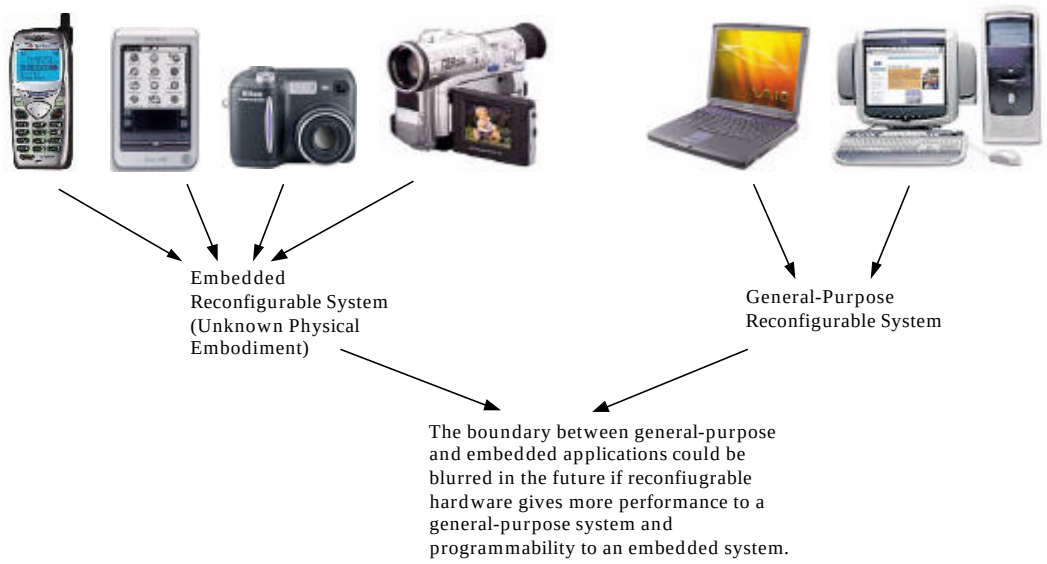


Figure 1-1: Envisioned applications of reconfigurable computing and roadmap.

In the general-purpose case, the system could be built with reconfigurable hardware as a high-end system, or as an alternative architecture where marginal hardware real estate (e.g. extra cache or processing units prove to produce marginal performance benefits) is traded for reconfigurable hardware. The cost consideration is irrelevant after-the-fact the initial system is built and an unknown and unbounded number of future standard or non-standard applications are yet to be developed. Typically, applications running on general-purpose systems are piled up mostly after fabrication, especially after

[7] The system has enough (384MB) memory so memory paging is not necessary.

development methodology and tools become more mature.

These systems will have well thought out I/O interfaces to connect to portable devices or smart electronic appliances for data processing, storage, and other tasks. It is also possible some convergence in architecture and organization, and good modular designs will emerge that changes the notion of embedded and general-purpose computing.

Challenges to System Design and Application Development

Incorporating reconfigurable hardware into a computing platform presents new challenges in system architecture and design, reconfigurable application development tools, and performance validation. Figure 1-2 shows the different system and application development paths for embedded and general-purpose reconfigurable systems.

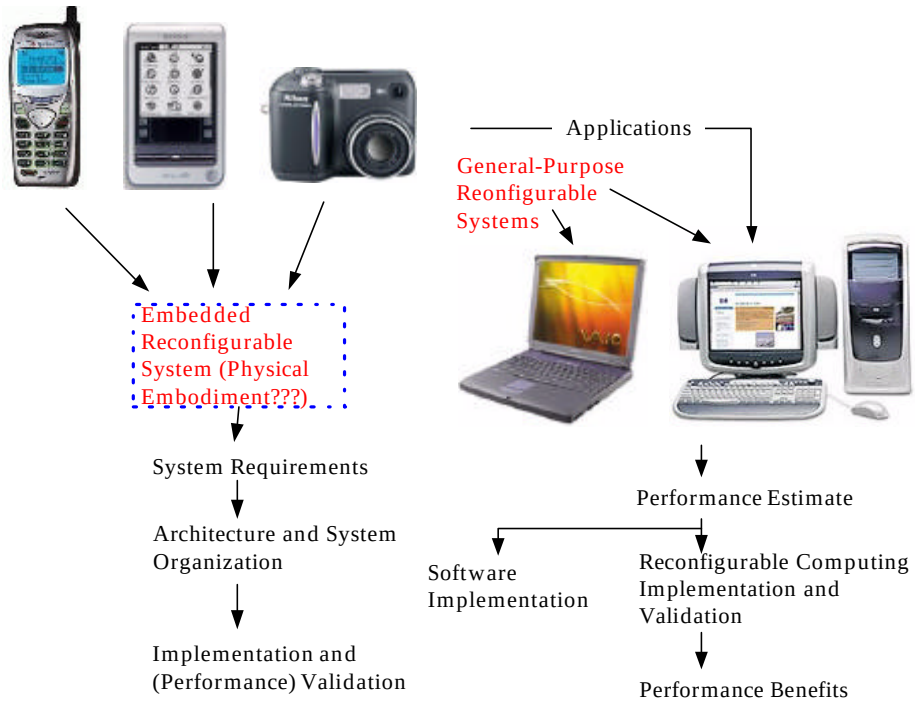


Figure 1-2: System and application development scenarios for embedded and general-purpose reconfigurable systems.

For the embedded system, a initial set of a small number of standard applications and features are targeted. The system is then built at the lowest cost possible. If the future expandability is planned or envisioned, then provisions, typically with some headroom in hardware real estate more than the required for the current set, must be considered during system design phase. The "cost" of hardware should be factored against the intended applications plus

The design challenge is to come up with a good architecture and system organization

that meets all requirements of this initial set of applications and also provide headroom for future application upgrades. System designers must select an appropriate combination of processor (or processor core) and reconfigurable logic device (or a reconfigurable logic core), and other system components under performance requirements. Cost consideration is very important for embedded devices, which typically target mass market. All components are subject to scrutiny for unnecessary hardware real estate. Further integration to reduce cost is also possible.

For general-purpose applications, there are two major challenges. One is to invent new "integrated" architectures, which pair general-purpose microprocessor cores with complimenting reconfigurable logic array cores. Pairing could potentially marginalized some microprocessors' capabilities and features. Microprocessor cores' real estate could be saved for reconfigurable cores' real estate, thus reducing the cost of adding reconfigurability. Exploration of general-purpose reconfigurable architectures is still in a very early stage.

The second, probably the biggest, challenge is to develop applications optimized for performance given the already present reconfigurable resources. The objective is to gain performance speedups as much as the built-in reconfigurable hardware allows and to as many application as possible. This objective is the justification of reconfigurable computing.

However, a new programming paradigm would be required for application exploration and development. The programming paradigm on a (non-reconfigurable) general-purpose computer is one such that performance is "measured" matter-of-factly after codes have been written. The same paradigm does not apply for reconfigurable programming as it often takes a much longer time and more significant expertise than conventional software programming.

An analogy to this added difficulty is to compare programming for distributed computing on heterogeneous computing resources versus programming for a uniprocessor system. Since the same task can be done on one single system and programming is much more easier (because program paradigm is well-established and tools are mature). It would only make sense to distribute tasks if the final performance in its totality is better than the single resource. The programming language, architecture, organization, and compiler for each computing resource can be different, resulting in different execution speeds for each task. Programmers must take great care to ensure correct program behavior and performance gains.

So far no proper programming model has been established for reconfigurable computing. Traditional software programming and FPGA design flow are being used as an makeshift programming paradigm. This disharmony is being patched up by human expertise. A more efficient programming paradigm must be contemplated and new tools for reconfigurable programming then can be created.

2 This Thesis

Several early research projects reported three orders of magnitude performance gains by using an ad hoc reconfigurable subsystem to augment a general-purpose system on some kernel functions (see Appendix A). The potential of such a high performance gain triggered a flurry of research activities over the years [1, 2]. Many believed that reconfigurable hardware, offering ASIC-like performance for a wide range of applications, would usher in a new computing revolution changing the old notion of computing we know of today.

Despite more than a decade of research and experiments, reconfigurable computing remains in research laboratories with no practical applicability to the large number of applications currently running on general-purpose systems. Nor have we seen multiple static or dynamic reconfigurations applied successfully in other areas of computing, such as embedded systems. Reconfigurable hardware, such as SRAM-based FPGA[8], is still, for the most part, being used for its traditional role as a fast prototyping low-cost ASIC substitute. So what has prevented reconfigurable computing to become part of the mainstream, general-purpose or embedded computing environment?

2.1 Issues

The design process of a general-purpose processor based system and the development of its applications are well understood and the infrastructure is well established. As a result, it saves much engineering time and cost by taking advantage of that process and infrastructure. In light of semiconductor processing technology's relentless drive to smaller and faster transistors, the loss in time results in the loss in technical superiority. Such is the reason that many reconfigurable systems were rendered obsolete a few years after their inception and reconfigurable computing remains in research laboratories and its application remains exclusively in the domain of scientific and engineering problem solving.

Modern architectural, such as VLIW and Multimedia extension, and micro-architectural, such as speculative execution, superscalar, heavily pipelined functional units, out-of-order execution, and dynamic register renaming, innovations in general-purpose processors have made great performance improvement by exploiting some form of application specialization and instruction level parallelism (ILP). These advances can significantly reduce the performance benefits of reconfigurable hardware and make some past performance claims moot. Given the economical disadvantages of developing reconfigurable systems and applications, one must ensure at least the potential performance gains exist before committing significant amount of time and resources.

From an end-user's point of view, reconfigurable computing has not demonstrated that

[8] In this thesis, we use FPGA for SRAM-based FPGA interchangeably, unless otherwise stated.

it can provide high enough performance gains to more than a few pedagogical examples. Throwing cost-effectiveness into the equation, it is hardly convincing as a viable solution against other computing technology.

From the technology point of view, more research is needed to explore application and performance space to drive the cost down. Due to the ever-increasing complexity to today's VLSI building blocks, a multitude of technical issues remain. In fact, reconfigurable computing is still in its infancy when it comes the infrastructure needed to support its exploration and development. Specifically, we found:

1. We don't know how to evaluate performance gains properly. A wide range of speedup claims are reported in the literature. A close examination raises questions on how to interpret these numbers. These questions include: the general-purpose reference platforms seemed to be randomly chosen; the reconfigurable platforms were upholstered with custom memory and I/O organizations that rendered them as application-specific solutions; the ISPs and RLDs implementations were not based on the same semiconductor process generation (ch 3 & 5).

2. We don't know what constitute a "good" (if this can be asserted at all) reconfigurable system architecture and organization. We don't know what constitute good component architectures and properties, in particular, those of microprocessors and reconfigurable logic. This reconfigurable system must allow us to explore added performance without losing programmability. Without pinning down architectures and system organization, we cannot evaluate performance benefits.

3. We don't know what properties affect reconfigurable logic's performance. Its performance is hard to predict, and we must go through the whole design process to find out. We ought to have a better way of "predicting", "approximating", if not, "bounding" it without time-consuming design process.

4. We don't know how to compare the performance of a fixed-logic microprocessor to the performance of a reconfigurable logic. The post-implementation measurement approach takes too long. We must have some idea about the potential of speedup for a particular task candidate before actual implementation.

5. We don't have more than rules of thumb as to what task characteristics are likely potential speedup candidates so that we can separate them out for reconfigurable computing. In this case, good or bad candidates are measured relatively against the competing ISP or RLA architectures that execute them. How do we measure the "fitness" of computing architectures given certain task properties?

6. Finally, we don't know how to efficiently and effectively program a given application for reconfigurable computing. Current development methodology for reconfigurable computing is an ad hoc marriage of traditional software programming and hardware design. Speedups are hard to predict, takes too

long, too much effort. We need better tools to explore reconfigurable application space.

These issues are intertwined and their interactions are often not separable. Until these issues are better understood, and perhaps solved, reconfigurable computing will see very limited use in a few special areas.

This thesis touches upon these problems and propose a way of harnessing performance benefits from reconfigurable hardware for media processing.

This thesis emphasizes analytical approaches and quantitative methods as much as possible. It is the view of the author that such approaches provide useful insights more conclusive than pure empirical results. As there are so much details to consider, it is hard to draw conclusions without analysis.

In the second half of this chapter, we outline the organization and relationships remaining materials in this thesis.

2.2 Thesis Outline

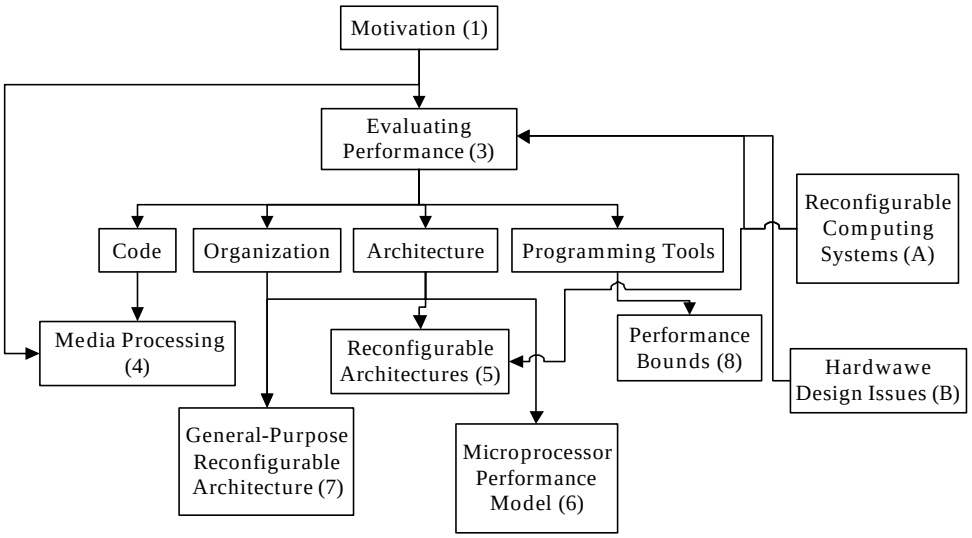


Figure 2-1: The organization of this thesis.

This thesis is organized in the following way:

Chapter 1 sets up the big picture, providing the motivation for more performance while maintaining programmability in new computing devices.

Chapter 2 discusses the issues this thesis tries to address. We begin by stressing that performance gains is the most important goal for reconfigurable computing.

Chapter 3 introduces what affect performance of a microprocessor-based system in its totality – code, architecture, organization, and compiler. It reminds us the complex interactions among these factors, and that we must set up the right frame of reference for performance evaluation. The rest of the thesis evolves from this chapter as shown in Figure 2-1.

Chapter 4 discusses the characteristics and performance requirements of media applications. These requirements serve as the metrics for performance evaluation. We use examples to show that some characteristics make microprocessors not ideal for media processing, which, in turn, present opportunities for reconfigurable speedups.

Chapter 5 reviews popular reconfigurable architectures and their performance-related properties. In particular, it reminds us of the trade-offs between reconfigurability, resource utilization, and performance. These trade-offs make predicting reconfigurable logic's performance difficult and put it at a speed disadvantage to fixed-logic devices. This chapter provides a theory to support this assertion. It then considers designs of any practical purposes and suggests an generous upper bound on the maximum clock speed.

Chapter 6 discusses properties contributing to modern microprocessors performance and different ways of performance evaluation. It explains the reasons for using an analytical approach. It analyzes the datapath of microprocessors and suggests performance evaluation techniques.

Chapter 7 proposes a integrated general-purpose reconfigurable architecture, which integrates a microprocessor core and a reconfigurable logic core across traditional device boundaries. It also discusses what constitutes the rest of the system and explains why each architectural or organizational choice is made. This architecture serves as the reference system on which we base our performance analysis. In the end, it reviews two other architectures and makes discuss potential problem areas.

Chapter 8 puts together everything set up in previous chapters - the application, the reference system architecture, reconfigurable architectures, and a microprocessor datapath model. This chapter proposes a quick way to estimate microprocessor's performance by performance bounds. This approach requires few properties on the task candidate. Given more information about the memory, it can also determine if the task is memory-bound or compute-bound. It then suggests how this performance analysis can be used as a front end tool for reconfigurable application development.

Chapter 9 summarizes all work and puts everything into perspective.

Appendix A briefly reviews previous research efforts on reconfigurable computing. It provides much of the backdrop this thesis originated.

Similarly, Appendix B tells from the user's point of view how despair the need for better tools support. Without it, reconfigurable computing will never be a viable computing option.

3 Evaluating Performance

Evaluating performance of microprocessor-based systems, or programmable devices in general, is not an obsolescent topic at all [3-5]. For general-purpose computers, it is still evolving with new technology and new applications [6]. This is a more pointed issue when we try to compare microprocessors with reconfigurable devices – both programmable, but in very different ways.

This chapter starts with this issue – comparing fixed-logic microprocessors with reconfigurable logic. It takes us back to the roots of performance of a computer – the complex interactions among code, architecture, organization, and compiler. We point out relevant issues under the reconfigurable computing context. This allows us to set up the right frame of reference for performance evaluation and explains how we approach it.

3.1 Issues

Due to the enormous amount of complexity, very few reconfigurable computer systems and applications were actually implemented. For the ones that were built, the designers had to settle for an makeshift system organization to keep the implementation effort at a manageable level. The resulting systems typically loosely combined an existing general-purpose system with an attached reconfigurable subsystem with its own dedicated memory and I/O subsystems. Appendix A lists some of these systems, their performance claims, and references to literature. Some the performance claims were on the order of thousands times over general-purpose hosts. However, questions arose on the meaningful performance comparisons between drastically different computing models, devices, and cost economics [7, 8].

Realizing the full potential of performance benefits can only be exploited without the constraints of existing computer architecture, later reconfigurable computing researches took on new reconfigurable architectures for general-purpose computing. Again, due to the enormous amount of efforts required in actual implementation, these efforts were confined to emulation or simulation without actual implementation [9-11]. Their reported performance gains are on the order of tens of percentage points to a few tens folds [10, 11], about two orders of magnitude less than previously claimed.

This wide range of performance claims underscores the difficulty in assessments of speedup benefits from reconfigurable computing. Such is the case even after actual reconfigurable program implementation (a posteriori measurement). Let alone the real challenge to reconfigurable program development is to be able to estimate speedups before actual implementations (a priori estimate).

There has been very few studies on proper speedup evaluation of reconfigurable applications. Lacking such knowledge often resulted in confusing and overrated speedup claims. In turn, it could mislead the directions of research and development efforts. To understand how to evaluate performance properly, we must know what

contribute to it and their individual contribution.

3.2 Codes-Architecture-Organization-Compiler

Performance evaluation, comparison, or measurement for ISP based machines must consider all factors that affect the final execution time of a task [12]. These include the instructions for the task, the architecture of the ISP, and the system architecture and organization. If the instructions are generated from a high-level description (e.g. a programming language) by a compiler, then the compiler plays a major role. The performance is affected by the codes-compiler-architecture-organization quadruplet (Figure 3-1).

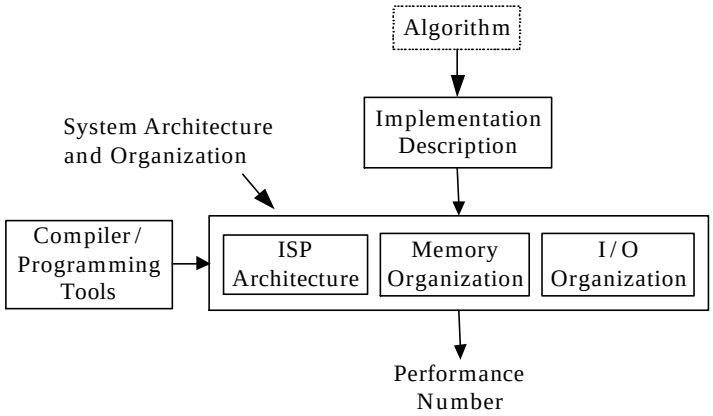


Figure 3-1: Performance is determined by the codes, the compiler, the architecture of the system and its components, and the organization of the components.

Computer architects try to balance compute, memory, and I/O capabilities and optimize against a select number of tasks (benchmark programs, kernels, or synthetic programs). The selection of optimizing targets is to represent typical workload. However, it is not statistically rigorous. Certainly it cannot cover an unbounded number of all computable tasks. No matter how computer architects try to balance compute, memory, and I/O performance, it is almost certain that no application will be an exact match to the system. Therefore, an application will either be processor bound, memory bound, or I/O bound.

Therefore, to improve performance for one particular application, we must determine where the bottleneck is. Then, we can either modify, add to the existing system, or design a new system to improve performance.

If a task is compute-bound, we can improve ISP architecture/microarchitecture on a per task basis. However, ISP architecture improvements are only justifiable when they are useful to a broad range of applications. That is, they must hit the "universally sampled" workload's statistical sweet spots. Otherwise, it is better off to create a separate device for that task, i.e. an ASIC solution.

If a task is memory-bound, it probably indicates the memory is too slow and/or memory organization is not optimal to the task. The solutions are to improve memory performance by using fast memory and/or to tailor memory organization for the task. Using faster memory requires little change to the architecture and organization of the rest of the system other than synchronously coupled devices must be fast enough to keep up with it. Changing memory organization requires re-architecting and re-organizing the rest of the system to match up. Unless this is beneficial to all applications or to all applications in a statistical sense.

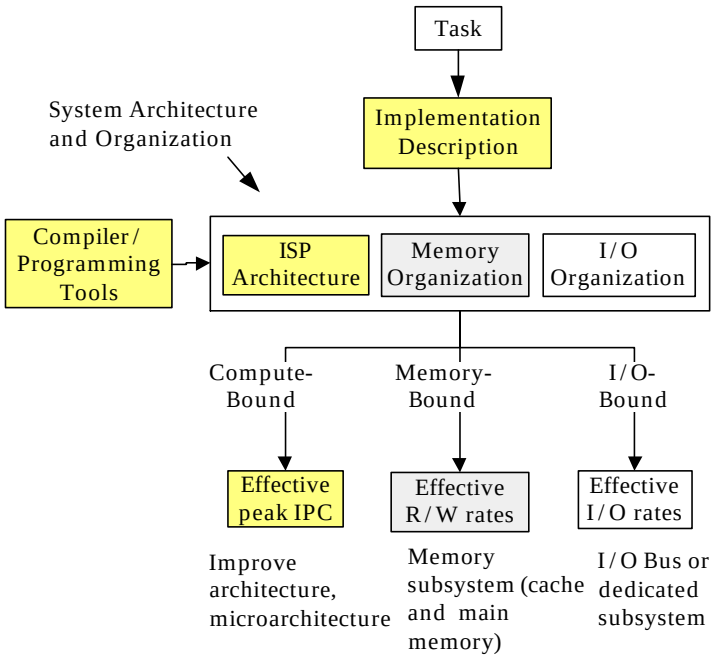


Figure 3-2: A task is either compute, memory, or I/O bound.

Similarly, if the task is I/O bound, we can add an I/O subsystem (as an add-on card) or design a new system with a higher I/O bus speed to solve the problem. General-purpose computers' evolutionary process had followed a performance improvement cycle of CPU-memory-I/O. Figure 3-2 shows this performance evaluation concept.

Since I/O is more application specific, which reduce its applicability, we will not treat it as the main application area for reconfigurable hardware in this thesis. It is understood that, when coupled with I/O, reconfigurable hardware can further improve performance.

To determine the bottleneck, the most direct and accurate way is to implement an application, then measure it by profiling. Profiling may require writing additional codes and may not yield obvious information on the bottleneck. It points the function or region of codes where we insert profiling information, but it doesn't tell us whether the CPU, the memory, or the I/O is the bottleneck. Another way is to calculate maximum instruction and/or data rate achievable from the actual ISP's architecture, then check if

the memory and I/O capabilities can match it. In other words, we are trying to find a minimum schedule for a task without considering memory and I/O speeds first. Another way to look at it is an abstract system architecture and organization with infinite memory and I/O bandwidth, an equivalent view when the program is compute-bound. This concept is explained in Figure 3-3.

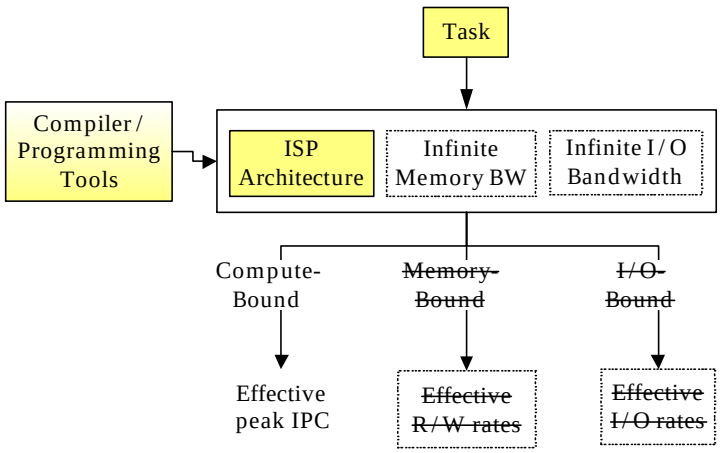


Figure 3-3: Determining performance bottleneck by emulating computer, memory, or I/O workloads.

This scheduling can be very complicated and most likely intractable for complex programs running on modern superscalar or VLIW architectures [13]. However, media processing has characteristics that lend themselves opportunities for simplification and approximation. This will be discussed in the next chapter.

One way to improve performance for some applications (or commonly used functions) to an existing system is to add a subsystem that improves the compute power, memory bandwidth, and/or I/O bandwidth of the host system (Figure 3-2). An "add-on" card can provide any combination of compute power, memory, and I/O bandwidth to particular tasks (Figure 3-4 (b)).

Often compute power is coupled with dedicated memory organization and standardized I/O protocols to maximize performance. I/O protocols are often coupled with transmission line physics and connectors' form factors. This makes sharing I/O devices difficult. In other words, for different applications requiring different I/O protocols, it is often necessary to create separate subsystems for each application, making the subsystems application-specific. This is exactly what we try to avoid using reconfigurable hardware.

Figure 3-4 shows the different ways of adding performance to a general-purpose machine (Figure 3-4 (a)). An application-specific subsystem is necessary to meet hard performance requirements or provide specific I/O capability (Figure 3-4 (b)). A reconfigurable computing system seeks to replace many application-specific devices and still provides additional performance versus a non-reconfigurable counterpart (Figure

3-4 (c)). Early reconfigurable subsystems aimed to improve all compute, memory, and I/O performance of an existing system (Appendix A). However, the overall performance benefits came not entirely from the adaptability of reconfigurable logic to specific tasks. It also came from newer fabrication technology and some degree of organizational customization.

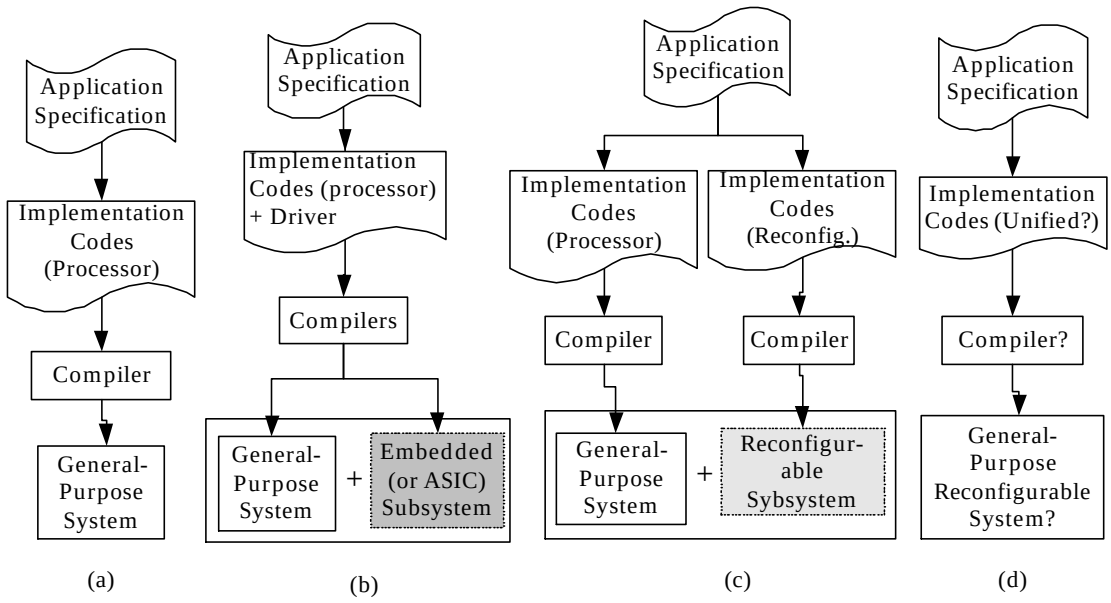


Figure 3-4: Four codes-architecture-organization-compiler structures: (a) general-purpose computer host (b) with add-on application-specific subsystem (c) with add-on reconfigurable subsystem (d) future reconfigurable system.

We should emphasize that organizational customization, though benefiting some applications, will reduce its degree of freedom as a "reconfigurable" computing device. This will push the reconfigurable subsystem more toward an application-specific device (Appendix A). To make reconfigurable computing a viable technology, we must move away from "very" application-specific to more "reconfigurable". This leaves us an open question: what is a "good" architecture and organization for a reconfigurable computer? How is everything in the code-architectures-organization-compilers quadruple tied up together (Figure 3-4 (d))?

3.3 Approach

To answer our question at the end of last section, we start by looking at how the code-architecture-organization-compiler and performance are knitted together in a non-reconfigurable computer. We offer a complete view from the performance evaluation stage at the processor design phase, to its incorporation in the design of a new system, finally to the performance benchmarking after the system is made. Figure 3-5 shows a typical design flow from the design of a new ISP to the measurement of performance on a system.

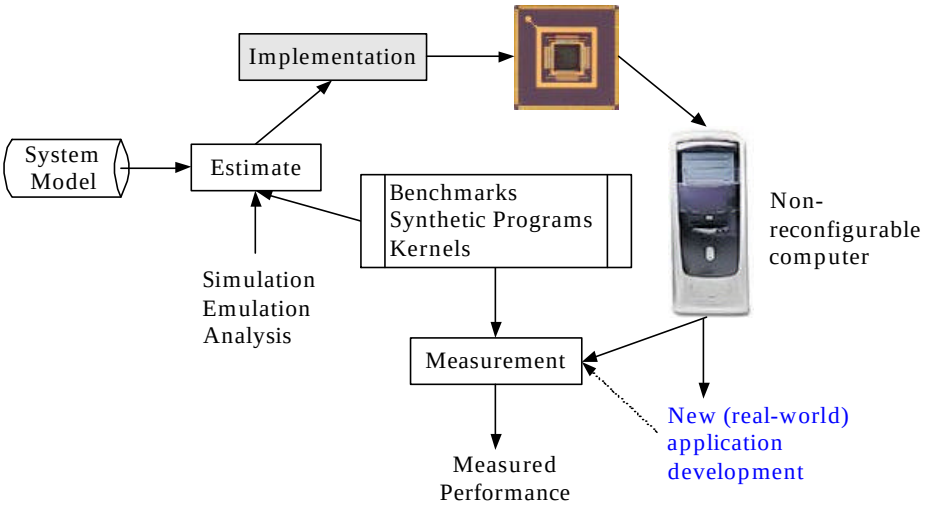


Figure 3-5: Non-reconfigurable computer system design and application development paradigm.

Each of these stages involves quite a significant amount of details. Nonetheless, several points are noteworthy:

1. The implementation takes a long time and incur high cost, thus some kind of performance estimate upfront offers some assurance that the design has competitive advantages. This is important not from the perspective of its capability but its survivability in the market place.
2. The system model (of a reference platform) provides necessary details for performance estimate. Processor designers optimize their design under the assumption it will be used in such a system. Of course, the processor can be used in any system and its full processing capability may not be fully utilized.
3. The performance estimate can arrive from three different approaches: simulation, emulation, and analysis. The evaluation "programs" are usually synthetic programs, or real kernel programs. Depending on the accuracy required, the estimate can be reported in IPC, clock cycles, or actual time. The amount of effort required also increases with the accuracy.
4. Once the processor is "designed-in" with the rest of system components, the actual performance can be "measured" against known benchmarks. However, for new application development, getting the most out of the system requires hand optimization. Unless an application has hard constraints, program development is usually best-effort, using a high-level programming language and relying on an optimizing compiler.

Now let's consider a future reconfigurable computer by first leaving the specifics about the reconfigurable logic and how it is integrated in the system or with the processor open. Instead, we start by arguing the following points:

1. The design of the processor part and the reconfigurable logic part should have minimal impact on each other for the following reasons:
    a. All existing applications can still run on the system without having to configure the reconfigurable logic. This forms a baseline for any application.
    b. Preserve current processor evolution to continue through architectural and fabrication improvement. Allow reconfigurable architectures to evolve around the processor architectures as complimentary entities.

2. Like the development of non-reconfigurable computer, the processor design part can use some system information and synthetic kernels for performance evaluation and optimization. However, the reconfigurable logic part should not be optimized toward these programs for the following reasons:
    a. The processor is already optimizing toward them, there will be less room for performance improvement. Isn't it reconfigurable logic's strength – to do what processors do poorly.
    b. Optimization of programmable architectures is a double-edged sword. If we optimize them for certain tasks, we lose on the others. Unless we have convincing usage statistics, we don't know for sure if we gain anything. Furthermore, usage is hardly static, it changes with time and technology.
    c. Reconfigurable logic's advantages varies significantly with its physical attributes, hard system details, as well as the actual tasks. It can be incorporated under different system architectures, organizations, and components. For example, it can be integrated with an ISP core on the same chip. The chip can be used in any new system design. It is impossible and meaningless to optimize for all of them.

3. New application development is opportunistic, not all applications will benefit. Performance speedups is "system specific". That is, unless everything is equal, the processor, the reconfigurable logic, and other system details, the speedups (or sometimes even slowdown) are likely different.

Our arguments stress that the real problem for the viability of reconfigurable computing lies in the application development stage. Figure 3-6 shows two application development paths. Boldface lines represent shared paths. Thin solid lines (together with the boldface lines) represent a path for pre-implementation estimate. Dotted lines (together with the boldface lines) represent a post-implementation measurement.

Since both paths can be cyclic, i.e. iterative, it is desirable to take the shorter path to minimize the time for each trial. The question boils down to whether we can get an estimate faster than we can implement and measure it.

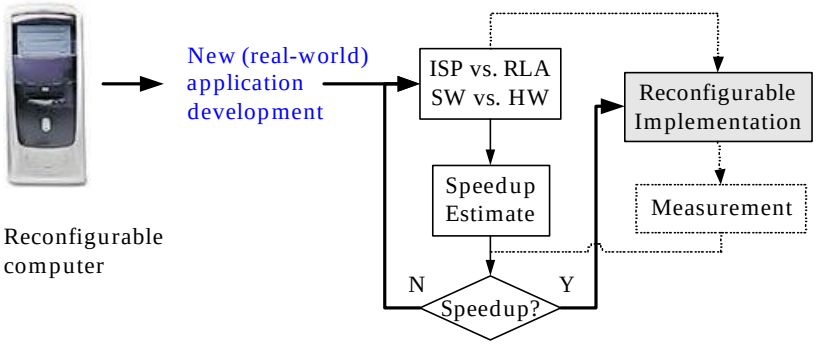


Figure 3-6: Reconfigurable computer application development process.

We know that processor architects have been exploiting instruction-level parallelism. This trend will continue and it will push the speedup opportunities to medium to large complex operations [14-16]. Implementing a medium to large complex reconfigurable function can take days to months of time depending also on the margin of performance gains. These observations suggest that we should look for ways to make speedup estimate quickly, thus avoiding potentially time-consuming implementation trials.

Therefore, we need to look at the factors affecting performance and determine whether and how we can simplify a complex system to make the application development easier.

Code

Programming on a reconfigurable computer (Figure 3-4 (d)) is still an area of researches. Most systems used a combination of a general-purpose programming language and a hardware description language [17]. [18] had to develop a logic description language when logic synthesis for FPGA was not available.

General-purpose programming languages (especially imperative and sequential) cannot describe the concurrent events of hardware. A restrictive subset of a general-purpose programming can be used for hardware description, but it requires programs to conform to coding guidelines, and possibly using pragmas or compiler directives. Even later when HDL-based logic synthesis tools became available, they required designers to follow coding guidelines to generate acceptable results.

Whether there is a language suitable to describe software and hardware functionality and structures at any level of details is an open question. An evolutionary programming paradigm followed by most CCMs is shown in Figure 3-4 (c). We leave this question open, instead we move to a higher-level representation – data flow graph (DFG). This representation, with additional directives, can be translated into actual codes automatically or manually. We come to this choice for the following reasons.

1. Media processing exhibit dataflow pattern due to the nature of applications, in particular, in the region of interest – heavy data processing loops (chapter 4).
2. Starting from this representation allows us concentrate on the processing task

itself, not the actual coding. We do not have to rely on programmers' masterdom of the language(s) and/or compilers' ability to fully exploit the capability of the hardware.

3. It allows to examine the "fitness" (e.g. parallelism) of microprocessors for media processing at this functional level (chapter 4 & 6). Once we code it with an imperative language, we may lose sight of the parallelism in the original task.

Two requirements for the DGA representation is that it should be acyclic and that the inputs and outputs be specified with their sizes. Further optimization is possible if the actual dynamic ranges of each input or output is specified. This will allow us to use the minimum data representation required for intermediate data.

Organization

Computer organization is part of the equation in determining the performance of the system. Given the same components (except the glue logic that pieces them together), one can design systems with different performance numbers for the same programs.

The "proper" organization of a reconfigurable computer (Figure 3-4 (c) or (d)) is still an open research question. Figure 3-7 (a) shows an makeshift organization, which is an evolutionary step from non-reconfigurable computer. However, performance bottleneck and technology feasibility push us to think beyond traditional organizational boundaries (chapter 0). This push breaks organizational boundaries and allows us to re-think architectures at the component level (Figure 3-7 (b)).

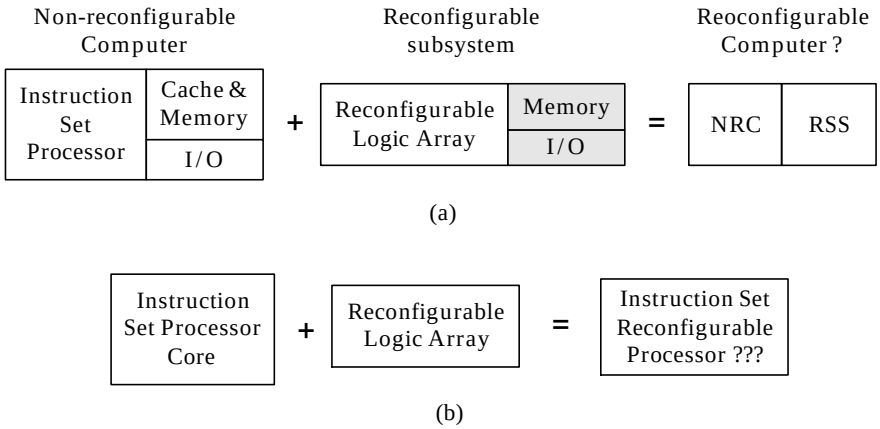


Figure 3-7: Searching for new architectures (a) an ad-hoc organization based on existing organization (b) an integrated organization

With the backdrop of more system integration and architecture innovation in the future, we recognize the following points:

1. It is technically feasible to integrate microprocessor core and reconfigurable logic onto one device and it is just one of many possibilities of future integration. The integration can extend beyond processor and reconfigurable logic to include

memory and I/O on the same chip.

2. The reconfigurable logic should not have dedicated memory and it should not be tied up with specific I/O standards (Appendix A). Architects should avert organizing reconfigurable logic for any particular task.
3. The system memory should be organized in a way that both ISP and reconfigurable logic have the fastest access to the memory. Any organization should not sacrifice ISP's access speed in exchange for reconfigurable logic's access speed (chapter 7).
4. Performance related system information (e.g. secondary cache and memory organization, bandwidth, etc) is available (chapter 6).

Architectures

As we are still searching for a "favorable" system and component architectures (Figure 3-4 (d)), we do not assume knowledge of the exact partitioning boundaries between fixed-logic instruction set core and reconfigurable logic core. However, we do acknowledge that there are numerous instruction-set architectures (ISA) and reconfigurable architectures (RA). One way or the other, all programmable architectures are more favorable to some applications over the others. It is likely a myriad of ISA/RA combinations can exist for different markets.

1. We require the architecture contains a microprocessor core such that with the rest of the non-reconfigurable components they constitute a programmable system. This core is exemplified by current commercial ISP architectures: single-issue, superscalar or VLIW, etc.
2. There are also a plethora of reconfigurable architectures of different granularities in logic and routing configurability. We also do not exclude any architecture, including multi-context reconfigurable logic [7], other than:
    a. Our intended use is to compute, embedded memory is allowed but will not be considered in our performance evaluation.
    b. Specialized peripheral circuitry surrounding a typical commercial FPGA is stripped for the same reason.
3. Cache, bus interface, memory controller can be customized later. This allows fine tuning for better integration and sometimes trade-off for cost and utilization concern is possible.
4. All performance-related information on the microprocessor core is available (chapter 6)
5. All performance-related information on the reconfigurable logic core (all timing information plus physical properties such as array sizes, organization, number of interconnection ports to non-reconfigurable hardware) (chapter 5).

Compiler - Programming Tools

Programming a reconfigurable system (Figure 3-4 (c) and Appendix A) required both

highly skilled hardware designers and software programmers [17, 19]. There were no well-established development processes, nor design tools targeted toward solving unique problems in programming a reconfigurable system. It required a well-orchestrated, synergistic hardware and software development effort using tools designed for other purposes. Therefore, it was often an iterative and time-consuming process, especially when a particular application called for high performance and efficient utilization of FPGA resources.

In addition to complexity, the problem was in part the architecture was ad hoc and so were the programming language and tools. Noticeably, there was less focus on the programming tools than on the architecture or applications of the system. In particular, tools to help determine the "performance-directed" partitioning of tasks between the ISP core and the reconfigurable logic core was lacking.

We have intended to leave programming language alone and instead focus on a higher-level representation – DFG (see section on code). Following our discussion in 3.1, our focus is to quickly evaluate and compare ISP and reconfigurable logic's performance for a task.

There are three approaches to performance evaluation: performance measurement, analytical performance modeling, and simulation-based performance modeling [20]. Analytical performance modeling and simulation-based performance modeling are the usual ways of performance evaluation for microprocessor architecture exploration in the design phase (Figure 3-5). Such a priori evaluation in necessary to assure meeting performance goals and avoid expensive non-recurring engineering cost. However, either simulation or emulation uses the actual design as the starting point, which is what we try to avoid.

In light of performance gains and task partitioning are very sensitive to system details, we expect those two approaches infeasible. Instead, we are looking at ways to analytically predict performance for the ISP core and the reconfigurable logic core. We set several goals for this analysis approach:

1. The analysis should be progressively more accurate given progressively more details about each core.
2. The analysis on the ISP core should yield optimistic estimate as what is possible custom codes. This estimate can be used to check if the codes running on the ISP core is efficient.
3. The analysis can be integrated with future compilers for partitioning exploration.

4 Media Processing

To talk about performance of programmable systems, we must put them in context. In this chapter, we discuss the characteristics of media processing from microprocessor and computer architecture perspective. From this perspective, we discover areas of weakness inherent in a fixed-logic programmable device. We use examples to illustrate these points, which reconfigurable can explore .

4.1 Characteristics

Media processing finds its root in the general areas of multidimensional signal processing, pattern/object recognition, and synthesis. These areas cover a large number of applications. Some specific examples include speech recognition/synthesis, digital audio, digital photography, digital video, video conferencing, computer graphics, distance learning, pattern recognition, object recognition, data mining, etc. Whether for the purpose of information communication or for entertainment, media processing has become one of the most exciting area of computing.

[21] has a good coverage on multidimensional digital signal processing. [22] [23] contain media processing in numerous communication applications. More advanced applications in pattern and object recognition based on media signals are still a largely unfulfilled field. [24, 25] present computational characteristics of such applications. [26, 27] are two such examples involving image and video.

There are important intrinsic characteristics to consider when implementing a media application to a computing device. Computational properties and structures are difficult to quantify. While a systematic and quantitative characterization of application properties doesn't exist, some qualitative observations will help us develop intuitive sense.

Continuous-Time Origins

In general, media processing involves the use of input and/or output devices that are continuous-time analog signals from sensory input devices or to output devices for human's perceptual processing. The analog signals are digitized for processing, storage, and/or sent over a communication channel, eventually transformed back to analog forms for human perception. Figure 4-1 shows that media processing has its data originated from analog sources even though any particular subtask may not involve any analog inputs or outputs.

A typical end-user level multimedia application involves processing of one or several continuous streams of data from a source, to a destination, or both, in real time. The inputs can be data streams generated from the sampling of continuous-time analog signals, such as speech, audio, video, sensory or other forms of analog signals. They can also be descriptions, models, programs, and scripts, which describe the composition or synthesis process of an output data stream. The outputs can be data streams processed from the input streams or rendered data streams from input descriptions. They can also be commands calculated from the input streams for controlling other devices. The

stream path from the input to the output may involve a network. Figure 4-2 shows some typical processing tasks after analog-to-digital capturing phase.

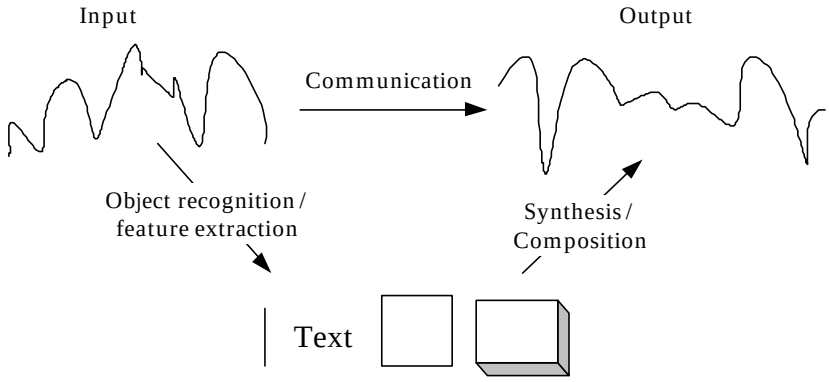


Figure 4-1: Media processing covers a broad range of applications in digital signal processing, communications, pattern/object recognition, etc.

The two boldface red lines Figure 4-2 also show complete applications may consist of several subtasks chained together in time. Note that between processing tasks, the data may go through a switch or a temporary buffer for data rate regulation between two subtasks. This continuous-time origin at the inputs or outputs lends media processing algorithms more naturally represented as data flow graphs versus control flow graphs.

Data locality

Since media data are sampled analog signals, their relationship inherit from what is in the original analog signals. Storing and accessing data in memory are most efficient when this sampling order is preserved since no additional information is necessary to identify each datum in the original time or space domain. The identity of each datum is implicit in the order.

For this reason, the time and space relationship in the original signals is preserved array data. Media processing often involves neighborhood operations, rarely causing cache misses [6].

Small discrete data types

There are only a few most commonly used media data types at the inputs and outputs (Figure 4-2). This is due to our perceptual limitations. We can discern only a relatively small number of levels for color, sound, etc.

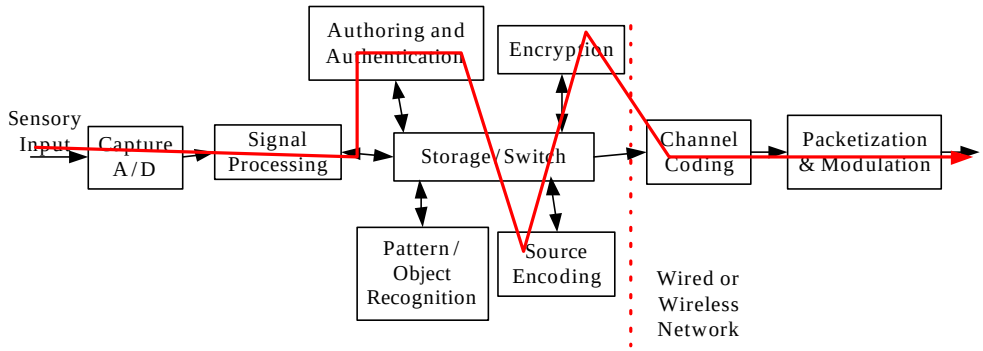
Fuzzy data precision

Unlike scientific numeric computing or mechanical CAD, where precision is mandatory, we are quite tolerate to errors in media signals. This lax data precision requirement can be exploited to reduce processing requirement or traded off for other requirements.

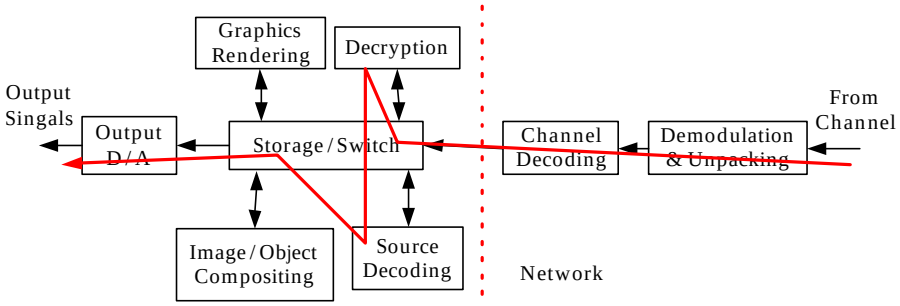
Data parallelism

Parallelism arises in media processing due to:

- Multiple independent channels of sensory data.
- Data can be gathered in memory and presented to the next processing device in parallel.



(a) Media processing at source end



(b) Media processing at destination end

Figure 4-2: Typical multimedia processing tasks involves a data stream at the input and/or output.

Large amount of data

Large amount of data, possibly unbounded input and/or output data due to runtime variables. For example, we cannot anticipate when an user will turn off his digital television.

Same operations

These samples undergo the same processing operations.

Datapath intensive

Media processing involve more datapath processing than branching. This is also, in part, due to the large amount of data between any semantically meaningful boundaries, where special processing is required, (e.g. the edge pixels in an image) in the data.

Data processing operations require data processing instructions and data transfer instructions translates into more three-address instructions than the rest of the instructions in the instruction set (6.1). Thus we focus on the datapath processing capability of ISP less the instruction flow control aspects of the ISP.

4.2 Performance Metrics

The input/output bandwidth for multimedia computing is high due to the large amount of data samples created or consumed continuously in time at the analog/digital interface. A processor on the average must consume these data points at the same rate. This requirement, in turn, translates into both high computational and high communication bandwidths at intermediate processing stages (Figure 4-2). This problem is exacerbated when in a multi-tasking environment, where multiple media applications can be launched.

Therefore the most important multimedia applications performance requirements is throughput, or the data processing rate. Depending on the application, latency may or may not be as important. Figure 4-3 shows four scenarios of throughput and latency requirement.

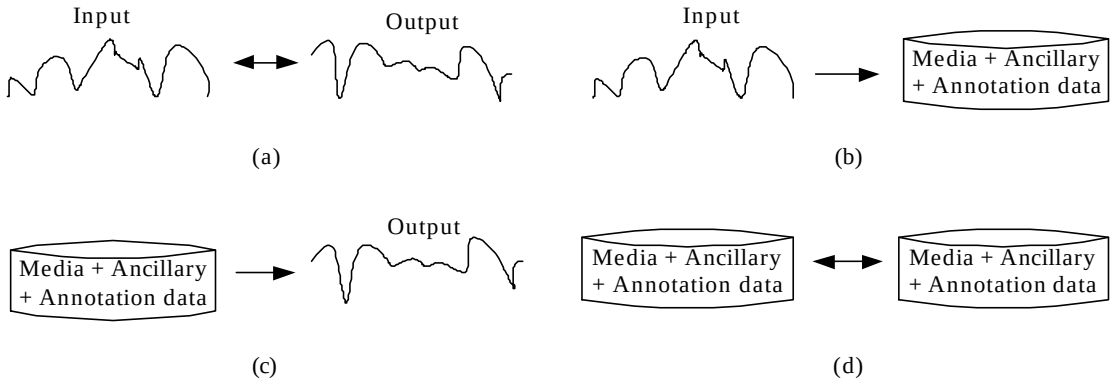


Figure 4-3: Four media processing application scenarios: (a) both input and output require constant data rate (b) only input data must be consumed at a minimum rate (c) output must be produced at a minimum rate (d) data processed from one form and stored back.

For example, (a) shows a scenario for communication applications, such as videoconferencing, audio and video telephony, real-time broadcasting where throughput must exceed a minimum and latency must be bounded. Throughput must meet minimum for (b) and (c) because the analog source (destination) generates (consumes) data at a certain rate. Neither is critical for (d) as the data is processed from one then stored back. Regardless of the applications, on a programmable multi-tasking system more media processing (or other) tasks can run if the throughput of each task can be improved.

For most two-way multimedia applications involving symmetrical communication, such

as videoconferencing, video telephony, Internet phone, and interactive Internet video games, the latency must be small enough to make such applications viable in the marketplace. Other applications, such as automatic target recognition and robot vision, must also meet this requirement for the nature of these time-critical applications. Other local, one-way, or two-way asymmetrical applications, such as virtual reality, interactive graphics viewing, digital audiovisual services also require minimum latency.

4.3 Implementation Perspective

In this section, we examine mismatches between media processing characteristics and common microprocessor architectures. We then look at whether these mismatches can be exploited by reconfigurable logic. It is better to begin with an example.

Figure 4-4 shows an color transformation plus alpha blending example. It has all the characteristics enumerated in the last section. A dynamic range analysis on inputs and all dependent variables including the final output results in the minimum-sized data representation shown in blue numbers.

A finer control on data precision is possible if we know exactly what the dynamic ranges of individual inputs. Sometime data values do not cover the full range of the size (for example, the Y, C_r, C_b values conforming to CCIR 601). In this case, half of the input variables are constants. We can use this information to further reduce the minimum-sized data representations of intermediate variables (constant propagation). The dynamic ranges are shown in purple, the minimum sizes are shown is red, and the smallest microprocessor data types are shown in green.

We are interested in the latency and throughput of this task. The latency is the time from which all inputs are valid to the time the output is valid. The throughput is the inverse of initiation interval, the time between each set of input data can be updated.

We also assume the restrict ourselves in only considering this task as indivisible. This allows to concentrate on region scheduling for local optimization. However, this is sufficient to get our points across – to show weaknesses which ISP processor can never overcome.

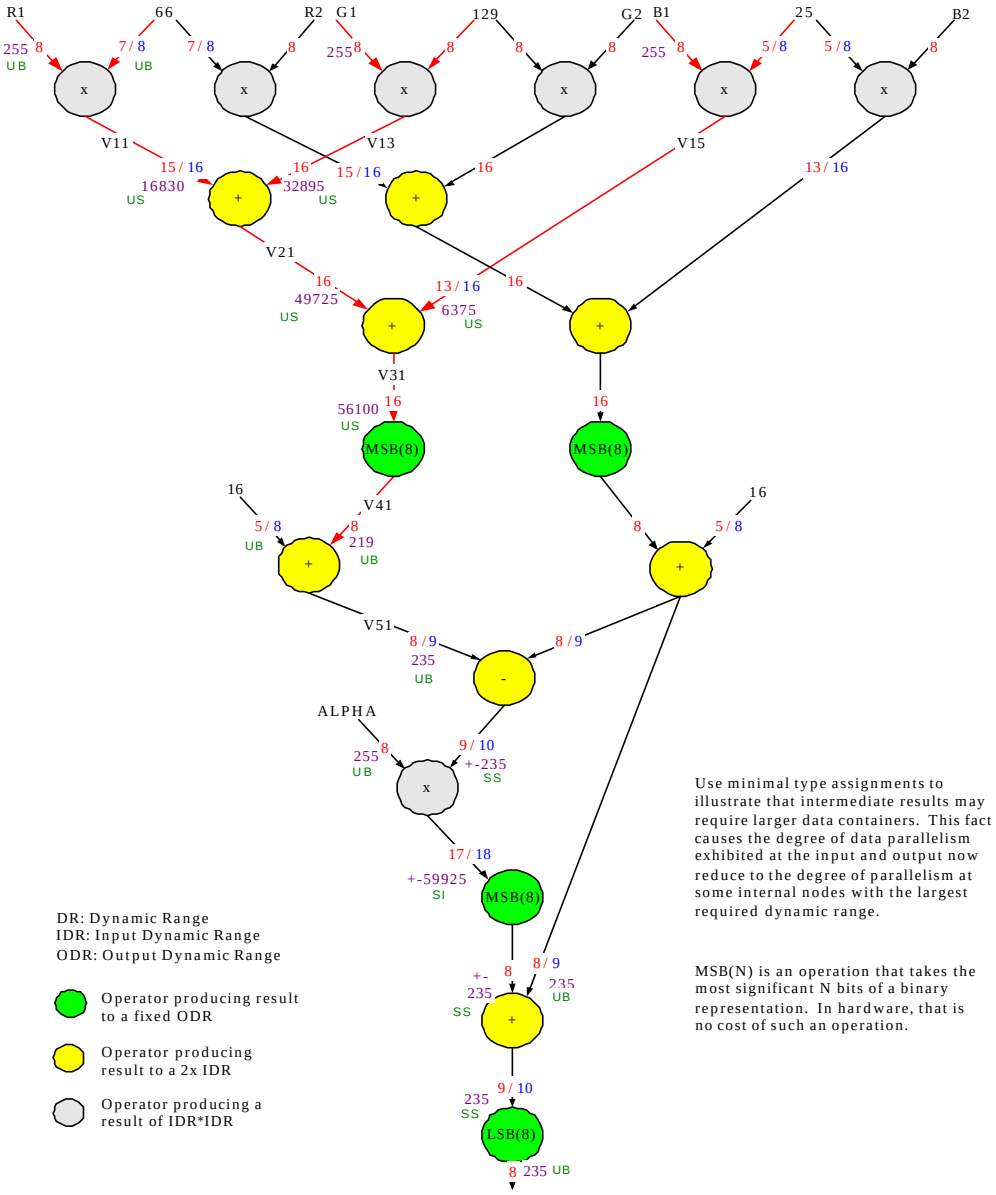


Figure 4-4: A color transformation and alpha blending example (shown only the luminance).

4.3.1 Instruction Rate

Figure 4-5 shows a portion of Figure 4-4 (edges shown in red) in a register transfer style. Figure 4-5 (a) shows a task fragment implemented in SIMD instructions. Figure 4-5 (b) is the non-SIMD version. We also assume there is only one functional unit, which does all arithmetic and logical operations, including multiply. These operations may be associated with different costs (latencies) to reflect actual processor design. This is not the case for modern processors with more than one integer units. This will be considered later.

The rectangles represent general-purpose (without solid dividends) and SIMD (with solid dividends) registers. The SIMD registers consist of smaller regular registers denoting that they are segmented. Half shaded rectangles represent registers with small data occupants. The arrowed lines represent operations supported natively by the processor.

The subword pack/unpack operations required in figure are common and are supported by most SIMD extensions. However, it is very likely that each architecture directly (through specific instructions) supports a different subset of all possible combinations of packing and unpacking of different subword sizes. This subset constitutes a basic set from which all permutations can be realized, through some subword shuffling may require multiple packing/unpacking instructions.

In the above example, the input data for the MMX architecture are already packed, while for the non-MMX version they are not packed. We assume previous processing steps already prepare (pack/unpack) the data according to the respective architecture. However, we do not assume, for the MMX-capable architecture, the input data is packed for maximum parallelism for the current function (i.e. color space). This is to reflect that it is unlikely for any particular one subword organization to remain optimal throughout the whole application. Therefore, even though we can optimize one subword organization for a particular function or a particular processing segment, the global optimal subword organization may be different than this local optimal organization. The above figure promptly exhibits the complexity of optimal packing and unpacking, which could vary for each operation.

From left to right, we arrange the instructions in their dependency order and it has the notion of time (for single-ALU processors). We align the registers so that the length reflect the number of passes through the ALU. In the single-ALU case, we can derive the actual time by summing all instructions with their respective costs. We also observe a couple of points for performance improvement:

1. Analysis can be based on input data “container (as containers are fixed in ISPs)” types, not actual “legal” input dynamic range, but can cause use of larger container than necessary, which degrade performance. If we know the input “legal” input dynamic range, we can use precise types in hardware .
2. Subword organization and shuffling can be locally analyzed based on task segments whose start and end subwords are of the same dynamic range. This can reduce the complexity of analyzing a long path by breaking it down to segments. If no such segments can be found, the analysis is done for the whole path.

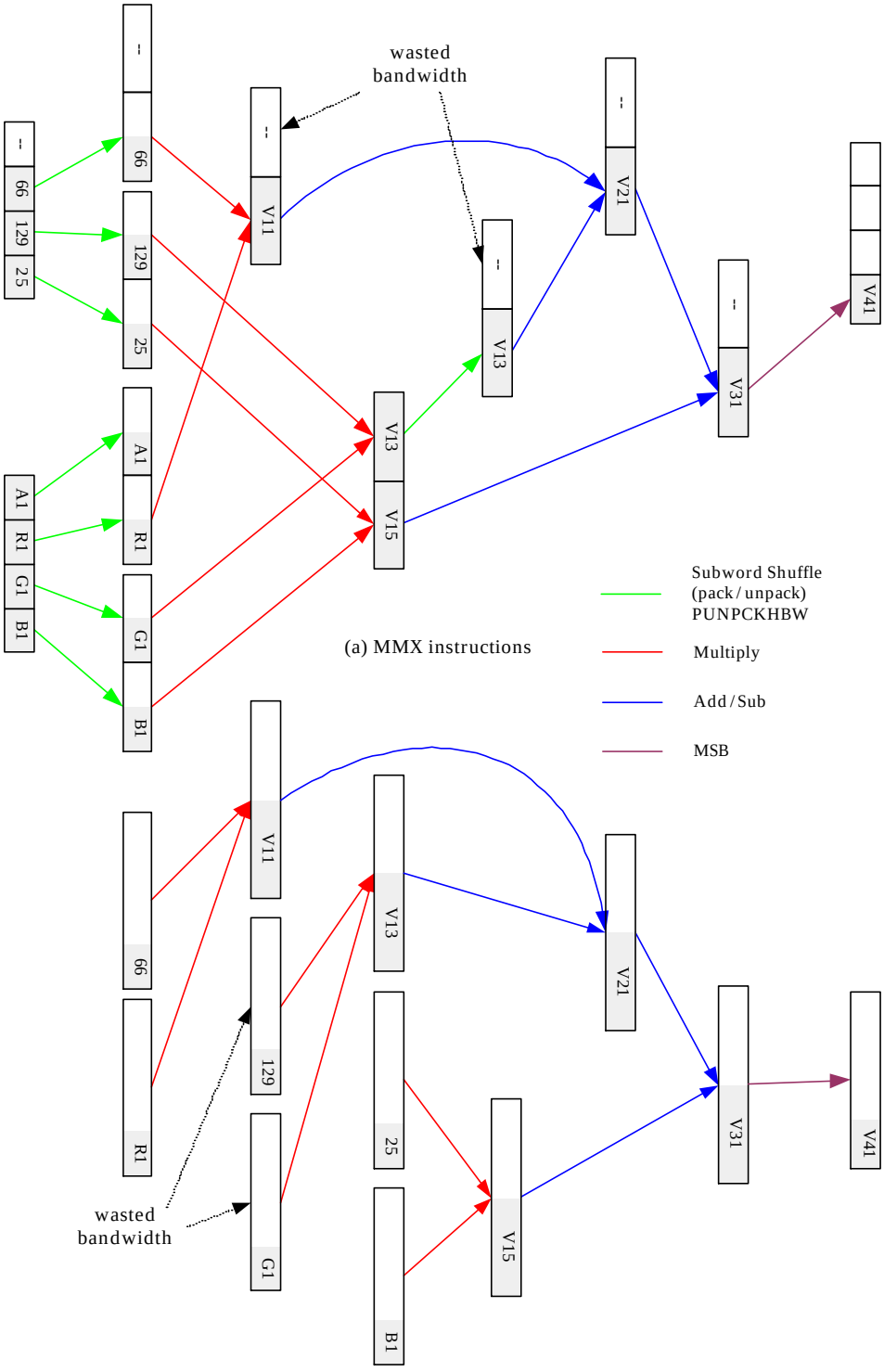


Figure 4-5: Code segment of color space transform represented as a sequence of three-register instructions on (a) MMX architecture (b) non-MMX architecture.

Figure 4-5 reveals three areas where instruction set processors can never circumvent:

1. Whether "segmented" or not, microprocessors datapath (registers and functional units) are "fixed-sized containers". Except for integer division (can be viewed as right shift operations), truncation (floor), and saturation (ceiling), the dynamic range of the output always grows for all other arithmetic operations. Traversing down the DFG, the dynamic ranges of variables is monotonically increasing requiring bigger and bigger data containers. What appears to be m-subword parallelism at the input can quickly vanish due to the growing dynamic range of intermediate variables. This is confirmed by studies on the SIMD effectiveness to performance improvement [28-30].

   For any instruction set architectures, even the ones supporting SIMD, only a small number of fixed data types (e.g. 8-bit, 16-bit, 32-bit, 64-bit) can be realistically segmented. Further division will turn the datapath into fine-grained bit-level functional units, which is what FPGA really is. At that point, the processor can no longer run efficiently for the regular types (8-bit, 16-bit, 32-bit, 64-bit). Therefore, there is always some wasted processing bandwidth for irregular data types. This problem is manifested when all the intermediate results are irregular types.

   For non-SIMD architecture, the bandwidth waste is more evident as shown in Figure 4-5 (b). Having only one type of data container prevents algorithmic innovation and performance fine-tuning. For example, one may not need 24-bit color when browsing a video archive for content. A small latency and a high frame rate are more desirable. An 8-bit color may be sufficient for content differentiation and can be processed much faster than a 24-bit color. To a non-SIMD architectures, reducing data precision doesn't increase performance.

2. Segment ALU cannot perform inter-subword operations. If such is desired, one must carefully (re-)arrange operand subword positions. We must carefully re-arrange intermediate subword locations so that results can be in the right subword locations for the next instruction. Not only programmers must pay great attention to this subword shuffling, but also it costs extra instructions whenever a re-shuffling is needed – performance hit.

3. Another observation from Figure 4-5 suggests the longer the instruction sequence is, the longer the latency and throughput are. Even for superscalar or VLIW architectures, the number of functional units are limited due to register file design trade-off [31]. Therefore, the performance degrades with long processing instruction streams. This means the more complex the task is, the worse the performance. In addition, for the SIMD capable architectures, this means more data shuffling between instructions. So far no compiler can optimize subword organization.

4.3.2 Reconfigurable Logic

On the other hand, let's look at how the same task can be implemented with

reconfigurable logic.

Figure 4-6 shows the code segment is translated into high-level functions, such as multipliers, adders, etc. At this point, this representation is all combinational. We can insert registers at the high-level functional boundaries. However, this would not "balance" the delays between these functions. It is desirable to break down the high-level functions into lower-level blocks. It is possible one macro function has more than one "child". Each child implements the parent at different costs (e.g. area) and different performances. We can traverse the hierarchy, breaking down each abstraction layer, to the lowest level of the reconfigurable logic – technology library [32].

Along the hierarchical path, there are many places where we can optimize the performance of the design by chaining functional blocks at each level and re-balancing the delays among them. We can insert more registers at lower level as there are more functional block boundaries. At the lowest level, we can insert as many registers as the resources allow and reduce the delays by "retiming" [33]. Operation chaining, pipelining, and retiming are three techniques ideal for dataflow task.

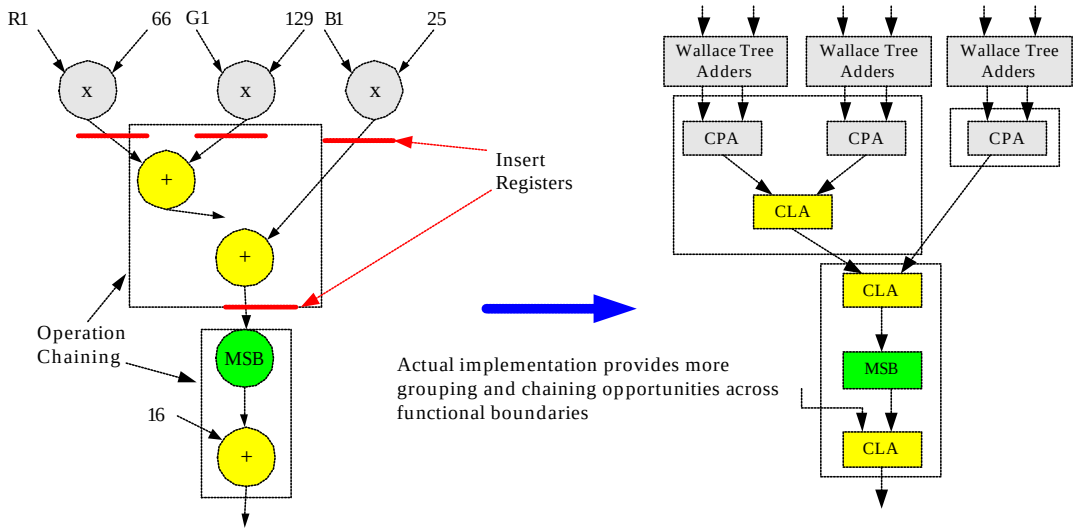


Figure 4-6: Code segments in Figure 4-4 translated into hardware macro functions.

Figure 4-7 shows that more optimization options are available at lower level of the design. To obtain better performance, this "architectural exploration" process is necessary.

Note that whether in Figure 4-6 or Figure 4-7, the latency is determined by the number of "pipeline stages". The throughput is determined by the longest delay between any tow successive registers.

In light of the three areas of weakness for microprocessor, reconfigurable logic can compliment them all. In particular, the following points correspond to ISP's areas of weakness:

1. The reconfigurable logic core has the same input data size and that the logic array is at least as wide as the input data size. Ideally, the should be twice (for intermediate data from multiplications) as wide plus some headroom for control. This is not a necessary requirement, but it makes better performance possible (chapter 5).
2. There is ample routing resources so that no longer paths between successive registers. However, there is a trade-off between routing resources and performance (and cost too, chapter 5).
3. The logic array should be deep enough to take in complex operations. However, there is a trade-off for cost and design time.

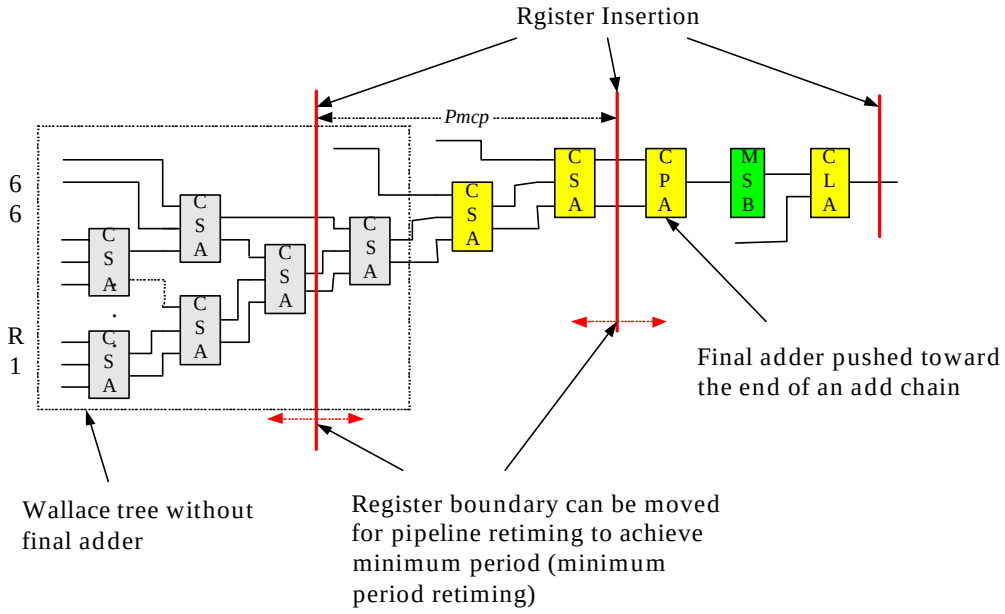


Figure 4-7: Pipeline retiming (or minimum period retiming) technique can be used to achieve high clock speed.

4.3.3 Coding Efforts

Turning scalar code into SIMD (e.g. MMX/SSE [34, 35]) vector code is not a straight path, and it is often necessary to rethink the algorithm from scratch when searching for optimal performance [28-30, 36]. The SIMD instruction sets can offer performance increases that range from zero to a theoretical m times. However, real applications have shown speedups in the tens of %, though that numbers for kernel functions are usually higher. Selecting a kernel function's boundary for MMX is also a tricky task. On one hand, small well-formed (no or little data expansion and misalignment) code segments can produce the largest kernel speedup at the expense of frequent function calls when put together in a master application. One the other hand, large code segments may contain ill-formed code producing small speedup using MMX.

These various difficulties serve as good arguments for reconfigurable hardware's potential for media processing. That is, it may be better off developing a reconfigurable application than an MMX application both for time and performance benefits. Taking one step further, it might be better to create an core architecture without MMX instructions but with reconfigurable logic. Not only can one simplify the design of such an architecture, but also improve the performance because the design of a segmented ALU and data packing and unpacking registers will require less logic.

From Figure 4-5 and Figure 4-7, we can calculate the latency and throughput in each case and decide whether this particular task should run on the ISP core or the reconfigurable logic core. It is easier to visualize this comparison by putting them side by side as shown in Figure 4-8.

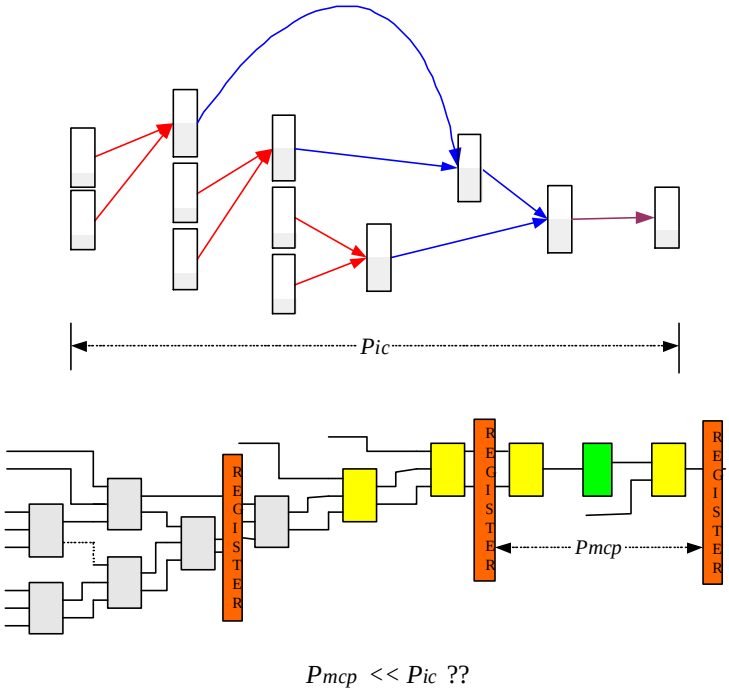


Figure 4-8: Instructions executed in sequence in the temporal domain vs. concurrent hardware execution in spatial domain.

Figure 4-8 suggests we can make somewhat quantitative performance comparisons if we can analyze the task and the implementation architectures, in particular the microprocessor's datapath and reconfigurable logic's pipelining limitations. What and how much information is needed (chapter 5 & 6)? What is the trade-off between routing resources and performance (chapter 5)?

5 Reconfigurable Architectures

The performance of reconfigurable logic depends on the applications to be implemented, the architectures of the reconfigurable hardware, the expertise of the designer, and the design tools[9]. In the case of developing applications on an existing reconfigurable computer, two more conflicting constraints add to the equation: a speedup as the performance constraint and the size of reconfigurable logic as the area constraint. With so many factors and constraints, how do we predict reconfigurable logic's performance?

In this chapter, we examine reconfigurable architectures with two most popular architectures, representative in their respective class. We discuss their logic compositions and routing structures. We also discuss the trade-offs between programmability and performance.

The requirement of programmability on reconfigurable logic makes non-deterministic routing delay unavoidable. The performance requirement affects the routability and thus programmability. Given this unpredictable routing delay, how do we estimate reconfigurable logic's performance?

We propose a "heuristic" approach to put a ceiling on the maximum clock rate of a reconfigurable device. We made minimum assumption about a real design. From there we calculate the minimum delay as the sum of a combinational delay in logic blocks and shortest propagation delays through the routing structures (switch and wire delays).

Finally, we argue that reconfigurable logic cannot achieve the same clock speed compared to a fixed-logic device fabricated with the same semiconductor. We propose a theory to make this claim.

5.1 Topology of Logic Cells and Routing Structures

Two general topological arrangements of logic cells and routing structures are most commonly seen.

5.1.1 Symmetrical Array

A symmetrical FPGA places arrays of identical logic cells and identical routing structures around each logic cells. Figure 5-1 shows a two-dimensional symmetrical arrangement of logic cells and interconnects. The symmetry is exhibited in the identity of each equal sized blocks partitioned by drawing imaginary, equally spaced vertical and horizontal lines between adjacent logic cells.

[9] This is due to the complexity of today's designs and designs to be done within a reasonable amount of time, not because human cannot do a better job.

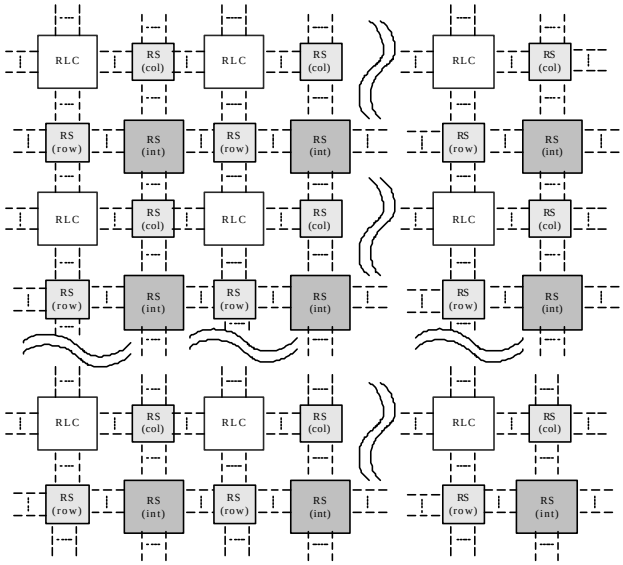


Figure 5-1: Topology and organization of a symmetrical FPGA

Any logic cell output to any logic cell input, point-to-point, connectivity is impossible as the routing requirement grows on an order of $O(n^4)$, where n is the number of logic cells in one dimension. Thus routing structures are typically restricted to neighbors or organized as a group to keep the growth of routing space linear, $O(n^2)$, with the number of logic cells . Still routing space usually takes up over 90% of the silicon space on a typical medium- to large-sized FPGA. One popular way to structure the routing is the segmented channel architecture as used by Xilinx's XC4000 FPGAs. A typical segmented routing structure consists of segmented wires of length 1, 2, 4, and longer[10]. Length is measured in terms of the cell-to-cell distance, or CLB-to-CLB distance. Figure 5-2 shows XC4000's topological organization of the single-length and double-length wires.

The horizontal and vertical single- and double-length lines intersect at a box called a programmable switch matrix (PSM). Each PSM consists of six pass transistors, representing all possible connection combinations among four line segments. There are single-length switch matrices in every row and column of CLBs and double-length switch matrices every two rows and columns[11]. Each switch matrices consists of programmable pass transistors used to establish connections between the lines. Single-length lines provide the greatest interconnect flexibility and offer fast routing between adjacent blocks. Short-length lines are not suitable for routing signals for long distances because it would require a signal to pass through many switch matrices to connect to a

[10] Quad-length line segments are offered in XC4000X only. XC4000X family also offers direct connection between adjacent CLBs.

[11] Similarly, there are switch matrices every four rows and columns for quad-length line segments in XC4000X.

far away net. Each pass through a switch matrix incurs a delay through the pass transistor. Therefore, short-length lines are normally used to connect signals within a localized area and to provide the branching for nets with fanout greater than one.

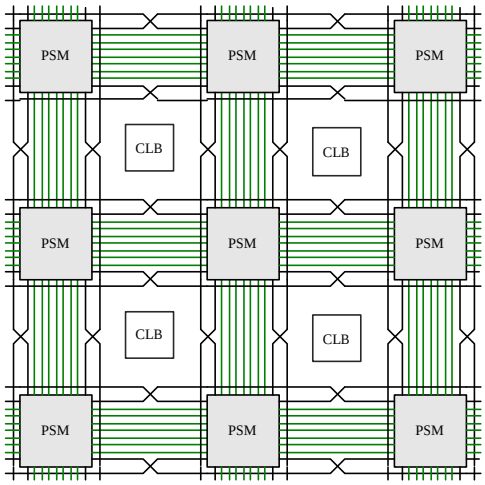


Figure 5-2: Xilinx's XC4000 single- and double length lines, with programmable switch matrices (PSMs)[12].

5.1.2 Hierarchical

The organization of logic cells and routing structures of a symmetrical array do not favor or "bias" toward one particular topological order or another. It is "generic' in this sense. A "hierarchical" organization groups a fixed number of logic blocks and routing structures into one level of hierarchy. Within a level of hierarchy, a fixed number of logic cells are physically placed closer in clusters on the silicon so that routing delays within the hierarchy are much smaller than those traveling outside the hierarchy. This fixed number is a kind of architectural "bias" or architectural specialization in anticipation of certain application characteristics, such as data type, in some market segments. As such, a hierarchical FPGA can offer slight performance advantages to applications with the same hierarchical structure but suffer some performance degradation to others of a dissimilar structure. One such popular FPGA family is Altera's Flex 10K family. Figure 5-3 shows the structure and placement of logic cells and interconnects.

[12] Courtesy of Xilinx Corporation.

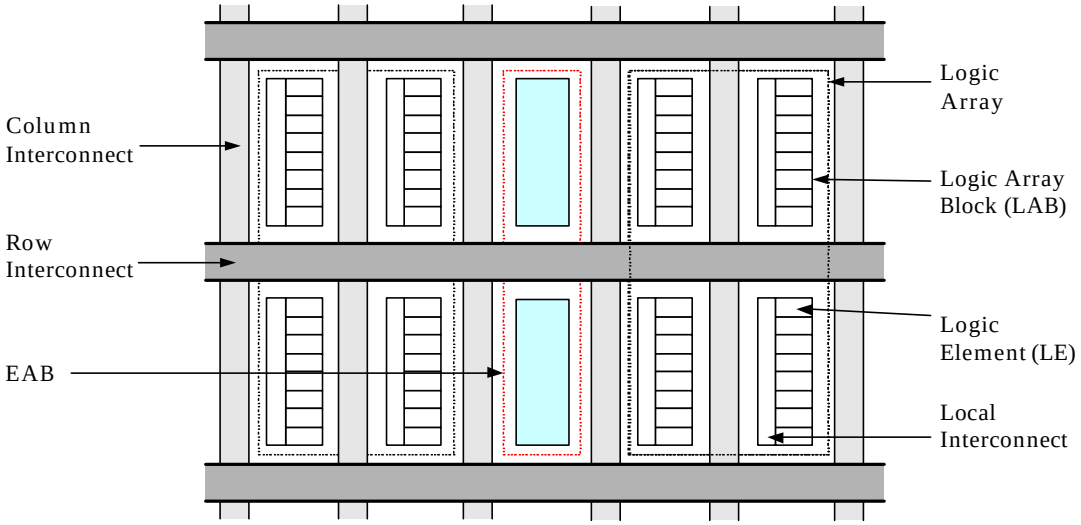


Figure 5-3: Altera Flex 10K architecture.

In the FLEX 10K architecture, connections between logic elements (LE) and device I/O pins are provided by the FastTrack Interconnect, which is a series of continuous horizontal and vertical routing channels spanning the entire device. Each row of logic array block (LAB) is served by a dedicated row interconnect. The column interconnects route signals between rows. For improved routing, the row interconnect is comprised of a combination of full-length and half-length channels. The full-length channels connect to all LABs in a row; the half-length channels connect to the LABs in half of the row. Both row and column interconnects can drive I/O pins. This global routing structure provides predictable performance, even in complex designs.

Figure 5-4 shows the details of routing structures between a logic element and row and column interconnect.

This routing structure is "anti-symmetrical" in that a row channel can be driven by an LE or by one of three column channels, whereas the column channel can be driven by an LE or one row channel. The switches in this case are multiplexers[13]. These multiplexers allow column channels to drive row channels even when all eight LEs in an LAB drive the row interconnect. A signal from the column interconnect, which can be either the output of an LE or an input from an I/O pin, must be routed to the row interconnect before it can enter an LAB. Access to row and column channels can be switched between LEs in adjacent pairs of LABs. This routing flexibility enables routing resources to be used more efficiently.

[13] A multiplexer is an active device that increases signal strength. Compared to a pass transistor, it can drive longer wires or it reduces the delay to the same length of wires.

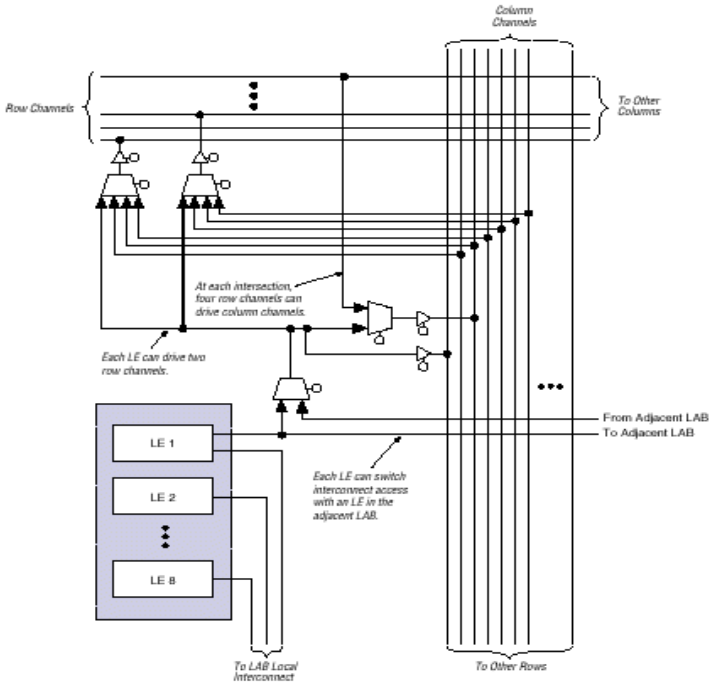


Figure 5-4: Altera FLEX 10K family's LAB connections to row and column interconnect.

5.2 Minimum Clock Period Estimation

In last section, we have shown the topological structures of logic cells and routing resources of two most popular commercial FPGA devices. While future reconfigurable devices surely will have different variations in its topology and organization in logic and routing structures. Once fabricated certain "intrinsic" properties are thus fixed. These intrinsic properties are determined by the semiconductor process technology and the particular topology, structure, organization, and circuit implementation of logic and interconnects. Without specifying an application in the form of a netlist of the actual target reconfigurable device, the exact benchmark of the reconfigurable device cannot be determined. However, from the intrinsic properties, one can obtain a "heuristic" performance upper bound, such as a minimum clock period, or maximum clock rate, for all non-trivial applications. CAD tools for reconfigurable designs should expose relevant intrinsic properties early in the design process to quickly rule out impossible kernel candidates for reconfigurable implementation. Tools should also provide a quick estimate of potential speedup. This potential speedup can be used as a speedup target to guide further refinement and optimization of a design.

In chapter 3, we argue that, for a reconfigurable device to have any performance advantages over a contemporary general-purpose microprocessor, it must implement an application of medium-to-large complexity, excluding bit-level processing commonly found in logic emulation. Medium complexity is loosely and relatively defined, from the viewpoint of an instruction set processor, as computation requiring several to tens of

instructions to produce an output datum per input datum.

By virtue of the above definition of a medium to large complexity design, a straightforward design specification in a high-level programming language, or a hardware description language, is impossible to automatically generate a high-performance implementation close to an achievable optimum with efficient utilization of the reconfigurable resources. Thus some expertise and hand optimization with progressively improved performance are always required to fully exploit the potential of a reconfigurable device. Depending on the application, this can mean a few man-days to tens of man-months. One must consider the investment in effort against the backdrop of possible improvement by re-structuring and possibly hand-optimizing a software counterpart[14].

Current compiler technology still cannot take full advantage of all the hardware features in a modern general-purpose processor that can help increase performance. One such example is the multimedia instructions in almost all of the current state-of-the-art microprocessor. This is due to the fact that a general programming language, by definition of being "general", does not contain application-specific syntactic constructs supported by special hardware features.

To obtain an upper bound on the maximum clock rate for a reconfigurable device, some assumptions and arguments must be made.

1. We know the timing models of logic blocks, wires, switches and capacitive loadings parameters when a reconfigurable device is manufactured. These pieces of data are available from the semiconductor foundry. Considering the possible integration of reconfigurable logic with a general-purpose core in the future, these pieces of information come with no cost.
2. We assume a realistic medium to large design contains a combinational datapath of several to tens of arithmetic and/or logical operations deep. Any combinational logic can be represented by a Boolean network or sum-of-products, which can be mapped to a network of k-input LUTs [37]. Performance-driven technology mapping with pipeline retiming (or minimum period retiming), such as the one described in [38, 39], can be used to obtain a minimum clock period. This step can be used as a successive step with finer target architectural details to generate greater accuracy since it requires more time to compute.
3. By either manual or automatic insertion of buffers, internal nodes, and pipeline stages in the data as well as in the control path, we can always achieve a design of throughput one. An equivalent statement is that we can always slow down the clock so that the critical path delay is smaller than one clock cycle. And we can insert equal number of registers on all combinational paths then retime them

[14] Re-structuring codes to follow certain coding styles that a compiler knows how to optimize.

to reduce the minimum clock period[15]. In this case, all combinational delays are at most one path delay from the input of LUT to the register. This, of course, is a pedagogical scenario assuming that the reconfigurable hardware always has enough resources for extra logic and routing needs due to buffering, pipelining, and replication of internal nodes[16]. The number of internal nodes in a Boolean network of n-inputs can grow exponentially making this impossible for large designs. However, this ideal scenario does serve as a strong lower bound on the minimum clock period, which represents the best-case scenario for comparison with a general-purpose processor.

4. At least some internal nodes have more than one fan-outs making at least one routing path cross one row and column boundary[17]. Figure 5-5 shows two routing delay paths with a fan-out of two. The longer delay is the one crossing the row and column boundary.

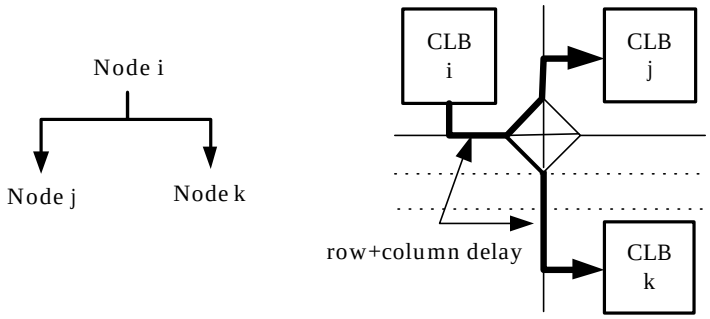


Figure 5-5: Assuming there exists a node with fanout of two in a real design.

5.2.1 Combinational Delay

We can estimate combination delays by summing up all the delays in the combinational path. For example, in the case of Xilinx's XC4000 FPGA, the combinational delay is the sum of all component delays on the combination path, shown as a blue path in Figure 5-6.

---

[15] For sequential circuits, such as a state machine, retiming requires derivation of new initial states for the sequential part. [40] describes a forward retiming technique to solve that problem.

[16] In fact, one can make an argument that area and speed trade-off is also true for reconfigurable hardware. Therefore, if performance is the goal, the utilization of reconfigurable resources is not likely high. Also, in general, the larger the chip, the longer the global routing delay will be. This seems to work against the previous argument. However, the "penalty" is not as severe as having to route a signal through more switches or LUTs.

[17] If all internal nodes have only one fan-out, then the combinational network becomes a tree or a forest.

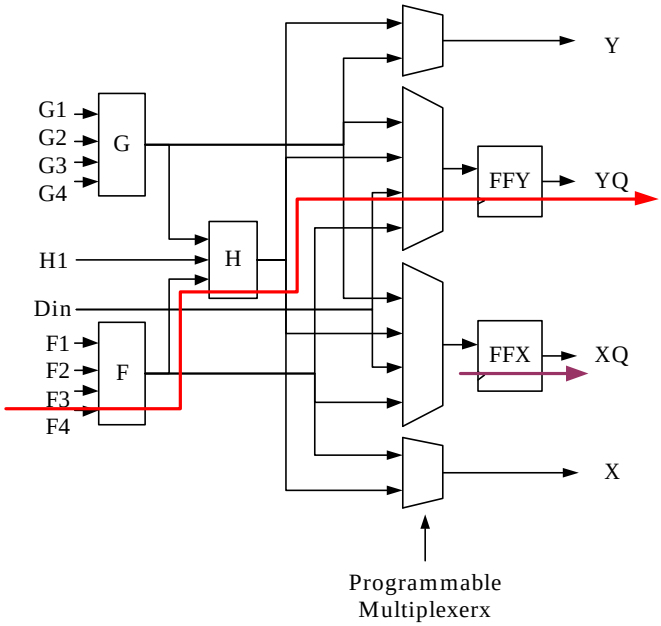


Figure 5-6: Estimation of combinational delay. Example uses Xilinx XC4000 family FPGAs.

The purple line shows the clock to output delay through the flip-flop. This delay will be added to the final total delay between two registers to calculate the minimum clock period.

Similarly, other logic cell architectures can be estimated this way. Shown in Figure 5-7, the red line is the combinational path delay of Altera's FLEX 10K family FPGA.

5.2.2 Routing Delay

Routing delays typically constitutes 40% - 60% of the total delay, which is much greater than that for mask-programmed gate arrays [41]. Three key factors affect interconnect delays in an actual design implemented on FPGA devices: the routing architecture of the chip, which comprises wires and switches interconnecting logic cells; the application in the form of a RTL description; and the CAD tools used to implement the application. FPGA routing (as well as logic cell) architectures varies due to practical constraints and some architectural specialization toward particular application segments. This phenomenon will continue to exist and continue to evolve. Each architecture imposes different structural and physical constraints that only dedicated CAD tools can generate quality results. Optimal technology mapping, placement, and route tools for a particular FPGA architecture, under different constraints, are often open research problems. Many of them are either shown to be NP-complete, no polynomial time solutions have been found yet, or no proof has been contemplated [42, 43].

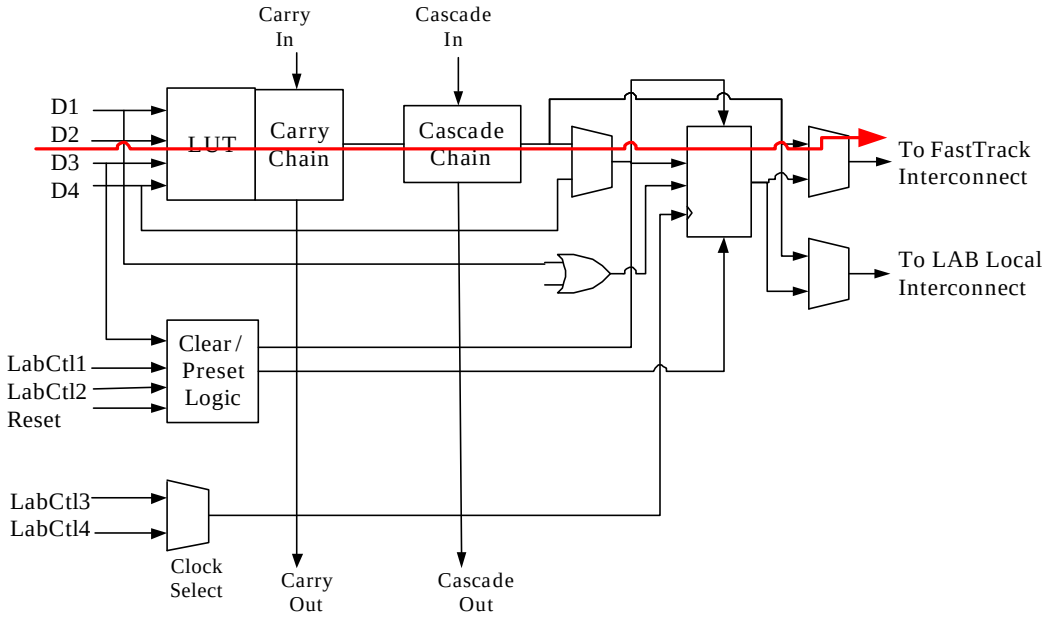


Figure 5-7: Altera FLEX 10K family FPGA combinational path delay.

Switches, wires and capacitive loadings cause signal delays. Accurate delay calculation is possible only after routing. However, one can obtain a lower bound, without specifying an actual application, based on previous assumptions 1-4. Figure 5-8 shows the minimum routing delay in a segmented channel architecture is due to three pass transistors and three segment length wire delays.

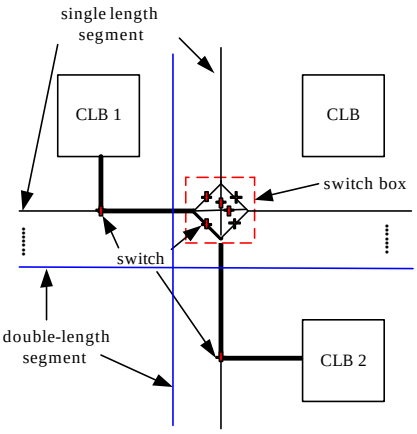


Figure 5-8: Estimation of a lower bound on routing delay (e.g. switch based routing structure).

Capacitive loadings will add some additional delay, but, as a "theoretical" lower bound on routing delay, this omission strengthens the bound. So delay due to three pass transistors and three single-length segments, available from semiconductor foundry as technology library, can be used as a (strong) lower bound estimate for the segmented

channel architecture.

Similarly, one can establish a lower bound of routing delay for the Altera's FastTrack routing architecture. In Figure 5-9, a signal traverses from a logic element in A to a logic element in B through a column then onto row routing channel.

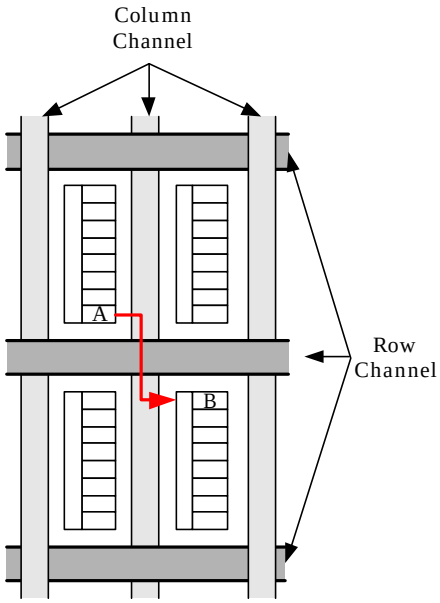


Figure 5-9: Estimate of minimum routing delay in Altera's FastTrack routing architecture.

Unlike Xilinx XC4000, multiplexers make up of the switches in FLEX 10K, as shown in Figure 5-4. The row and column routing channels are global. That is, they span the entire chip in width and length in each horizontal and vertical dimension. This means delays due to wire and capacitive loads are larger in large-sized FLEX 10K chips. XC4000 family also has long segments that span the entire chip horizontally and vertically.

FPGA architecture with global routing paths is the major reason that its minimum clock period doesn't scale well with the shrinking size of transistors if the size of the chip remains the same. XC4000 family also has long line segments that span the entire chip. If a design has high fan-out internal nodes, routing through long line segments, instead of a series of short segments, greatly improves performance [44]. The number of switches traversed by a long line is much smaller than the number of switches traversed by a series of short segments. However, long line segments can reduce the routability of a FPGA since they take up chip space, which can otherwise used for short line segments. Figure 5-10 shows the combinational and routing delays in FLEX 10K devices with the same speed rating[18]. The combinational delays are pretty much the same while the

[18] It is assumed the process technology that these devices were fabricated was the same.

routing delays increase with the size of the chip.

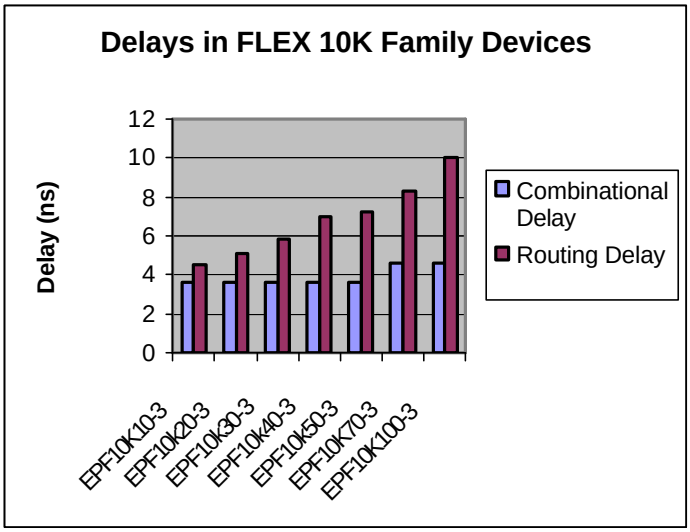


Figure 5-10: Delays in Altera FLEX 10k family devices (speed rating -3)

Figure 5-11 shows the routing delays as percentage of minimum clock period as the size of the chip increases. The comparison is based on devices made by the same process technology, a 0.42 micro, four layer metal process. The result is expected because FLEX 10K devices have global routing channels, which make global routing delay longer in proportion with the size of the chip.

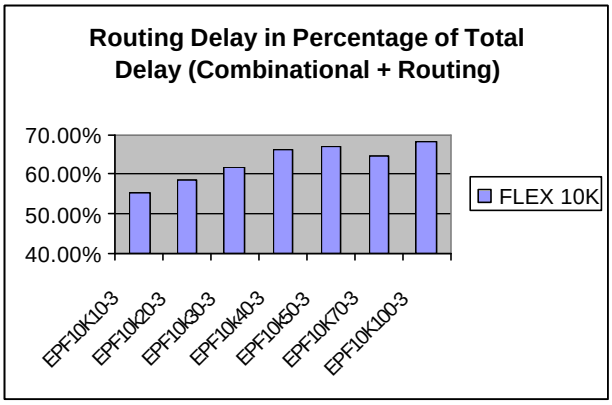


Figure 5-11: Percentage of Routing Delay in Minimum Clock Period increase with the Size of the FPGA.

Figure 5-12 shows the percentage of routing delay can increase if the chip size remains

the same and the architecture of the chip remains the same[19]. That is, logic and routing structures are simply replicated to make up the extra die area but global routing channels still span the chip horizontally and vertically.

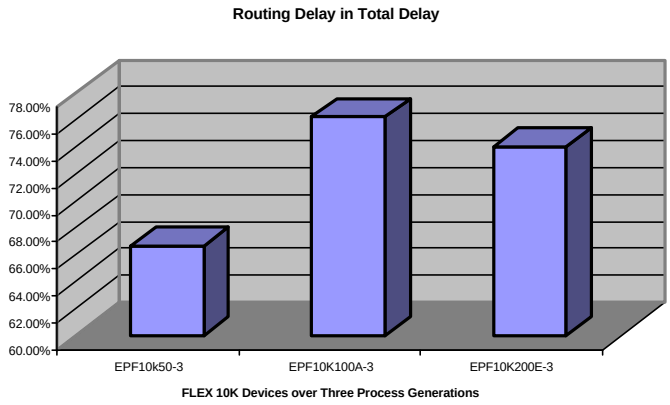


Figure 5-12: Routing Delay in FPGA Grow with Process Technology Upgrades[20].

As a transistor's size shrinks, one must reconsider this effect to various source of performance related issues. Individual delay components may change as well as its relative effect to the overall delay as formulated by Amdahl's law. New architectures or micro-architectures must reduce the overall delay to take advantage of the smaller feature sizes[21]. Routing delay is the largest and most unpredictable delay component in a FPGA design. It can affect the performance on the order of 100% or more, while combinational delay is completely predictable after technology mapping thus has less room for improvement.

This is not the case with ASIC or general-purpose processor since both routing and combinational delays in these devices can be reduced with technology upgrades, while FPGA must provide "general" routing structures that cannot be "optimized". Therefore, the lower bound on the minimum clock period, or the maximum clock rate of a

---

[19] Here we assume the chip size scales linearly with the number of gates on the chip and linearly with the square of minimum feature size. Since these three devices span three process generations (0.42, 0.35, 0.25), we assume that the chip area is the same for these three devices.

[20] EPF10K50-3 is made in 0.42 micron, four layer metal process; EPF10K100A-3 in 0.35 micron, five layer metal process; EPF10K200E-3 in 0.25 micron, optimized five layer metal process. This optimization contributes to the drop in routing delay in FLEX 10KE family devices.

[21] Large FPGAs break up the chip-wide global routing channels into half-sized routing channels. However, this creates more complexity for the routing tools. Tools often need to be modified or created to fully utilize to architectural or micro-architectural changes in new FPGAs. In fact, architectural changes can cause a domino effect in the tool chain that the whole methodology needs to change. This has been a major problem for the FPGA users since software tools have been far behind FPGA's increasing versatility and complexity.

reconfigurable device, has not followed the Moore's law as the general-purpose processors have. This disparity in the maximum clock rate between a general-purpose processor and a reconfigurable device can grow wider given the same process technology and the chip areas unchanged. An examination of performance improvement on these two computing media over the last decade supports this claim.

In 1991, the state-of-the-art microprocessor ran around 33 MHz. An FPGA (Xilinx XC3090) could run as fast as 33 MHz as reported in [45]. Today a Pentium 4 processor runs as fast as 1.7GHz (as of June, 2001), a fifty-fold increase over ten years ago. A survey on the performance claims in the Altera's Megafunction library reveals top speeds reaches only around 200 MHz. Even taking a most optimistic estimate on ideal designs from Xilinx Virtex-II devices' data sheet, the top speed is around 400 MHz, only a ten-fold increase since 1991. These numbers are not exact, but should provide a good comparison basis. This disparity is a very important factor to consider for reconfigurable computing - whether current reconfigurable architectures can maintain their performance advantages over general-purpose processor as semiconductor process technology keeps moving forward; and whether new reconfigurable architectures, especially, the routing structures, are needed. Indeed, from the above observation and analysis of the current FPGA routing structures, faster interconnects are necessary to keep pace with heavily optimized interconnects in microprocessors[22].

5.2.3 Clocking: Reconfigurable logic vs. Fixed Logic

Why configured FPGA seemingly cannot run as fast as a microprocessor fabricated with the same semiconductor processes? We offer an explanation from two perspectives.

1. Microprocessors have more optimization options at all levels of the circuit, while FPGA does not.
2. The majority of silicon area on FPGA is dedicated to routing structures, the average "distance" between registers is longer than that of a microprocessor.

All logic definitions on the microprocessor are known at design time, therefore, the following can be optimized to the design specification [46]:

1. Pipelining can reduce combinational delays between registers.
2. Retiming can balance combinational delays on a path so that the dynamic range is small.
3. Custom gates can be designed to increase or decrease drive strength according to actual output loads.
4. Fabrication process can be tweaked (e.g. doping level, metallization) for further optimization.

[22] Xilinx Virtex-II devices have replaced the pass transistors with active buffers to reduce delays.

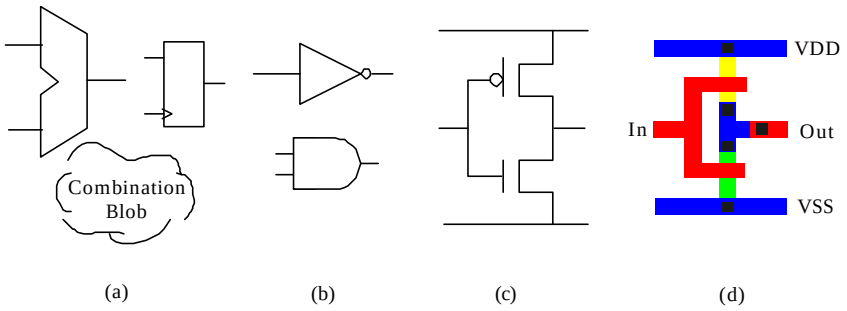


Figure 5-13: VLSI design hierarchy (a) register transfer level (b) gate level (c) circuit level (d) physical layout

In addition, we can attack this question from a different angle. The "average" combinational delays between registers is somewhat correlated to the register density (number of registers per unit area). In the case of reconfigurable logic, we must consider the actual "active" registers, the ones that are clocked in the configuration. Figure 5-13 (a) shows all registers in a fixed-logic design are clocked, while not all registers in a RLA can be used. To summarize:

1. The ratio of raw register density of (a) vs. (b) is conservative. The majority of FPGA's area is dedicated to routing [7, 47, 48] [49].
2. The active register density of (a) vs. (b) is higher than their raw register density ratio, suggesting greater distance between active registers in (b).
3. Paths in (b) can have additional delays due to switches, muxes, etc.

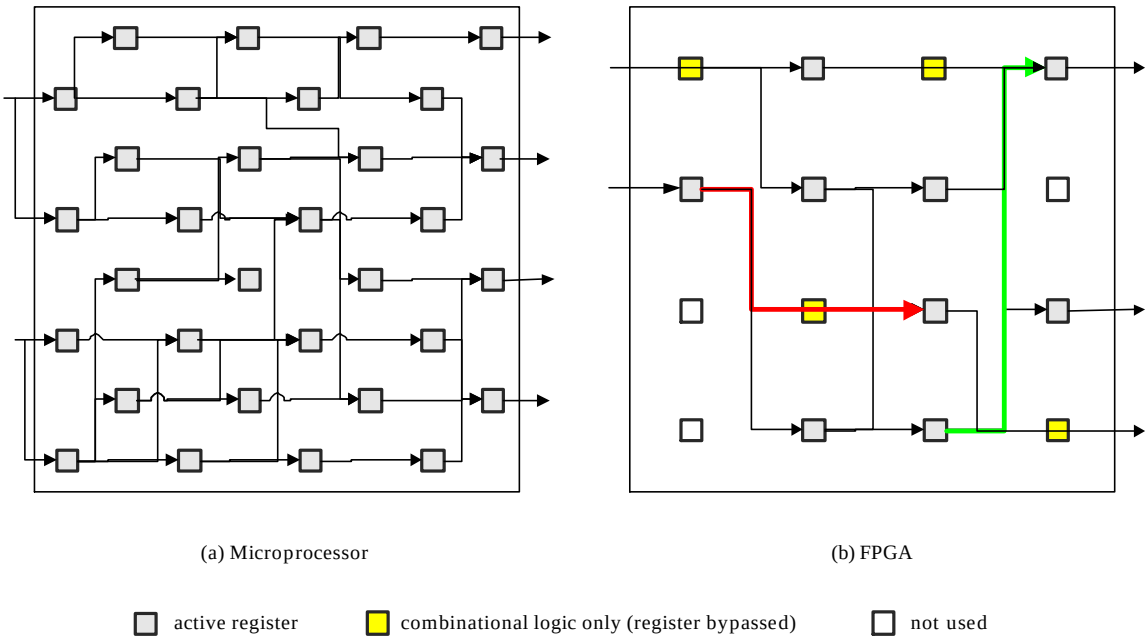


Figure 5-14: Register packing density (a) every register in a microprocessor is active (b) not every logic cell is used for every configuration, some are used for its combinational part only.

6 Performance of Microprocessors

From Figure 3-3, we see a microprocessor plays the most important role in the complex interactions among performance contributors. How do we evaluate a particular processor's performance given a task? Figure 4-5 suggests we can calculate it through analysis of instructions execution and properties of microprocessors' datapath.

In this chapter, we examine the factors affecting microprocessor's performance. We point out that given the same fabrication processes, microprocessor performance improvements come from three areas: instruction supply, data supply and instruction processing. These areas translate into three problems: branch prediction, memory bandwidth, and effective data flow.

We focus on microprocessors' ability (or inability) to maintain effective data flow because that is the area reconfigurable can be advantageous. Therefore, we are interested in the data processing rate, or the throughput of the functional units. A relationship between the data processing rate with the better known "instruction rate" can be established if we know the proportions of the data processing instructions in a task.

We then visualize instruction execution as instruction flow through a two-dimensional array of functional units. The one dimension represents the multiplicity of functional units, while the other dimension is time.

We then specify a set of variables associated with the properties of microprocessor datapath. This set includes the number of functional units, their operation and operand types, the latencies and throughputs and clock speed. Together with the instruction execution model, we can calculate the execution time.

6.1 Performance Analysis

The broad area of performance evaluation can be divided into the three categories of performance measurement, analytic performance modeling, and simulation based performance modeling. The first category, performance measurement refers to measurements done on actual hardware such as a test board or actual system box. Simulation modeling typically uses software to simulate the processor or system and predict performance. An execution based simulator simulates execution of the actual design, generating program results and essentially runs code much the real hardware would.

Both performance measurement and simulation are too expensive propositions for reconfigurable system and/or application development. In the former case, we have to build the system and/or develop the application, and risk our efforts being fruitless. In the later case, we need to build processor behavioral models and essentially completed the reconfigurable logic design. That requires significant investment in time too. Both methods require substantial amount of time and speedup is not guaranteed.

We must look for ways to evaluate speedup candidates that give us good enough assurance to warrant further development. It seems an analytical approach would avoid the expensive cost of the trial-and-error approaches. However, modern microprocessors are very complex and include a wide range of architectures. How much information to we need to analyze a microprocessor's performance? We want to keep this to a minimum while still being able to include all architectures.

Fortunately, we can take advantage of media processing characteristics and concentrate on the datapath of microprocessors. As Figure 4-5 illustrates, we should start with instruction counting.

6.1.1 Instruction Rate

In general, microprocessor instructions can be subdivided into the following categories: integer instructions (INTI), floating-point instructions (FPI), load and store instructions (LSI), flow control instructions (FCI), processor control instructions (PCI), memory synchronization instructions (MSI), memory control instructions (MCI), and external control instructions (ECI) [35].

Instructions can also be dichotomized into three-address instructions (TAI) and the rest. Three-address instructions involve moving data in general-purpose registers (or SIMD registers) to other registers, possible via functional units, or cache, and vice versa. We can further divide three-address instructions into data transfer instructions (DTI) and data processing instructions (DPI). In other words, DTI is equivalent to LSI, while DPI is INTI plus FPI in our terminology.

A "datapath-intensive" task involves more TAI than the rest of instructions. This ideal task for reconfigurable co-processing is data-intensive, has long runs of TAIs, has little data dependency. We can re-order instructions to make longer runs of TAIs, creating more speedups. Figure 6-1 shows how we separate instructions and look for opportunities for reconfigurable co-processing.

In addition, we can "schedule" these instructions, find a minimum schedule (if possible), and compare it with a fully pipelined reconfigurable function (4.3).

A performance comparison of microprocessors using MIPS or MFLOPS gives us basis to make general statements for all applications. A more accurate comparison can be made once the target application is specified. By running the actual application on the target system, one can obtain program execution time, which is a product of the following three factors.

The most accurate microprocessor performance metric for a particular program is the total execution time. Here we extend the meaning of program to include functions and basic blocks. The program runtime is the product of instruction count, clock per instruction, and clock period.

$$\text{Program runtime} = \text{Instruction Count} * \text{CPI} * \text{Clock Frequency}$$

$$T = N * C_I * F_p$$

where T is the program runtime, N instruction count, C_i clock periods per instruction, and F_p clock frequency.

We can also gauge how well a program is executed on an microprocessor by instruction processing rate.

$$\text{Instruction Processing Rate} = \frac{\text{Instruction Count}}{\text{Program runtime}} = \text{IPC} * \text{Clock Period } (C_p)$$

$$I_R = N/T = I_c * C_p$$

where I_R is instruction processing rate (instructions/second), I_c number of instructions per clock period, and C_p the clock period.

Clock frequency (C_p) or clock period is largely dominated by a transistor's switching speed, which is turn, depends largely on the implementation semiconductor technology. Instructions per cycle (IPC) indicates how many instructions an ISP can execute per cycle – a measure of its superscalar capability. Whether IPC or CPI, they reflect an ISP's architectural or micro-architectural capabilities, compiler's ability to explore those capabilities, and a program's characteristics to take advantage of those capabilities.

An instruction stream can contain instructions involving the functional units, or not involving any functional unit of a processor. Here our definition of a functional unit denotes a notion of data transformation. In other words, it involves arithmetic, logical, shift, rotation, etc. The rest of the instructions, whether it is branch, load/store, memory synchronization, cache management, processor control etc., we call it control and data moving instructions.

We can view a running program as a stream of instructions feeding an ISP. This instruction stream can be split into the data processing instruction stream, and the control and data moving stream as shown in Figure 6-1. Figure 6-1 also shows that processor architecture affects the data processing rate, and that system organization (cache and memory) affects the load and store instruction rate.

We define instruction throughput to be the data processing rate, or data processing instructions per second. A program may contain very different ratios of data processing instructions versus control and data moving instructions. For media processing, we argue that the data processing instructions take up a majority of the total instructions. In addition, media processing instruction stream often consist of large blocks of data processing instructions interspersed with intermittent control instructions.

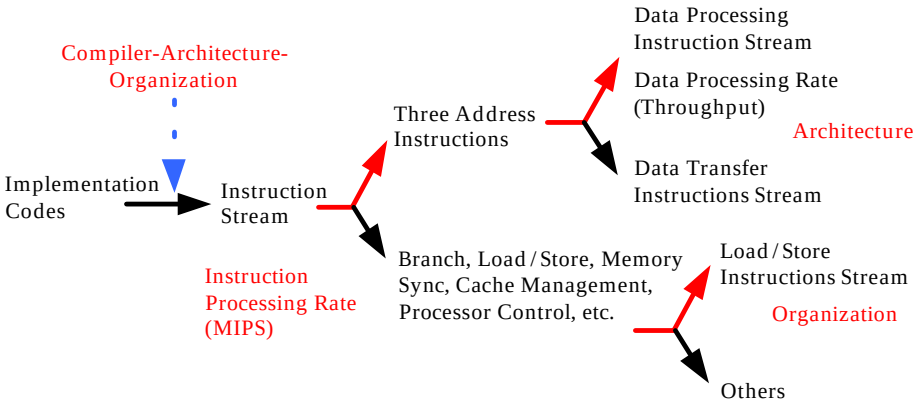


Figure 6-1: An instruction can be separated into data processing stream and control and data moving stream.

6.1.2 Factors

No mater what performance evaluation methods we choose, a microprocessor's performance is affected by the following factors:

1. **Semiconductor Fabrication Technology**
   Advances in semiconductor processing technology have been the major contributor to the superior performance of the latest generations of processors. Semiconductor processing technology has enabled the size reduction of transistors by half every eighteen months over the last two decades as predicted by the Moore's law. Shrinking minimum feature sizes translates into reducing gate delays thus increasing maximum clock speed. This has been the major source of performance gains in computing devices thus far.
2. **Architectures**
   VLIW and superscalar increase CPI, SIMD reduces instruction count.
3. **Micro-architecture**
   - Pipelined datapath increase clock speed (smaller C_p)
   - Dynamic register renaming to eliminate WAW pipeline hazard, increase CPI
   - Branch Prediction (ignored for multimedia applications because of low BP)
   - Out-of-order execution increase CPI

Among different architectural families but fabricated with the same generation of technology – micro-architectures affect scalability with technology process upgrades, thus indirectly affect the maximum clock rate. In addition, the degree of superscalabilty of each architecture directly contributes to the MIPS and MFLOPS numbers.

6.1.3 Speedup

The performance gains that can be obtained by improving the performance of some

portion (let's call it A) of an application can be calculated using Amdahl's law. Amdahl's law states that the performance improvement to be gained from using faster hardware for A is limited by the fraction of the time the hardware can be used. Specifically, Amdahl's law can be formulated as the following:

$$SU = \frac{1}{(1-F) + \frac{F}{S}}$$

Speeding up the "critical block", which takes up the largest percentage execution time, may not produce the largest speedup. Other blocks may take smaller percentages of execution time but have much individual speedups. This could result in a higher overall speedup. Therefore, picking the "optimal" block should be considered first.

In addition, we may not be able to put a speedup candidate on the reconfigurable logic core (RLC). The task may be too big to fit on the RLC. Then we can look for other opportunities. On the other hand, the reconfigurable logic core may have ample resources to accommodate more than one block. It is possible to integrate several blocks into one. Therefore, choosing a SW/HW partition that meets all constraints and performance goals is always an iterative process. We argue that obtaining as much performance possible requires a significant amount of effort.

6.1.4 SIMD

The most direct support to media processing is the addition of multimedia instructions. SIMD, coupled with segmented ALUs, explores subword parallelism. However, SIMD suffers the most when the task requires lots of data re-alignment between successive instructions and the subwords grow out of current subword sizes (4.3).

Figure 6-2 shows how we visualize the SIMD datapath. Note that the apparent subword parallelism at the input may not be fully realized due to growth of subwords' dynamic ranges during the entire task. Segmented ALUs use saturation arithmetic, where any operations on the subwords do not carry into the next neighboring subwords. Figure 6-2 (a) shows that the full subword bandwidth is utilized, if the dynamic ranges of all subwords stay unchanged. However, this is hardly the case for few real-world applications. Repeated saturations on the intermediate variables will accumulate and propagate producing undesirable results. Figure 6-2 (b) shows that the full subword parallelism cannot be realized because multiplications or any variable whose dynamic range "grows" with processing stages (shown are segmented multipliers of 2x precision of subword operands).

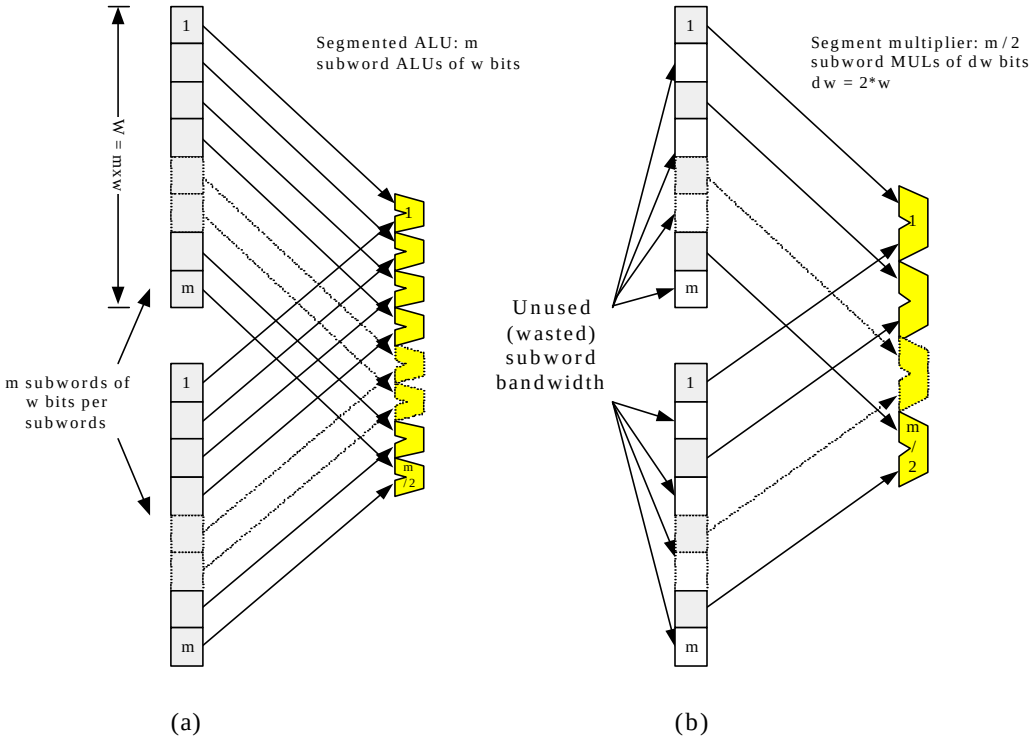


Figure 6-2: (a) Segmented (subword) ALUs for saturated arithmetic, (b) Segmented (subword) ALUs for arithmetic, logical, shifting, and any other operations

Table 6-1 shows the nomenclature used in this thesis and the Intel's. Note that we use *m* to denote the "degree" of subword parallelism. An m-subword parallelism means there are m subwords in a word.

Table 6-1: Nomenclature of "packed" data in SIMD context. Numbers represent the number of bytes in the data.

This Thesis		Intel	
Word	2^m	Quad word	8
Subword	2^{m-1}	Double word	4
	:	Word	2
	:	Byte	1
	1		

In section 4.3, we discover that, even with SIMD instructions, a microprocessor still cannot take advantage of all parallelism in the task. For example, Intel's MMX technology pack 8 bytes into a quad word. Ideally, there is 8 degree of parallelism. For the reasons discussed in section 4.3, only it cannot be exploited by a fixed-logic design.

We compile a few research results to show this is the case. Table 6-2 shows that the

overall performance gains are less than the highest kernel gains. In one case, the overall gain is negative, affirming the belief that programming with SIMD instructions requires much of knowledge of the hardware, or one may not produce a desirable result.

Table 6-2: Reported speedups using MMX instructions for some multimedia applications.

Applications	Overall Gains	Kernel Gains	References
MPEG-I Decoder	15%-25%		[50]
MPEG-II Decoder	40%-50%	50%-250%	[50]
H.263 Encoder*	67%	5%-70%	[28]
JPEG**	-50%	60%	[29]

(*: Compare hand-optimized scalar code vs. hand-optimized MMX code. However, some MMX kernel functions, e.g. DCT/IDCT were coded in much less precision than the scalar code. Therefore, the speedup should be taken as optimistic. **: Negative speedup is due to repeated MMX library calls to functions operated on 8x8 blocks. In other words, the application is not globally hand-optimized.)

Table 6-2 also suggests that there is room left for speedup opportunities from reconfigurable logic. This claim is based on the analysis in Figure 4-5 and confirmed from data in Table 6-2. All these kernel functions have 8-fold data parallelism in the memory word, but were only able to much less that half of it.

6.2 Datapath Performance Analysis

In section 4.3, we describe how we can derive performance estimate with the knowledge of the microprocessor. How do we formalize it?

First we need to identify critical parameters and architectural points that affect an ISP's datapath performance. We hope to extract a minimum set of such parameters and use them in our performance bound estimation and speedup opportunity exploration, and integrate these steps in a large framework for reconfigurable application development.

We start by making generalization on the datapath, in particular the register file and functional units, of a microprocessor. Our generalization is based on an observation that ISP datapath designs converge into a few good architectural points [51-53]. These points include multiple functional unit data paths (FUDP), separate register file for integer and floating-point FUDP, and pipelined FUDP.

6.2.1 Datapath Properties

Figure 6-3 shows a simplified ISP FUDP with only one functional unit attached to the register file. A FUDP consists of a function unit plus two register read ports and one register write port. The inputs (operands) to the FUDP come from either Load/Store unit (i.e. memory) or its own output from previous cycle.

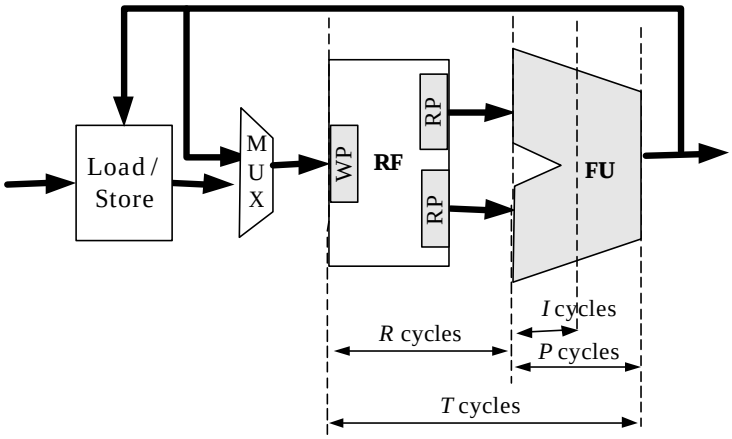


Figure 6-3: A single pipelined functional unit connected to a register file.

The functional unit is *P*-stage pipelined and can process data every *I* cycles. We call *I*, the initiation interval. We allow the register file to have a small latency of *R* cycles from write ports to read ports. For most ISPs, *R* is zero. The total latency of the pipeline is *T* cycles, $T=P+R$. This total latency is defined as the number of cycles from a data entering the write port to a dependent result written back to the write port. We call *T* and *I*, the pipeline properties of a functional unit.

A simple cycle operation has $P=I=1$, and a multi-cycle operation has a $P=I$ not equal to one. A pipelined operation has *P* divisible by *I* but $P>I$. A throughput of 1 operations has $I=1$.

In reality, a FU can have multiple pipeline latencies and initiation intervals. It can execute multiple sets of operations, or the same operations but on different operand sizes, with fixed but different latencies and initiation intervals. We call a group of operators with the same latency a latency group (LG). We call this grouping as latency grouping. We represent the pipeline latency and initiation interval as P_l and I_l respectively, where *l* is the *lth* latency group sorted from the smallest *l*.

Some integer units are designed to execute different arithmetic operations with different latencies. Some execute the same operations but with different latencies for different operand sizes. Typically, this is to reduce delay for small size data. We call it a multiple-latency integer (MLIU) unit. For example, PowerPC603 has only one integer unit for arithmetic, logical, and multiplication. Multiplication requires more cycles to

complete. This means it is multi-cycled as a multiplier. Alpha 21164 has two integer units, one of them is MLIU and one is single-latency integer unit (SLIU).

We assume each FU has two read ports from the register file and one write port to register file. We do not know of any architectures that have multiple FUs sharing the same read ports and write ports because this will cause port content and defeats the purpose of providing multiple functional units to the same register file. Two register file read ports, one register file write port and one FU completes one datapath pipeline.

In actuality, superscalar or VLIW processors all have multiple functional units and most have multiple register files according to data types. The register files may be connected to multiple functional units at the same time, but each functional unit has its own two read ports and one write port.

Figure 6-4 shows an multi-FU datapath connected to the same register file. Typically the functional units sharing the same register file operate on the same data representations. For example, integer and float-point units have separate register files. They may implement some different of arithmetic or logical operations with many other overlapping operations. Each functional unit may have different total pipeline latencies depending on the operation and operand size.

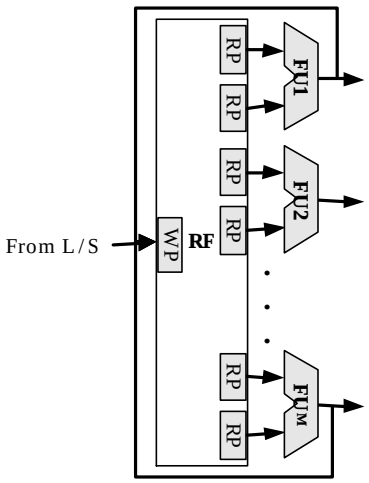


Figure 6-4: Multiple functional units sharing the same register file.

Sharing the same register file lets the member functional unit pipelines obtain previous results so that register moves (incur overhead) between register files can be reduced. For example, typically, integer functional units (ALUs and multipliers) share the same register file. Floating point, and MMX data type use separate register file. An arithmetic ALU pipeline can use a result from a multiplier pipeline for multiply-accumulation and vice versa.

It would seem that connecting all types functional units and as many of each one type to

the same register file would reduce unnecessary traffic between register files. However, we have a very different data representation for integer and real numbers, represented as floating-point (FP) numbers. Integer and floating-point representations cannot be converted into one another without conversion cost. In addition, their dynamic ranges are different. It is possible to lose data precision through conversion. Therefore, all architectures keep separate integer and FP FUPs.

Even operating on the same type of data, the number of functional units connected to the same register file cannot scale indefinitely without hitting register bandwidth and subsequently memory bandwidth limits first. In addition, it has practical implementation drawback on the performance of register file [54].

General purpose registers serve as variable cache for the functional units it connects to. Each functional units can consume up to two variables as its operands. Therefore, the number of (general-purpose) registers in a register file must be twice the number of functional units, i.e. the same number as the read ports. However, this minimum is far from enough as it implies only two live variables (or one live variable and a constant) can be kept per functional unit processing pipeline. Real applications simply have more live variables and/or constants most of the time. These extra live variables will "spill" (register spilling) over into memory (most likely cache first). Get data from or sending data to load/store units will incur extra cycles (either cache or memory) compared to getting data from register file.

Therefore the number of registers in a register file, N_r, should scale with the number functional units, M, or we create a separate register file to accommodate more FU. The formal case makes register allocation and scheduling more difficult. The latter case creates register move overhead when variables are stored in other register files. Modern microprocessors are carefully designed that register bandwidth can sustain the processing bandwidth.

Increasing registers in a register file brings up an undesirable performance penalty to the read/write ports. Whether they are implemented as buses or decoder/mux pair, the high fanouts will have an impact on the capacitive loading, possibly making it a critical path [51].

6.2.2 Data Processing Execution Model

We can view a stream of data processing instructions as temporal expansion of our FUDP model. Chained together, it can be viewed as a N-stage pipeline with different pipeline properties. For example, an temporal expansion of a block of N data processing instructions is shown in Figure 6-5.

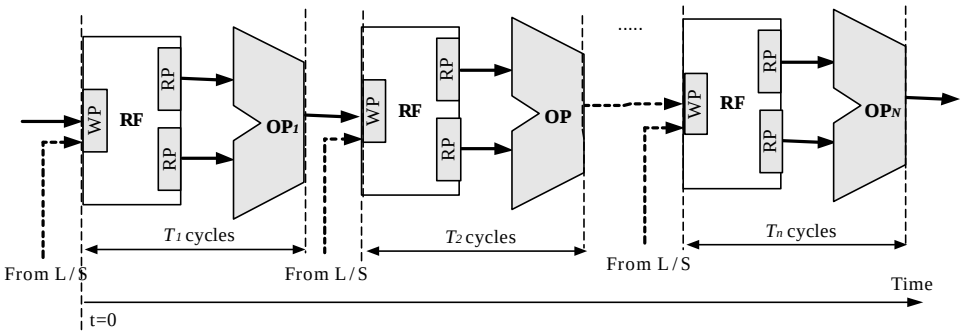


Figure 6-5: Temporal expansion of *N* data processing instructions.

Note that this view implicitly implies each of these n instructions (except the first) are dependent on previous instruction. Therefore, it can be viewed as how a 100% sequential block of instructions will execute on this FUDP. We can calculate the total number of cycles (sum of all T_i) assuming that all data are available before each instruction executes.

This temporal expansion conceptualization also provides us a comparison basis to an equivalent spatial arrangement of these operations on RLC in register transfer style. We can remove the registers and make further optimization through behavioral optimization of arithmetic and pipeline retiming.

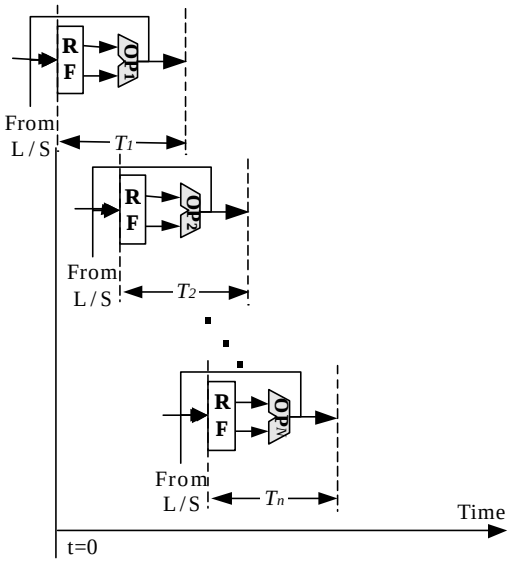


Figure 6-6: A block of *N* instructions execute on a pipeline functional unit.

However, Figure 6-5 does not reflect the pipeline opportunities when successive instructions are independent and of the same latency group. A more accurate picture is shown in Figure 6-6. Note that due to pipelining, successive instructions executions

overlap in time.

Similarly, a temporal expansion of *N* data processing instructions on superscalar or VLIW processors lead us to the following conceptualization.

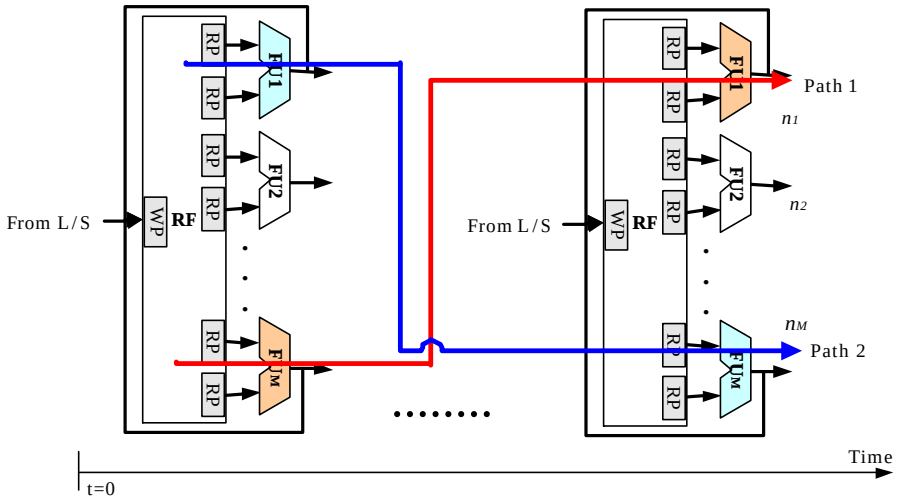


Figure 6-7: Temporal expansion of multiple FUDP. *N* instructions may be distributed among *M* functional units.

Path 1 and Path 2 represent two possible parallel routes for data to flow through the temporal expansion pipeline. At each register file to FU boundary, there may be multiple choices of routes if the next instruction can be executed on more than one FU. At the end of the paths, two variable are generated. Though not shown due to the difficulty of diagramming, there may be pipeline opportunities along both paths. The colored functional units execute instructions along the dependency graph. Uncolored functional units are not used because they don't support the instructions or they execute slower than other functional units for the same instructions.

Having multiple functional units avoids forced serialization due to resource constraints. It is equivalent to creating false dependency. Figure 6-7 shows that the paths may look more like the original computation structure, whether is a single DAG or multiple independent DAGs.

We can follow Path 1 and Path 2, chain up the whole register transfer paths and compared it to an implementation on reconfigurable logic core (Figure 6-7).

6.2.3 Performance Properties

So far, our performance modeling require only the following parameters

- M : # of functional units attached to a register file
- L_i : latency groups, [(operator group, operand size), latency]

- I : initiation interval
- T : pipeline latencyC_p : clock period, or $1/F_c$

In addition, we know under performance trade-off and limits of physics:

- M is a small number, usually 2-3 per register file, and cannot scale indefinitely [54].
    - RF performance will suffer because it must be able to broadcast to more recipients (constraints of physics) – bus, switch.
    - Will suffer "register spilling" if the number of registers not increase accordingly, then we have the same problem as above.
    - Memory cannot keep up.
- L_i are small. L^i is at most the number of (operator, operand size) pairs for each FUDP. We model a functional unit having multiple latency group if it is one of the following two:
    - FU performs different functions (+, -, *, etc.) at different costs (powerPC 603 [55], Alpha21164 [56]). We know operators such as +, -, &&, | |, shift, etc. are often grouped into one latency group, and *, % are in separate latency groups.
    - FU perform the same functions on different operand sizes at different costs (603 [55], 604 [57], alpha 21164 [56]).

  We model this property by [(operator group, operand size), latency] pairing.
- Pipeline stages cannot be infinite (T/I relatively small) before other components become bottleneck. Pipelining is to increase throughput by reducing clock period. However, register and memory bandwidth must keep up with the processing bandwidth.

Table 6-3 list three popular microprocessors, whose performance properties are reported in the literature.

Table 6-3: Performance properties of three popular microprocessors.

Processor	PowerPC 603		PowerPC 604			Alpha 21164		
Functional Unit	IU1		IU1	IU2	IU3	IU1	IU2	
Operator	+/-	*	+/-	+/-	*	+/-	+/-	*
$T_{i,l}/I_{i,l}$	1/1	4/2	1/1	1/1	4/2	1/1	1/1	8/4

7 Architecture of Reconfigurable System

How should hardware reconfigurability be added to a general-purpose system? What should the architecture and organization of a GPRS look like? Figure 3-7 elicits the questions remain for a high-performance reconfigurable computer architecture and organization. What implications each architectural point and organization elicits? And what tradeoffs and constraints that must come with them?

In this chapter, we propose a general-purpose reconfigurable processor architecture, which consists of a ISP core and a reconfigurable core, and necessary fixed-function hardware subsystems supporting reconfiguration and operations involving reconfigurable hardware. This architecture is based on our experience in building media processing hardware [58, 59] and conclusions from our discussion on past architectures in Appendix A.

In our architecture, we recognize the intrinsic differences in the achievable minimum clock periods between a fixed-logic custom designed microprocessor core and an uncommitted array of reconfigurable logic cells. We preserve the processor core's ability to optimize for higher clock speed by not forcing synchronicity with the reconfigurable logic core. This asynchronicity between the two cores has profound implication on the "achievable performance" by both cores.

We do not incorporate any special hardware in order to keep is generality. We define all necessary hardwired subsystems and their functionalities in order to support reconfigurable computing. These subsystems should be integrated with the processor and reconfigurable core.

Our architecture provides a fast control mechanism from microprocessor core to RLC. It can be used as a secondary one-way data transfer from microprocessor core to RLC, though it will tie up the microprocessor. This mechanism requires a simple addition to the microprocessor core's control register. Though we do not specifically require an addition to the microprocessor's instruction set to use this register, it is much more desirable to add such an instruction. This will allow compiler to automatically generate code using such an instruction instead of having a programmer to instantiate it.

Finally we compare other reconfigurable architectures and discuss pros and cons from the performance standpoint.

7.1 Technology Backdrop

As minimum feature size keeps shrinking, the same amount of logic occupies less and less area. The area vacated can be used to integrate more logic, embedded memory, or analog circuitry according specific applications. Transistor count on a microprocessor chip keeps increasing and soon it will, if not already, reach a point of diminishing return in performance if no change in microprocessor architecture or micro-architecture can be made. From a system point of view,

1. Integration of cores

2. Logic and memory will be fabricated on the same chip in the future. It will increase memory bandwidth and memory bottlenecks will be alleviated. More applications will change from memory bound to compute bound. Thus more applications will benefit from reconfigurable hardware.
3. It is likely, in the process of applying reconfigurable applications, that we discover that some processor real estate can be traded off for reconfigurable real estate. For example, a large cache takes up a significant amount of silicon area. However, its effect on performance may be marginal. Reconfigurable logic becomes one of many options for a "synthesizable" processor.
4. The cost of reconfigurable devices are a non-event since it can be amortized with increasing volumes. The cost gaps between reconfigurable devices and microprocessors are closing in fast due to the manufacturing prowess of dedicated semiconductor foundries.

7.2 System Architecture and Organization

In this section, we describe the architecture and organization of a general-purpose reconfigurable core. This core consists a microprocessor core, a reconfigurable core, and a fixed function subsystem to support reconfiguration and reconfigurable computing. We explain each architectural and organizational choice by examining past systems.

7.2.1 Core Pairs

As different applications require different computing power and have different cost considerations, future reconfigurable computing machines will probably encompass a variety of architectures. Different pairing of microprocessor cores and reconfigurable logic cores will likely coexist.

The one important system design and operational feature is to allow microprocessor and reconfigurable core be asynchronous. This is to allow microprocessor to run at full speed and to allow more speedup candidates for reconfigurable core.

Microprocessor Core

Whether a microprocessor or a reconfigurable core, an architecture favors some applications, whose program structures map well to its internal structures, and loses its advantages over some other applications. We do not know global application statistics. We cannot optimize architectures with maximum likelihood. Even if we can, there will always be market segments in need of some kind of specialization, in other words, architectures biased toward some structures Therefore, to specify a particular microprocessor core as the "best" choice is moot. Though our target application domain is media processing, we want our architecture to be general so that our performance evaluation method is equally applicable to all microprocessor architectures. Thus the microprocessor core in our architecture can be a superscalar microprocessor core, a VLIW core, or a single-issue microprocessor core. However, to make discuss easier, we will use a superscalar general-purpose core from here own. We should stress that this is not a requirement, but a convenience for discussion.

Reconfigurable Logic Core

The same argument goes to the reconfigurable logic core. We don't favor one architecture over another. Throughout this thesis, we deliberately leave the architecture of reconfigurable logic core unspecified. This non-commitment allows us to include a broad range of architectures.

To have a clearer picture of this core, we use commercial FPGAs as a visualization aid. Our core is an array of logic cells, such as the ones in Figure 5-1 or Figure 5-3. However, there are several points

- The core does not have embedded memory. Imagine the hierarchical array architecture in Figure 5-3 without the embedded memory.
- Though LUTs can be used as memory, we restrict our discussion on performance evaluation strictly on its logic capability.
- The core does not have the peripheral circuitry found in many commercial FPGAs so that area can be saved.
- There are many reconfigurable architectures with different "granularities". In our reconfigurable logic core model, this is also allowed. We understand granularities can affect performance for certain tasks. Our performance evaluation does not restrict it. More accurate estimation of the minimum clock period can be made if we nail down the granularity.

There are two "preferable" physical attributes on our reconfigurable core. One is that it has medium to large number of reconfigurable logic cells (> 1000). Many media processing functions are of medium to high complexity. There is little speed advantage to be gained when the task is very simple. This reflects the fact that ILP is well exploited by modern microprocessors.

The implication of this medium to high complexity to design effort, coupled with performance requirement, is that a simple straightforward compile will not generate satisfactory results. There is a trade-off between reconfigurable core size with potentially more speedup candidates and higher speedup. However, this is a decision for the system designers to make.

The second of the preferable physical attribute is it is organized as $(2W+c)$ x D cell array, where W is the memory word size, c is a constant a bit more than $2m$. These parameters are defined in Figure 6-2. The reconfigurable core is oriented $(2W+c)$ side connected to the FIFO Figure 7-3. The reason for this physical attribute is that it allows intermediate data sizes to grow. However, this is not a hard requirement for our performance estimation. It is a sensible design decision.

Organization

Where should reconfigurable hardware be placed in a general-purpose computing system? How does reconfigurable hardware access data in the host system? Ideally reconfigurable hardware should be put in a place where it has access to the maximum data bandwidth achievable by the system memory. Given that reconfigurable hardware

and a host CPU were implemented on separate silicon chips[23], there were two logical placements for reconfigurable hardware– local bus and I/O bus.

An I/O bus is designed for other subsystems to access system memory and other system I/O devices. It is easier to design a reconfigurable subsystem with an I/O interface to attach to an existing main system. It does not require re-design of the host general-purpose system.

The local bus of a particular CPU is typically proprietary and inaccessible (unless the chipset has built in supports) to other devices. It is also faster than its I/O bus. A local bus is short and is not open to support other peripheral devices, whereas an I/O bus is longer, thus slower, and is designed to let other standard devices to connect to the host system. Figure 7-1 shows an over simplified diagram of a reconfigurable subsystem attached to a general-purpose system's I/O bus.

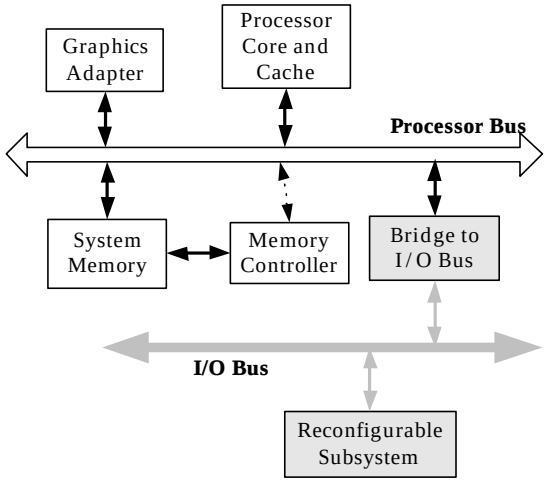


Figure 7-1: A reconfigurable subsystem attached to the I/O bus of a general-purpose system.

PAM, Splash, and other reconfigurable systems are attached processors. Therefore, their maximum achievable bandwidth to the main memory is limited by the I/O's capacity. This was the reason both PAM's and Splash's second version came out with hardware support to a faster I/O standard.

This organization let us leverage on existing general-purpose system infrastructure. No massive system re-design or re-writing of operating system software is required. One only needs to plug the reconfigurable subsystem to the system and use existing

The drawback of this organization is reconfigurable hardware has a slower bus to get data from or into system memory. The maximum sustainable data rate, or bandwidth,

[23] It is possible to integrate reconfigurable hardware with a core CPU and other peripheral circuitry, but the potential performance gain has not been verified to justify the concerns of economic and complexity.

for the I/O bus is no greater than that of the system bus. In addition, the overhead (in particular, latency) to send small amount of data through the I/O bus is high (compared to the local bus). Thus this organization favors transfers of large data sizes to the reconfigurable subsystem. PAM and Splash all went through revisions, which included faster I/O interfaces.

Figure 7-2 shows a different organization – putting reconfigurable hardware subsystem on the processor's local bus. Putting reconfigurable hardware on the CPU local bus would imply that the reconfigurable hardware subsystem is not simply "attached" to the host system through an open standard I/O interface. One must re-design the general-purpose host system to support the reconfigurable subsystem specifically.

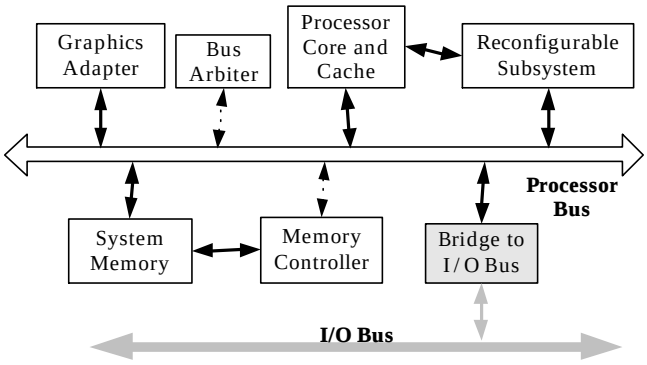


Figure 7-2: A reconfigurable subsystem directly sitting on the processor bus.

This organization, though giving the reconfigurable subsystem direct access to the system memory, requires much more hardware and software support. For example, the reconfigurable subsystem must contain host CPU interface, bus interface (processor specific) and memory interface hardware. Issues regarding how reconfigurable hardware subsystem interacts with CPU, how it accesses system memory, and other system resources must be resolved and built into hardware.

Besides the design and debug considerations of the system, one must devise a verification strategy for the reconfigurable subsystem since its behavior varies from one application to another. Each design must be verified in-system to guarantee its correct behavior. Each design could require different test strategies. In the author's opinion, this presents a major roadblock for general-purpose reconfigurable systems. A discussion on how to verify a hardware design on reconfigurable substrate is still an open research question (not much in the research community at this point) and is beyond the scope of this work.

In addition, to support the reconfigurable subsystem (initialization, reconfiguration, read/write, data transfer, etc.) in addition to all the devices found in a general-purpose computer, a full-fledged operating system must be re-written.

It requires significant time, (human as well as financial) resources, and expertise to

implement a GPRS with organization such as the one depicted in Figure 7-2. To demonstrate the benefits and viability of GPRS, one can remove the irrelevant, massive infrastructure hardware and software required to a full-fledged general-purpose system, and design a simpler proof-of-concept subsystem. It allows us to keep the complexity and development time to a manageable degree. Most importantly, it allows us to focus on the organization of the microprocessor-RP subsystem, the architectures of the microprocessor and RP, the applications, and development process. Furthermore, it closely resembles the kind of system structures and organizations of ERS, and recent core-based reconfigurable devices. CHIDI and Prism are two such examples [59, 60].

7.2.2 Reconfigurable Subsystem

Bus Interface

Reconfigurable hardware attached to a bus such as Figure 7-1 and Figure 7-2 requires a bus interface controller (BIC). This bus interface controller is an integral part of the reconfigurable subsystem. BIC can be implemented in a separate ASIC or FPGA for a particular bus. It should not be implemented with reconfigurable resources from the reconfigurable hardware.

Variable Clock Speed

Reconfigurable hardware functions should not run synchronously with the bus clock. However, BIC must run synchronously with the bus clock to capture the bus activity. Data from the bus must also be synchronized to the reconfigurable hardware functions. Therefore, a data-regulating buffer is also an integral part of a reconfigurable subsystem. We can model this by an imaginary clock boundary separating the reconfigurable subsystem into two logical subsystems as depicted in Figure 7-3.

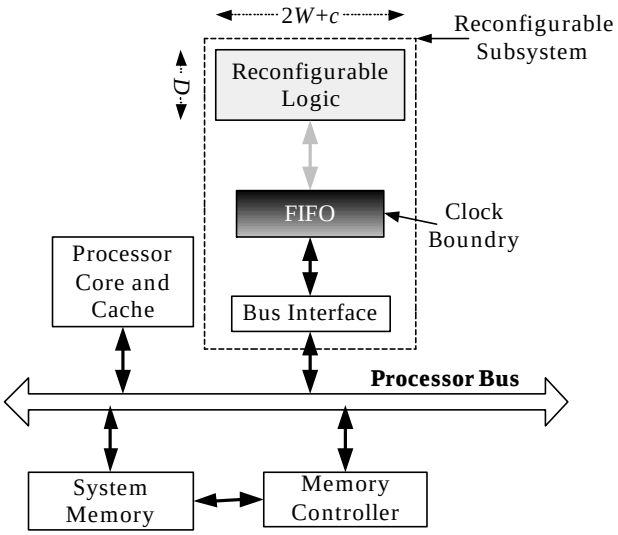


Figure 7-3: Bus interface and FIFO are an integral part of the reconfigurable subsystem, but

should be implemented with fixed logic (not reconfigurable).

In order to run as fast as the hardware functions can on reconfigurable logic array, the reconfigurable subsystem must contain a programmable clock generation circuit as part of the reconfigurable subsystem. Though this means the hardware functions will run asynchronously with the bus, it removes the constraints that the hardware functions must run only a few multiples (or quotients) of the bus speed.

The programmable clock generation circuit must provide a range of clock speeds with finer intervals according to the intrinsic properties of the reconfigurable hardware. The upper bound of the range should be determined by the minimum clock period estimate of a particular reconfigurable architecture. The lower bound should be determined by what is a reasonable speed estimate below which very little performance gain is possible.

The actual clock speed is the floor value of the interval within which a hardware function maximum speed falls. It is statically bound to a particular hardware function.

The programmable clock generation circuit must also have a processor interface. The CPU programs the clock as part of the (re-) configuration operation of the reconfigurable substrate. The processor interface within the programmable clock generation circuit must also be part of the reconfigurable subsystem.

Local Memory Subsystem

In the attached processor architectures as in Figure 7-1, access to memory through an I/O bus incurs a high latency. It renders addressing small data sets a very inefficient operation. Even for the local bus architectures, higher memory bandwidth (though localized within the reconfigurable subsystem) is possible with wider memory word width coupled with faster alternative memory technology. Therefore, PAM, Splash, PRISM, and CHIDI all include a local fast memory (SRAM). This memory can serve two purposes: as a fast memory page of the system memory, or as a lookup table for computation.

Though fast local memory reduces the latency and increase throughput for the data in the page, it incurs overhead in transferring data from the system memory. For applications, such as multimedia, whose data can be viewed as an infinite stream, data paging does not improve overall processing throughput. The ultimate performance is still limited by the bus's capability unless the whole program can run in the dedicated memory.

Furthermore, the slower memory and fixed data word width is part of a cost structure (e.g. DRAM is smaller, consume less power) that influenced the decision on the choice of the type of memory and the organization of a general-purpose computer. It is not an intrinsic property of a general-purpose system, nor has it to do with a general-purpose processor's datapath's performance. One can certainly design a general-purpose system based on SRAM to improve system performance.

The need for local memory in systems such as Figure 7-1 was an organizational shortfall, not one of the computing limitations of microprocessors. For systems organized as Figure 7-2, the need is moot. However, that fast local memory contribution often confused people and was often compounded into many reconfigurable subsystem's performance benchmarks boasting speedup numbers, in particular, on memory-bound problems. This effect should not be attributed to reconfigurable logic's intrinsic performance advantage over microprocessors.

Our reconfigurable subsystem architecture does not use dedicated memory. Data come from the main memory. This makes the microprocessor core and RP core face the same memory performance characteristics. It is left to system designers to decide what memory performance they need for their applications.

FIFO

Since reconfigurable substrate runs asynchronously with the bus, a FIFO is necessary to regulate the speed difference between the bus and the reconfigurable subsystem. If the reconfigurable subsystem has external I/O, additional FIFO between the hardware functions and I/O is necessary to regulate the data rate difference there. Thus, FIFO is also an essential part of the reconfigurable subsystem. It also undermines the need for a local memory, which can be used as a buffer. Figure 7-4 shows a FIFO subsystem for regulating data between the system bus the reconfigurable substrate. Signals on the left hand side are synchronous to the bus interface. Signals on the right hand side are synchronous to the reconfigurable hardware function's operating clock. The graded shade indicates a clock boundary.

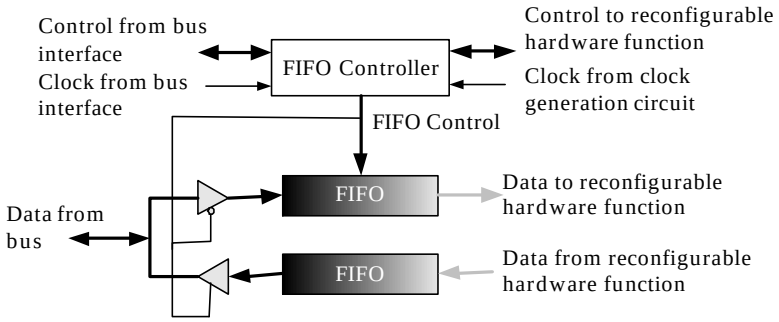


Figure 7-4: A FIFO subsystem buffering data from system to reconfigurable substrate and vice versa.

Configuration Control Subsystem

In order for the CPU to program the reconfigurable logic, a configuration control subsystem (CCS) is necessary as part of the reconfigurable subsystem. Typically,

configuration is a slow process compared to CPU's clock speed. It is often done serially[24]. The configuration control subsystem mediates between CPU writes of the configuration bitstream and the actual reconfiguration process.

There are many ways to implement CCS with different cost and efficiency considerations. For example, CPU can write a configuration in 32-bit words (in a 32-bit architecture) into the shift register of the CCS and signal to CCS a word is ready. CCS shifts the word bit by bit to the configuration memory of the reconfigurable logic. Then it signals to the CPU of the completion of a word through interrupt or raise a status flag in CPU's register space. The former incurs a context switching cost, and the later ties up CPU in a polling loop.

Another way to implement CCS is to use a FIFO to hold the entire configuration bitstream. CPU writes the FIFO, in one service request, the complete configuration. CCS then configures the reconfigurable logic and signals its completion through interrupt. This implementation will reduce the overhead on CPU's context switching or polling, and reduce the configuration time by making configuration continuous from the beginning to the end.

In our model, the CCS is not an essential part in the discussion of performance evaluation. For a hardware function operating on a large data set, the overhead can be amortized to become a non-factor. For multimedia applications, this is often the case. We will use the second implementation for our model, since it reduces synchronization cost (thus performance) at the expense of the cost of a small amount of memory.

Processor Register Interface

It is often necessary for the host CPU to write initialization, control, or formal parameters to a hardware function implemented on the reconfigurable logic. For example, the coefficients (formal parameters) of the color space transformation can be implemented as a hardwired or "programmable" function (Figure 4-4). In the programmable form, the coefficients can be written from the microprocessor while the function does not need to be reconfigured for different sets of coefficients. The programmable 3x3 matrix multiply can do any 3x3 multiplication with the correct input data types. However, it loses some performance versus a hardwired version since more optimization is possible with known coefficients.

Since reconfigurable logic's clock is asynchronous to the CPU clock, this register interface also creates a clock boundary within the reconfigurable logic. The synchronization of CPU writes is done through a set of control registers.

The control register consists of common and application-specific control bits. The common control bits include reset and start. Reset is to make sure that a hardware

[24] This depends on how one implement the configuration memory. There is a trade-off between parallel configuration and cost. Parallel configuration requires more configuration circuitry; that is, more space.

function starts from a known state, or to abort current operation and start all over. Start tells a hardware function that CPU has set up all information it would need to run correctly and the controller of the hardware function can start its operation.

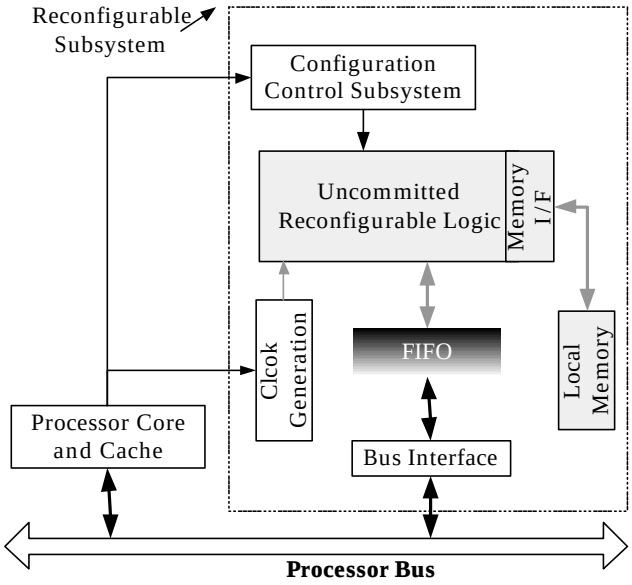


Figure 7-5: Reconfigurable subsystem showing constituent fixed functions and a reconfigurable logic core

The parameters are written through a set of application registers. For example, if we implement the color space transformation with a programmable 3x3 matrix multiply hardware function, the CPU must write nine coefficients (8-bit each in this case) to the hardware function at the beginning of its operation. The nine coefficients require at least three CPU write cycles (assuming we pack the coefficient into three 32-bit words). Five memory addresses must be allocated for these registers. Figure 7-6 shows the fixed and application specific register space.

There is a strong coupling between the memory mappings of the control, initialization and the formal parameters registers and software drivers. The processor register interface hardware and the corresponding pieces of software that initialize, control, and configure the parameters are considered in unity to guarantee correct operations.

Our new architecture puts the register interface right inside the register file in an microprocessor. A dedicated special register for reconfigurable logic serves this purpose. This would require no addition or change to the instruction set (if we ask the compiler not to allocate this register for any other general purpose uses). We can simply add one more register to the register file, if there are still register addresses not mapped to any registers. Or we can remove one general-purpose register from the register address space and replace a special register for reconfigurable logic (SRRL).

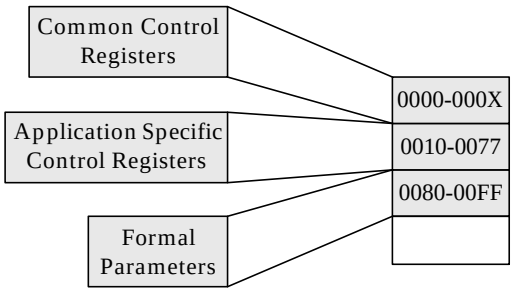


Figure 7-6: Memory address for common registers is fixed, while application specific registers addresses can change with applications[25].

Figure 7-7 shows a 32-bit SRRL for a 32-bit processor (or a processor core). The sync bit is located at the MSB position so that it can be predicated as a sign bit. The 7-bit address field is enough to provide 128 distinct addresses. Along with the 24-bit data field, this format can write 2G register bits, which is orders of magnitude more than the large FPGA can offer today.

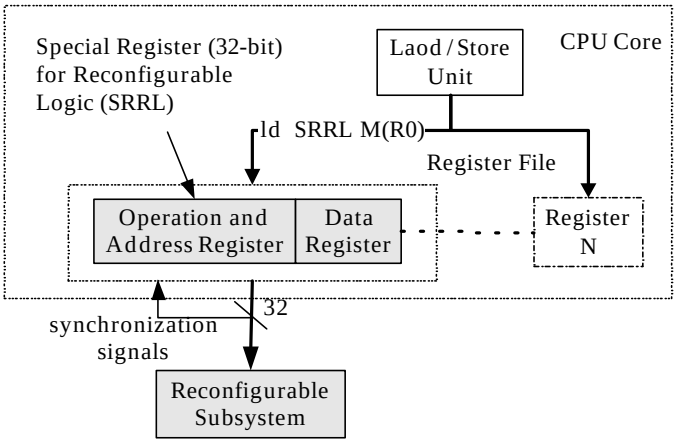


Figure 7-7: A special register for reconfigurable logic (SRRL) in an microprocessor's register file.

Since writing by the CPU to the register and reading from register by the hardware function is asynchronous, we need to set up a mechanism for synchronization. This mechanism is through a semaphore bit, or sync bit, in the special register (see Figure 7-7). When CPU writes to SRRL, it sets the sync bit to one indicating a control, initialization, or parameter word is available in the data field. The corresponding address is in the address field. The synchronization state machine (SSM) will latch on the data and acknowledge this write by resetting the synchronization bit to 0.

---

[25] The address map is randomly chosen. For practical consideration, one should choose a mapping that minimizes required decoding logic, for example, one-hot mapping when less than 7 addresses are required for an application.

SSM is running with the same clock with the rest of the hardware function. It is application dependent. However, SSM is a soft firmware[26], just like the application independent register interface, that it must be included in every reconfigurable hardware function.

CPU can only write to SRRL when the sync bit is 0. Thus, in the register interface driver software, SRRL should be always read first by the CPU to check the sync bit before it attempts to write.

Memory Addressing Unit

Different microprocessor architectures do have different addressing capabilities that will affect their performances. Custom addressing capabilities can be built in the reconfigurable core to compliment the microprocessor core's addressing capability. However, since we allow reconfigurable core to run asynchronously with the processor core and the its clock speed is programmable, it would be very inefficient to synchronize on every memory access. This will probably offset any advantages.

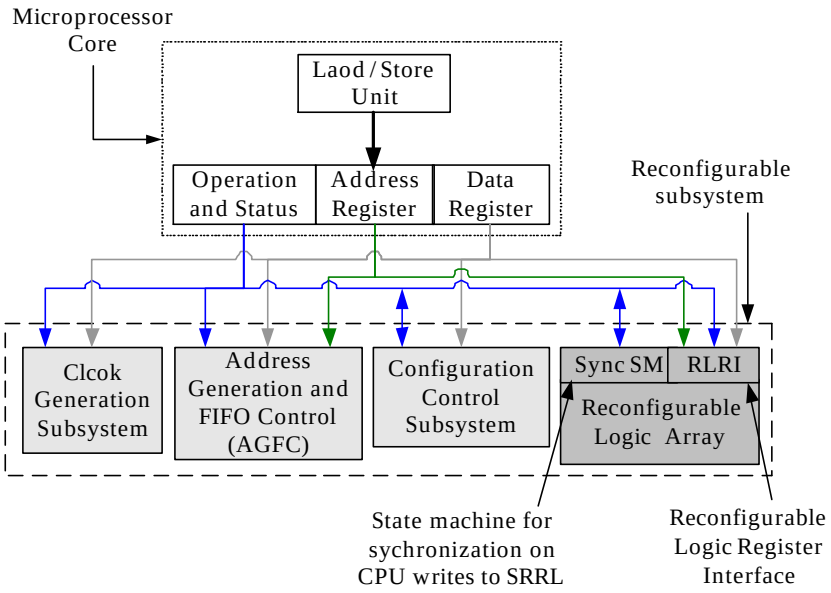


Figure 7-8: Reconfigurable system model showing the special register interface.

Therefore, it is best to implement it outside of reconfigurable core as a fixed function unit. If we implement it as a fixed function unit, it won't be able to reconfigurable for the target applications. Hence the performance benefits (if any) from a hardwired

[26] It sounds like an oxymoron. What it means a hardware functionality that must be included in every configuration, yet its placement in the reconfigurable logic need not be fixed.

memory address generator should not be attributed to the advantages of a reconfigurable core.

General-Purpose Reconfigurable Core

With the cores and fixed-function subsystems supporting reconfigurable computing, we now have a general-purpose reconfigurable core as shown in Figure 7-9. The microprocessor core can be taken from an existing superscalar, VLIW, or any microprocessor core. We add one special register (SRRL) and one special instruction to the instruction set.

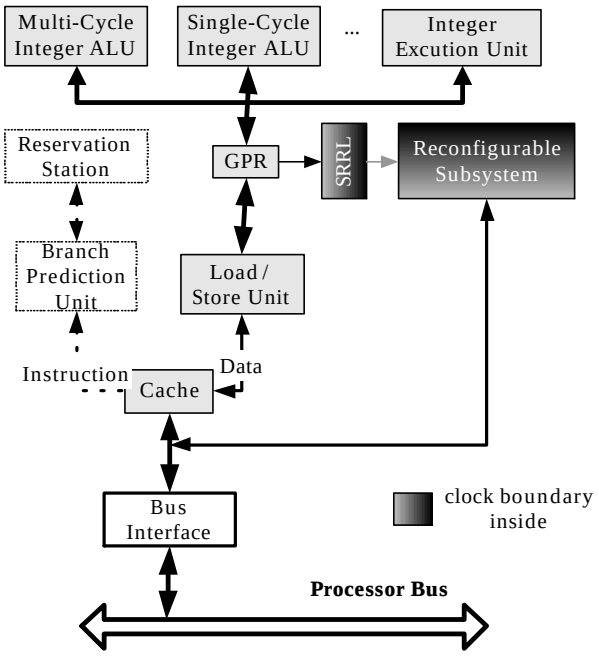


Figure 7-9: An integrate general-purpose reconfigurable architecture. The microprocessor core and reconfigurable logic core run asynchronously.

This general-purpose reconfigurable architecture can be integrated on one chip so that the microprocessor core and RLC are based on the same semiconductor technology (Same minimum feature size, same core operating voltage, same metal processes).

Finally, we can visualize a datapath model for the integrated general-purpose reconfigurable architecture. Figure 7-10 shows the microprocessor datapath model in Figure 6-4 with the reconfigurable logic core. The fixed function logic for the rest of reconfigurable subsystem is omitted for clarity. This model suggests both the microprocessor core and reconfigurable logic core are subject to the same memory system constraints.

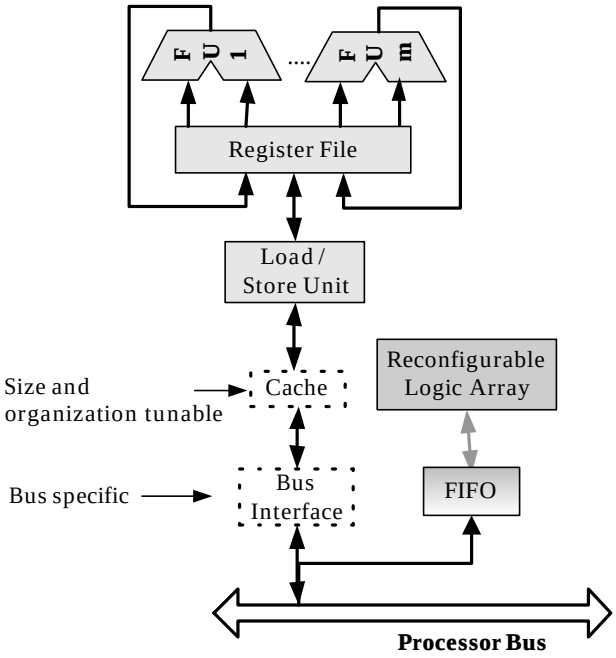


Figure 7-10: A datapath model of a general-purpose reconfigurable core.

Note that so far, we have not target our microprocessor cores to any specific architecture, nor have we targeted the reconfigurable logic core to a particular architecture. This "lazy binding" makes our approach to performance evaluation applicable to a wide range of architectures.

7.3 Other Architectures

In this section, we explain why we made the decision to let the microprocessor core and reconfigurable logic core run asynchronously. We also examine the performance implication if we force the microprocessor core and reconfigurable logic core run synchronously. We use some systems as examples and look into their limitations as a result of forcing synchronicity.

Some architectures put reconfigurable logic directly on the datapath such as RAW [9, 10], or as an alternative datapath for the microprocessor datapath such as PRISC and GARP [10, 61, 62]. In the former case, every instruction execution must go through the reconfigurable logic. In the later case, reconfigurable logic serves as alternative execution unit.

RAW is a drastically different architecture, which consists of arrays of small datapath and local instruction and data memory. The reconfigurable logic sits on the register-ALU-reconfigurable path. Because of this organization, every instruction execution requires a configuration. This also means that the clock rate must tolerate the delay through the reconfigurable logic on top of the ALU delay, which can adversely affect the

achievable clock rate with fixed-logic ALU.

PRISC puts reconfigurable logic in the register file datapath, make it an alternative functional unit Figure 7-11. This implies the PFU must have a fixed latency the rest of the functional units would not know when its computation is done. It also means it must be synchronous to the rest of the datapath because a register file cannot be efficient if it needs to synchronize on every data. In addition, it is not clear how it is integrated with register file without penalizing it for non-deterministic loading characteristics. Finally, the rest of the functional units are fixed-logic execution units. From our conclusion in chapter 8, PFU must implement medium to complex task, or bit-wise logic to have any speedup advantages. In that case, PFU must be relatively large and it must implement more than a few instructions. That makes synchronization problem worse since making a few complex reconfigurable functions with a fixed latency is hard to achieve.

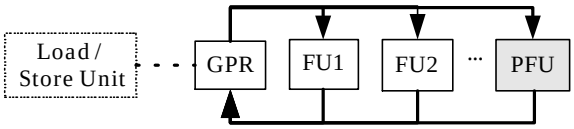


Figure 7-11: The PRISC architecture. A programmable functional units is sitting on the register bus.

Indeed, PRISC reported synchronization complexity and had to restrict PFU's use for two inputs and one output combinational functions. That is equivalent to logic emulation. This is not surprising from our conclusion in chapter 8. Therefore, PRISC does not offer performance benefits to any medium-size data types or simple arithmetic operations.

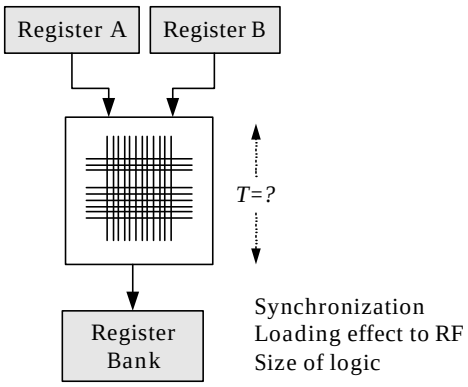


Figure 7-12: Problems with PFU as a functional unit attached to a register file

Figure 7-12 help us the understand the problems if a piece of reconfigurable logic sits between register ports.

Garp's architecture is very similar to ours except that GARP is connected to processor cache (Figure 7-13). As a result it must run synchronously with the processor. That means its clock speed cannot be change with the application. It also must run a few times slower than the processor core.

Fixing the clock speed adds one more constraint to potential speedup candidate on top of the performance gain requirement and area constraints. The clock rate must be fixed at the system design phase. If it is fixed too high, it can exclude many potential speedup candidates leaving it virtually useless. If this is fixed too low, some performance gains cannot be realized.

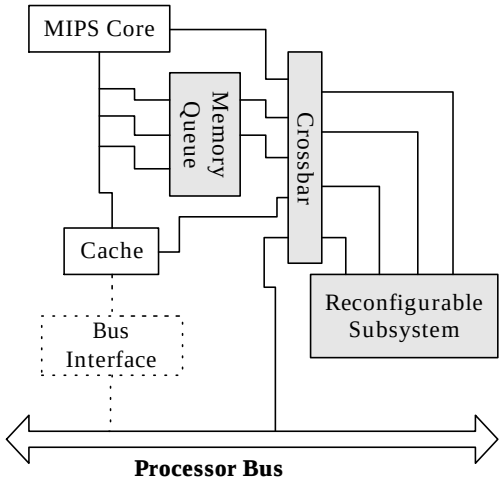


Figure 7-13: Garp architecture [11].

Since GARP, does not have explicit synchronization mechanism, its latency must be fixed at the design time. Fixed latency further reduces the number of potential speedup candidates if they can only run with a larger latency at that clock rate. For other candidate functions which can be run with a smaller latency, extra sets of registers are necessary to add delay cycles to meet the latency constraint. These extraneous registers can make the original function too big for the reconfigurable logic core.

Our objective on reconfigurable architectures is to avoid tying too much specificity at the system design time instead of the application development time, which is what defines reconfigurable computing.

In the next chapter, we develop analytical performance estimates using our architecture as a performance evaluation platform.

8 Performance Analysis

Reconfigurable computing suffers from limited applications due to long application development cycle and hard-to-predict performance benefits. To over come these problems, we must create more applications with less amount of time.

In this chapter, we use the datapath performance model derived in 6.2 for the microprocessor core, and the general-purpose reconfigurable architecture in Figure 7-9 as future general-purpose reconfigurable processor. Given minimum knowledge about the system and a task, we would like to know quickly if it is a speedup candidate. This minimum set of properties includes the number of operations (defined as operations directly supported by the microprocessor instruction set), the types of operations, the operand types, and the number of input and output variables for memory bandwidth limit.

We then establish an optimistic upper bound by assuming all *N* operations are parallelizable, that is, there is no data dependency among them. We then establish a lower bound by assuming that all *N* operations are serial. That is, all operations depend on their immediate predecessors.

Using these two bounds and information on the number of memory accesses to support those *N* instructions, we can determine if the task is bound by memory. The speedup potential is confined in a region where memory bandwidth is higher than the upper bound memory bandwidth.

With these bounds and memory bandwidth not limiting, we can use our minimum clock period estimate to further rule out speedup candidates. This process will eliminate wrong candidates very quickly.

These estimates are formally formulated. They show how various factors affect processor datapath performance. From these formulas, we know what task properties and microprocessor properties to look for speedup candidates. These formulas also provide a mathematical base for some long observed rule of thumbs developed from experiments.

Finally, the whole process can be integrated as a front end tool for reconfigurable application space exploration.

8.1 Performance Estimates

Given a speedup candidate and a microprocessor datapath performance properties (modeled by Figure 6-4), can we quickly get some performance number without actually writing the code and measure the performance?

We assume we know the following information from our microprocessor properties and task properties.

- Multiple functional units, *M*
- Some operations can execute on different FU at the same or different costs
- Minimum information on task: the number of operations *N*, the type of each operation, and its operand size. These *N* operations must be supported by the microprocessor's instruction set natively.
- Number of inputs, n_i, number of output, n_o. Here we refer inputs and outputs as variables obtained from memory. This information is not need for performance bound estimate, but they will be used for compute-memory bound determination.

The task can be represented as a DAG, a parsed forest, or taken from a block of three-address instructions as discussed in chapter 6. However, we don't need the dependency information for our estimates.

Upper Bounds

The best possible performance (the minimum number of execution cycles required) is when there is no dependency among variables. That is all operations can execute at the same time. No dependency also means the data can be pipelined to a functional unit's maximum pipelining capability. In other words, the task is one such that

- Disregard dependency
- All parallel operations
- Maximum pipelining if available

The following equations formularize this problem.

EQ 8-1 $$N = \sum_{i=1}^{M} n_i$$

EQ 8-2 $$n_i = \sum_{l=1}^{L_i} n_{i,l}$$

where *N* is the total number of operations in a DAG, *M* is the number of functional units connected to a register file, n_i is the number of operations assigned to the *ith* FUDP, and $n_{i,l}$ is the number of operations assigned to its *lth* latency group, and L_i is the number of latency groups in the *ith* FUDP.

EQ 8-3 $$l_i = \sum_{l=1}^{L_i} g_{i,l}$$

EQ 8-4 $$L \neq \sum_{i=1}^{M} L_i$$

where l_i is the total latency of all operations assigned to the *ith* FUDP, $g_{i,l}$ is the total latency from operations assigned to *ith* FUDP's *lth* latency group, and L is the total number of latency groups in the whole RF-FU datapath model. Note that L is not the sum of L_i since different latency groups belonging to different FUDP can have the same T and I.

EQ 8-5 $$g_{i.l} = I_{i,l} * n_{i,l}$$

where $I_{i,l}$ is the initiation interval of *ith* FUDP and *lth* latency group. The question now becomes finding a set of $n_{i,l}$ such that minimizes the longest l_i over M. That is,

EQ 8-6 $$l^u = \min_{i,l}(\underset{i=1}{\overset{M}{MAX}}(l_i)) = \min_{i,l}(\underset{i=1}{\overset{M}{MAX}}(\sum_{l=1}^{L_i} I_{i,l} * n_{i,l}))$$

where *min* is a scheduling procedure. It finds the shortest "parallel" schedule by searching over all functional units and their latency groups (Figure 6-7). *min* can schedule multiple instructions for concurrent execution. For instructions which can execute on multiple functional units, *min* schedules the one with the smallest $I_{i,j}$. *MAX* is a function, which returns the maximum value in a set of M items.

We call this problem "finding the shortest distributed schedule" problem. We can think about this problem as trying to compact the depth of the temporal pipeline shown in Figure 6-7.

Lower Bound

Given a set of N operations and their operand types, and a microprocessor core, whose datapath is modeled by Figure 6-4, the longest execution happens when all operations must wait for previous operation to finish. That is, the lower bound is obtained by assuming "maximum dependency". The lower bound (on throughput) can be calculated by chaining all operations together one after another.

- Multiple functional units
- Same operations can be performed by multiple FU at possibly different costs

We can formulate it as with the following equations:

EQ 8-7 $$g_{i,l} = T_{i,l} * n_{i,l}$$

EQ 8-8 $$L_i = \sum_{l=1}^{L_i} g_{i,l}$$

EQ 8-9 $$l_l = \sum_{i=1}^{M} l_i$$

EQ 8-10 $$l^L = \underset{i,l}{Min}(l_l) = \underset{i,l}{Min}(\sum_{i=1}^{M} \sum_{l=1}^{L_i} T_{i,l} * n_{i,l})$$

where *Min* is a scheduling procedure. It searches over all functional units and latency groups for the shortest "serial" schedule. For each instruction, *Min* finds the smallest $T_{i,l}$, whose functional unit can execute the instruction. However, it cannot schedule another instruction until the previous one finishes execution.

The ratio of upper and lower bounds, R_B, tells us how far apart are these bounds, or the "spread". It gives an indication of how good are these estimates.

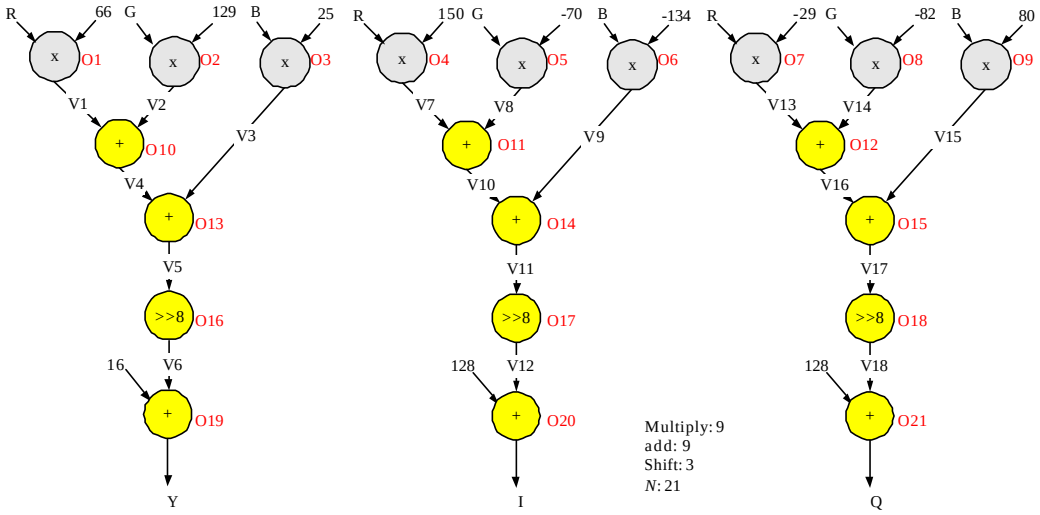
EQ 8-11 $$\text{Ratio of Bound (ROB) } R_B = \frac{l^U}{l^L} = \frac{\underset{i,l}{min}(\underset{i=1}{\overset{M}{MAX}}(\sum_{l=1}^{L_i} I_{i,l} * n_{i,l}))}{\underset{i,l}{Min}(\sum_{i=1}^{M} \sum_{l=1}^{L_i} T_{i,l} * n_{i,l})}$$

Equation 8-6 suggest the upper bound estimate is a function of M, L_i, $I_{i,l}$ and N. $I_{i,l}$ are constants for any (operator, operand size) pair. If M, L_i, and N are all unbound integers, the search for l^U will be $O((M-1)^{N+1})$. However, limits of physics dictate that M and L_i are small integers (6.2). In addition, in many cases, there is only one choice for expensive operators such as multiplication and/or division. Single resource makes determining l^U much simpler than EQ 8-6 suggests. Some very intuitive heuristics suffices. This is best explained by some examples.

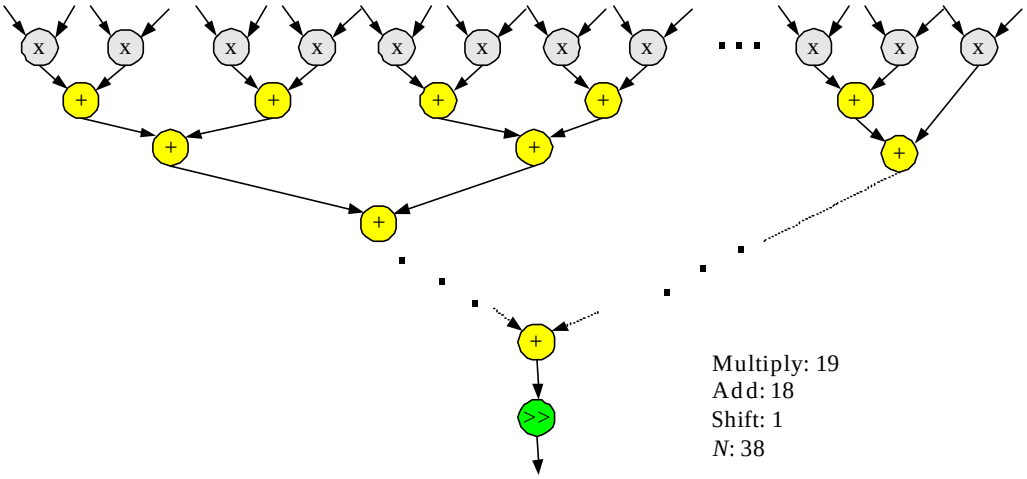
Examples

Figure 8-1 shows two typical media processing tasks. The rgb2yiq transform requires a total of 21 operations, nine multiplies (O1-O9) and twelve adds (O10-O21), while the 19-tap filter needs nineteen multiplies and nineteen adds. Note that we treat a shift as an add because the shift and add are implemented in the same integer unit and has the same performance properties.

Finding upper bound can be visualized as filling M buckets with N bricks (Figure 8-2). M buckets are M functional units. Instructions are bricks with certain heights. Some bricks' heights change with the buckets. Some bricks can only go to certain buckets. The objective is to find a filling that produces the shortest highest height.



(a)



(b)

Figure 8-1: Examples for bound estimates (a) color space transform (b) a 19-tap filter.

The filling strategy starts from the most restricted bricks that can go to only one bucket. If all such bricks are exhausted, then we take the bricks that have the most heights and fill them first. Because there are multiple choice of buckets, we choose the one the brick has the least height. This process repeat until all bricks are exhausted.

In our examples, we recognize that there is only one multiplier in each of these targets. The multiplication instructions can only go to one functional unit (the multiplier). Since nine multiplications alone take more time than the twelve adds, the remaining twelve

adds can go to other functional units.

Finding lower bound is equivalent to filling the buckets, where the heights of all buckets are increased by the same amount no matter which bucket actually gets the brick (Figure 8-3). For every brick which can fill multiple buckets, the one, which makes the brick shorter, gets it.

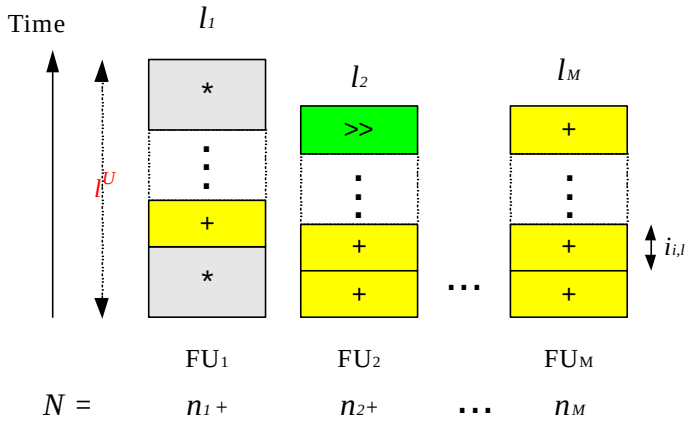


Figure 8-2: Finding upper bound visualized as minimizing the highest bucket.

Figure 8-2 shows that the upper bound can be viewed as a measure of *M*. Though some instructions can only execute on one functional unit, real applications have instruction mix that execute on all functional units.

Indeed, we can use a trivial instance to see how *M* is reflected in the upper bound estimate. Let's assume every functional unit have the same latency of *T* cycles and initiation interval of *I*. We can sum up the numbers of operations according to their types and divide them by the number of functional units for each type to obtain scheduling length n_i. We then multiply *I* and n_i for each functional unit to obtain the total number of cycles, l_i. The largest l_i of all functional units is the trivial upper bound, l^U. If all functional units are identical, then the instructions are evenly distributed among *M* functional units, ($n_i = N/M$).

Finding lower bound is much easier since we assume 100% dependency. This assumption becomes a reality when,

1. There is only on functional units that execute all instructions (e.g. PowerPC 603).
2. One functional unit execute all instructions faster than the rest (pedagogical).

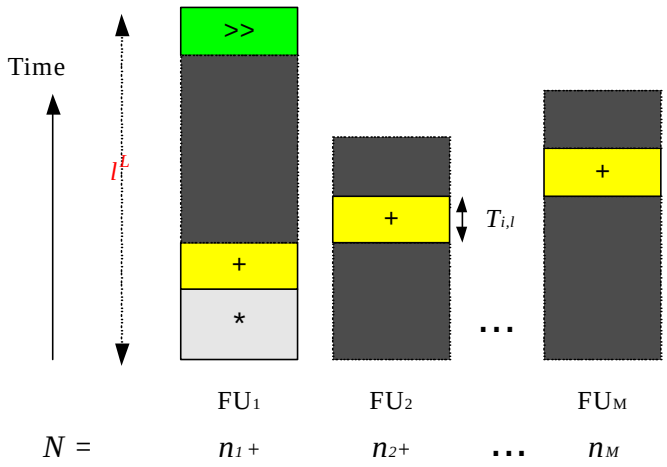


Figure 8-3: Finding lower bound visualized bucket filling. The heights of the buckets all increase by the same amount.

Table 8-1 shows these bounds and the range of bounds using PowerPC 603, PowerPC 604 and Alpha 21164 as example [51, 55, 57]. We show how instructions are distributed among mulit-FU processors for the upper bound estimate. The lower bound is not shown because it is much easier to picture it with Figure 8-3. From Table 8-1, we can make several observations:

1. The ROB is relatively small, and are very close to the number of FU.
2. UB, LB increase when increasing
   - N increases
   - Many expensive instructions such as multiplication.
   - There is only one functional units.

In other words, the following are true.

EQ 8-12 $$l^L \sim N$$

EQ 8-13 $$l^u \sim N / M$$

EQ 8-14 $$R_B = l^{L/} l^U \sim N /(N /M) \sim M$$

This is not surprising if we think about the bounds from some very intuitive point of views.

We obtained the lower bounds by assuming complete dependency. So all N instructions execute serially. Though instruction costs vary, the number of instructions still factor into the lower bound. If we use a pedagogical example – all instructions have the same cost, then EQ 8-11 produces NT. Even with different cost, we can still see how N factors into the equation by letting instruction costs represented by $(c+x)$, where c is a constant

representing the least expensive instructions and x depends on the instructions. Then EQ 8-11 will become cN plus a summation term.

Therefore we can view this simple lower bound as a measure of instruction length, N and T. So the lower bound exposes the effect of N and T, the latency of each operation, as no pipelining is possible if all data are serially dependent. Results shown in Table 8-1 confirms this observation. We call this the task load test.

Table 8-1: Performance upper bound and lower bounds on color transform example using three representative microprocessors (32-bit datapath).

Processor	PowerPC 603		PowerPC 604			Alpha 21164		
Functional Unit	IU1		IU1	IU2	IU3	IU1	IU2	
Operator	+/-	*	+/-	+/-	*	+ / -	+/-	*
$T_{i,l}/I_{i,l}$	1/1	4/2	1/1	1/1	4/2	1 / 1	1/1	8 / 4
Scheduling Length ($n_{i,l}$)	12	9	6	6	9	12	9	
Total Latency (l_i)	30		6	6	18	12	36	
Upper Bound (l^U)	30		18			36		
Lower Bound (l^L)	48		48			84		
ROB (R_B)	1.6		2.67			2.33		

(a)

Scheduling Length ($n_{i,l}$)	19	19	10	9	19	19	19	
Total Latency (l_i)	57		10	9	38	19	76	
Upper Bound (l^U)	57		38			76		
Lower Bound (l^L)	95		95			171		
ROB (R_B)	1.67		2.5			2.25		

(b)

For the upper bound, if we assume all functional units are identical, then from EQ 8-6 the upper bound is simply NI/M. In reality, functional units are not identical because they some operations takes more time than the other, and it is best to separate them. Thought is some cases, designers chose to share the circuit with simple instructions execute faster than more complex ones.

Therefore, the upper bound also exposes whether the instruction distribution from the cost point of view matches what is available. For example, since multiplier is scarce, the more multiplications in the instructions will make the cause the upper bound dominated by the availability of multiplier, thus *I* will be reflected in the upper bound. On the other hand, if the instructions are roughly distributed according to "distribution" of the functional units, the effect of *H/M* will dominate.

So it is fair to say the upper bound estimate is a test to the "fitness" of the number and the "distribution" of functional unit to the task's. We call this task characteristic test.

Finding Memory Bottleneck

So far we have ignored potential memory bandwidth problem. We have not required prior knowledge about the input and outputs of the task. That is, whether the inputs variables are memory reads and output variable are memory writes. If we know which input variables are coming from memory and which outputs are going the memory, we can use our bounds to check potential memory bound problems. Memory system performance parameters are either fixed as in an existing system, or as variables as design parameters for a new system.

Our upper bound estimate can be calculated to obtain upper bound memory bandwidth requirement, m^U. m^U is what is required to sustain upper bound data processing rate. Similarly, lower bound estimate translates into lower bound on memory bandwidth requirement m^L. We obtain m^U and m^L from l^U, and l^L.

$$P_{ic}^U = l^U \times C_p = C_P \times \min_{i,l}(MAX_{i=1}^{M}(\sum_{l=1}^{L_i} I_{i,l} * n_{i,l}))$$

EQ 8-15

$$P_{ic}^L = l^L \times C_p = C_{P\times}(\min_{i.l}(\sum_{i=1}^{M}\sum_{l=1}^{L_i} T_{i,l} * n_{i,l}))$$

EQ 8-16

where P_{ic}^U and P_{ic}^L are the total execution times for our task, and $P_{ic}^U < P_{ic}^L$

$$m^U = n_M / P_{ic}^U = n_M / [C_P \times \min_{i,l}(MAX_{i=1}^{M}(\sum_{l=1}^{L_i} I_{i,l} * n_{i,l}))]$$

EQ 8-17

$$m^L = n_M / P_{ic}^L = n_M / [C_{P\times}(\min_{i.l}(\sum_{i=1}^{M}\sum_{l=1}^{L_i} T_{i,l} * n_{i,l}))]$$

EQ 8-18

where, n_M is the number of memory words read from and written back to memory. Given memory performance information, we know the memory bandwidth, M_B. There are three cases with respect to M_B relative to m^U and m^L, as shown in

EQ 8-19 $$m^L > M_B \Rightarrow n_M / [C_{P\times} (\underset{i.l}{Min}(\sum_{i=1}^{M} \sum_{l=1}^{L_i} T_{i,l} * n_{i,l}))] > M_B$$

EQ 8-20

$$m^L < M_B < m^U \Rightarrow n_M / [C_{P\times} (\underset{i.l}{Min}(\sum_{i=1}^{M} \sum_{l=1}^{L_i} T_{i,l} * n_{i,l}))] < M_B < n_M / [C_P \times \underset{i,l}{min}(\underset{i=1}{\overset{M}{MAX}}(\sum_{l=1}^{L_i} I_{i,l} * n_{i,l}))]$$

EQ 8-21 $$m^U < M_B \Rightarrow n_M / [C_P \times \underset{i,l}{min}(\underset{i=1}{\overset{M}{MAX}}(\sum_{l=1}^{L_i} I_{i,l} * n_{i,l}))] < M_B$$

From these equations, we can derive what causes a task compute bound or memory bound.

1. M_B is smaller than both $m^U$and m^L . EQ 8-19 suggests either n_M (a) is large, C_p (b), $T_{i,l}$(c) and N (d) are small, or simply M_B is small.
    a. Too many memory accesses in the task
    b. High clock frequency, result of advanced semiconductor processes
    c. Since $T_{i,l}$ corresponds to [(operator group, operand size), latency], $T_{i,l}$ small means instructions in the task are inexpensive simple instructions.
    d. Too few instructions means the task is too simple. Consider the extreme case where there is no instructions (N->0), only memory accesses.
    e. Simply slow memory, not matched to the microprocessor, bad system design.

    This task is memory bound, we should look for other opportunities.

2. M_B is smaller than both m^U, but greater than m^L. Memory bandwidth very close to processing bandwidth. There may be some room for speedup, but speedup is going to be relatively small if any. The ROB, or the spread, should give an indication of how much room there is to maneuver.

3. M_B is greater than m^U. EQ 8-21 suggests that either it is n_M (a) being small, or C_p (b), $I_{i,l}$ (c), N/M (d) (from EQ 8-2), or M_B being large (e). This means:
    a. Few many access in task.
    b. Slow microprocessor, result of fabrication processes.
    c. Long initiation intervals, this more reflects that the instructions are expensive instructions.
    d. Either the task is complex or there is too few functional units.
    e. Memory bandwidth is too high? This is more pedagogical than real. Together with b, it means somebody use very old microprocessors with the fastest memory.

This is the best speedup candidate. However, it depends on how far apart between M_B and m^U as well as the ROB for the magnitude of speedup. The greater the distance between M_B and m^U, the speedup will be greater.

To determine whether a task is memory bound from EQ 8-17 and EQ 8-18, we can detect memory bound tasks and zone in on potential speedup area, as shown in Figure 8-4. Figure 8-4 (c) shows we can only gain speedups up to when it hits memory bandwidth limits.

In addition, we conclude that the properties that make a task compute-bound are high instruction counts, expensive instructions, and few memory accesses. The opposite is true for memory-bound task.

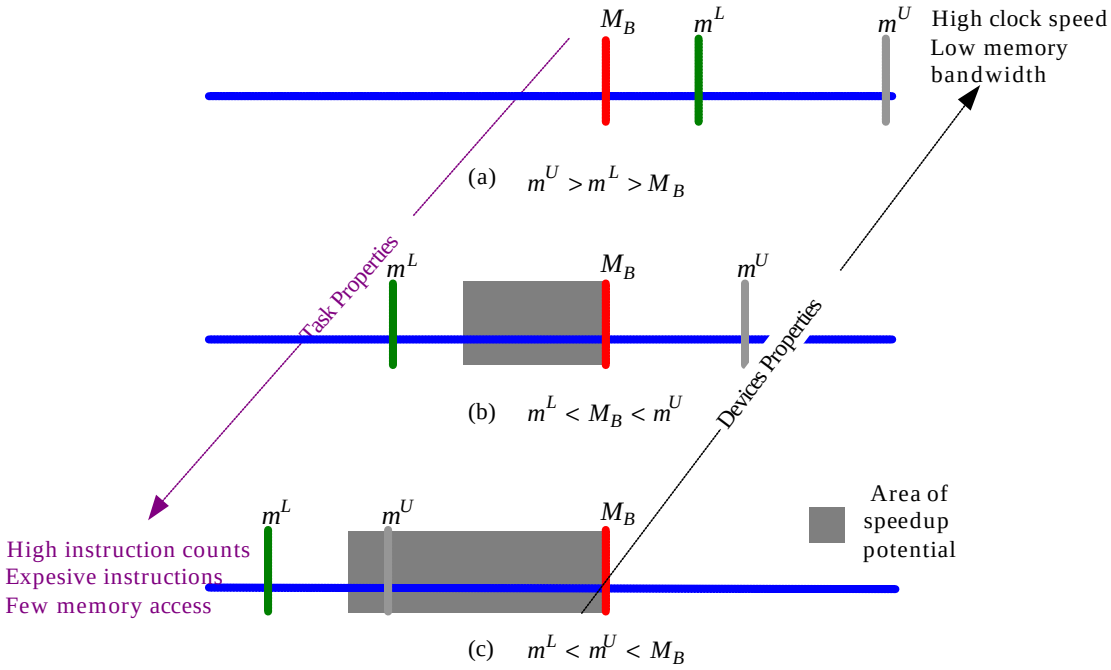


Figure 8-4: Speedup candidate decision diagram and properties that move the dividends.

This is why encryption, Boolean satisfiability can have thousand fold speedups [63]. In addition, what makes N increase without increasing memory access are those tasks with small sized variables, e.g. bits. Imagine all the variables in Figure 8-1 are bits. The number of operations per variable is not changed, but the number of variables we can packed into a word increases dramatically. As a result the total number of instructions increases. This is why logic simulation is really slow on microprocessor-based systems. Media processing sits in the middle of those cases. On one hand, the variables are smaller- to medium-sized. "Variable-packing" is modest. MMX cannot fully take advantage of this data packing if the precision of intermediate results grow. In the worst case, no packing is possible. That puts the potential speedup from reconfigurable

logic to *m*, plus the effect of *N*. Therefore, the number of operations per variable can determine the speedup magnitude.

In addition, we show that processor clock speed that factor into performance equations, though it often has to trade off with *M*, *T*, and *I* by the laws of physics.

Our investigation not only give us some performance bounds quickly, but it also yields insights to the relationship between microprocessor performance properties and task properties. In summary, our performance bounds

8.2 Application Scope

So far our discussion is limited to datapath with one RF, but it can be easily extended to include multi-RF architecture because:

- Our UB estimate assume no dependency, it doesn't matter which RF executes instructions . There is no RF transfer. We can treat it as more *M* attached to one RF with more LG to consider instruction distribution.
- Our LB estimate assume complete dependency. Though it is likely one RF execute some instructions faster and some slower than the other, we can account for RF transfers with by adding that fixed cost to functional units no in current RF.
- In reality, different RF are often used for different data representations. Therefore, there is only one or two RF to choose from. For example, integer, FP, and MMX have separate RFs.
- We assume there is enough registers in an microprocessor that register spilling will not happen. This argument is an observation on media processing characteristics – data flow and few branches. There are a few input or output variables representing streams of data. All intermediate variables (represented by internal processing nodes) have very short lifetime such that simultaneous live variables are small. There are very few control variables in a dataflow. Most contemporary microprocessors have more than 32 registers.

In addition we can extend our model to include effect load/store unit and cache, that is, the data transfer instructions. This is equivalent to covering all three-address instructions. If a register file cannot hold all life variables, then variables will spill to cache, them memory. Load/Store unit's throughput is fixed. If this number is smaller than either the consumption and/or production rates of our function, then the function in memory bound.

We have not considered estimates for SIMD-capable microprocessors. The reason is given as little information as the number of operations, the input sizes, and the operations, without dependency. We cannot possibly detect variable packing opportunities. Even with dependency, it still remains an open question. However, if the variables are already packed by hand, then we can obtain estimates the same way.

No MMX, but if data alignment instructions is in the task representation, we can still

account for it. The problem is to pack the data automatically. For microprocessors with SIMD hardware and instructions, such as MMX or any segmented datapath, we assume that no compilers can automatically detects SIMD parallelism. Therefore, we assume hand-coded library functions exist.

8.3 Front End Tool

So can do we use performance bounds for reconfigurable computing? From our conclusion in the last section, we know what makes good speedup candidates evaluated against a particular processor. We conclude that a good speedup candidate is complex, it requires operations the microprocessor has very few resource for and does it slowly, it has few memory accesses.

These conditions put us in a limited area to explore reconfigurable logic's performance benefits. They also suggest that complexity will become an issue on top of the unpredictable nature of reconfigurable logic. Therefore, our performance analysis will help to explore new areas of performance benefits with more certainty about the prospect of performance benefits. In particular, we contemplate two applications scenarios as suggested in Chapter 1.

1. When we are designing an embedded reconfigurable system (chapter 1) and are wondering which microprocessor core to integrate with reconfigurable logic on the same chip, we can make quick estimates with a few intend applications to choose the right combination. In addition, once we know the bound estimates, we can choose the memory devices so that memory bandwidth bottleneck will not become a oversight. On the other hand, we don't want to use more expansive than necessary.
2. When we have a new computer with a integrated reconfigurable microprocessor, we would like to develop more applications. This scenario requires exploring the partitioning space on a per application basis. In this case, we have multiple tasks of potential performance gains. The task for the largest potential gain may not be achievable due to the resources constraints. However, the second, third…, areas may offer some performance gains.

In both cases, we need to explore design space, find the right SW/HW partition. This process is iterative. With our conclusion that complex operations are more likely speedup candidates, the complexity will be equally thrust upon reconfigurable design. Thus we must reduce unnecessary efforts.

Our bound estimates can be integrated into design tools. With more information on the reconfigurable logic architecture, we can make better estimates. For example, Figure 8-5 shows how our estimates are used in a design process.

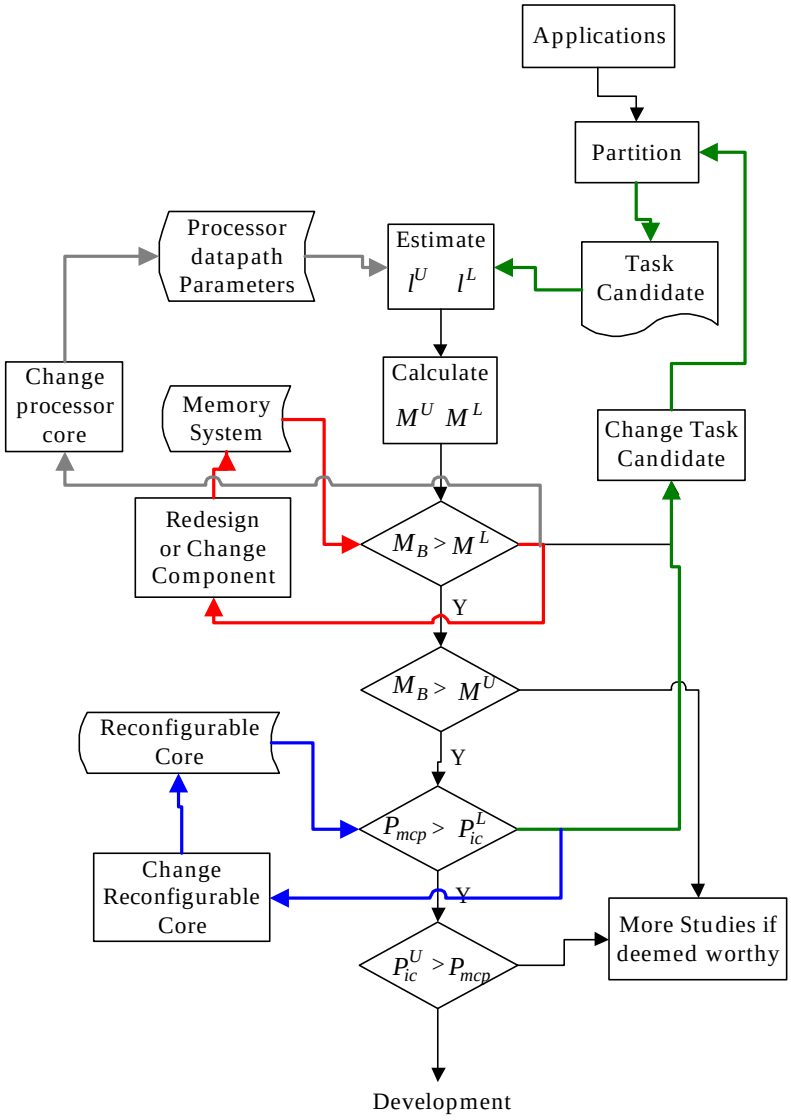


Figure 8-5: Front end for new reconfigurable system design

In this case, the problem is given a few initial applications we want to time-multiplex on a system, what is the best combination of a microprocessor core and some amount of reconfigurable logic with know architectural features. We imaging in the future, integration will be on a "on-demand" basis. That is, we can integrate cores of different architectures for different applications under different cost considerations.

Figure 8-5 shows that we can explore the system design space along four axes of variables – the processor core, the reconfigurable core, the speedup task, and the memory system.

The second scenario is critical to the concept of reconfigurable computing. If it is meant to provide added performance on top of a general-purpose computer for the mass, we

must be able to discover more applications for speedups. In this case, it does not even matter if the reconfigurable logic is integrated with the microprocessor. We can modify our performance properties accordingly. The main issue is given a system already equipped with reconfigurability, how does application developers explore it.

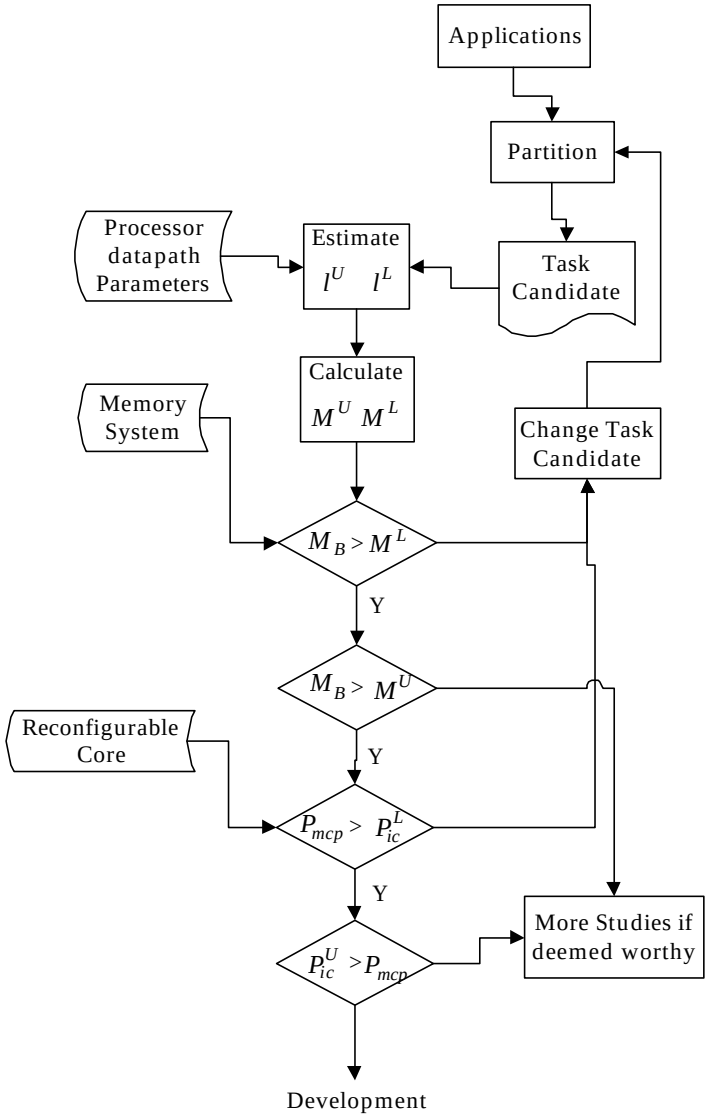


Figure 8-6: Front design flow for reconfigurable application development on existing systems.

Considering the reconfigurable is already fixated in the system, unlike the case in Figure 8-5 , where it is a variable. Now the physical limitations of the available resources in the reconfigurable logic becomes hard constraints. We have to contend with potential resource constraints on top of speedup requirements. Meeting these two contradicting requirement is no easy task. In addition, average developers are not experts in both software and hardware. They should have some intuitive idea about the speedup

candidates without writing profiling codes.

Thus our performance estimates can serve as front end tools to these developments.

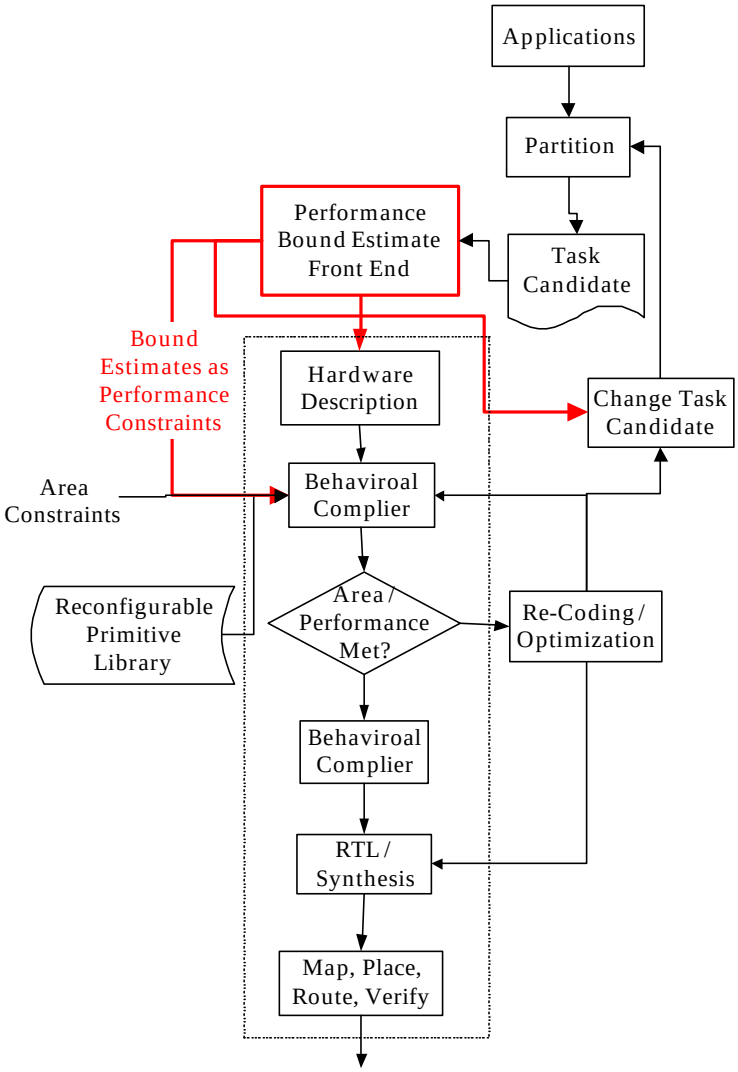


Figure 8-7: Performance bound estimates as a front end tool integrated into hardware design flow.

Figure 8-7 shows how our performance estimates can be integrated into the reconfigurable logic design flow After our estimates suggest that the target is a speedup candidate. We translate the task into a high-level hardware description. In fact, with more specificity, a DAG can serve as a high level hardware description. We use the upper bound as a performance constraint to the behavioral compiler.

A behavioral compiler can quickly estimate area and speed based on hardware primitive library from the reconfigurable logic. These are not accurate estimates since the design has not been placed and routed at this point. However, the area estimate provides the requirement for logic resource

If we are developing a reconfigurable application as in Figure 8-6, both area and performance goal must be met. If initial constraints are not met, we can refine either refine the description to explore the design space, or we can change a speedup candidate. If our are designing a new system as in Figure 8-5, we can opt for a bigger reconfigurable logic device, or a different architecture with better performance fit to the task.

9 Recap, Contributions

We set out by putting performance gains as a hard requirement for reconfigurable computing to be a viable computing paradigm. We envisioned reconfigurable logic will be incorporated ubiquitously in future embedded and general-purpose computers. To that end, we must achieve two conflicting requirements – high performance and programmability.

Past reconfigurable systems showed us promises of performance benefits. They failed to deliver the programmability part, though that was the other half of what reconfigurable computing is all about (Appendix A).

We re-think how performance is improved in a general-computer, and what make it general-purpose. We realize that a number of issues stand out with the coming of reconfigurable computing (Chapter 3).

Programmability involves ease of programming, which turns into more applications. We try not to make reconfigurable application development more difficult than it already is. We do this by minimizing dedicated hardware to the reconfigurable logic, which creates its own organization within the general-purpose organization (Chapter 7).

If we are talking about performance "gains", we must have clear reference to make our comparison. We do this by examining what are the properties that affect performance of a programmable device, in particular, microprocessors and reconfigurable devices. We eliminate the unfair advantages due to different technological backdrops in fabrication processes. Instead we focus on the architectural aspects. We realize that, for programmable devices, comparison can only be made under the context of a task (Chapter 5 & 6).

The difficulty is we cannot exhaustively make comparisons for an unbound number of tasks. Instead, we use a class of tasks, media processing, exhibiting some common characteristics as targets. The commonality makes our result of comparison more applicable when we keep the task specificity to a minimum (Chapter 4).

Media processing characteristics also allow us to "visualize" how tasks are executed on the microprocessors and reconfigurable devices. We view the processing as flow of data through processing nodes from input to output. In the case of microprocessors, this flow is visualized as instructions flowing through functional units replicated in temporal domain (Chapter 4 & 6).

We realize we can estimate the execution time required N instructions to flow through this fabric of functional units with a few variables and knowledge about the limits of these variables as they are confined by the law of physics and performance trade-offs (chapter 6 & 8).

We then can formalize performance estimates in mathematical forms. Our performance estimates use minimum information about the task. In fact, they don't even require

dependency information. The lower bound estimate reflects the effects of the complexity of the task and the total latency of the functional units. The upper bound reflects the effects of the "cardinality", the operator type, and the pipeline intervals of the functional units.

Using performance bound estimates, we check potential memory bottlenecks under the general-purpose system architecture we proposed earlier. We conclude that good speedup candidates require many microprocessor instructions to execute, they are slow to execute, and/or the microprocessor has very few such functional units (Chapter 8).

Finally, our performance bound estimates can be incorporated, as a front tool, into design of reconfigurable system or application development on a reconfigurable computer (Chapter 8 & Appendix B).

Contribution

This thesis proposes a "general-purpose" architecture intended to address the issues of too much customization, difficult to program in past systems.

The analysis of processor datapath performance is unique in that it takes the analytical route as opposed to the usual simulation or emulation route. The analysis formulizes microprocessor performance and shows how it is affected by the task and processor properties. The processor properties reflect the effects of architecture and fabrication technology. The findings of this analysis concur to experts' consensus, though through experience not through analysis.

Finally, this thesis tie up different and difficult issues in hardware architectures, applications, and design tools under one context and presents a more complete view to these issues.

A Reconfigurable Computing Systems

A.1 Custom Computing Machines

In 1960, Estrin described a "Fixed-Plus-Variable" structure system consisting of a high-speed general-purpose computer (the fixed part) operating in conjunction with a second system (the variable part) comprising large and small high-speed digital substructures [64]. The "variable" part was designed to compute the eigenvalues and eigenvectors of real symmetric matrices [65]. Due to technological constraints, his "variable" substructures had very limited variability and thus limited applicability. The introduction of the first commercial FPGA by Xilinx in 1986 rekindled Estrin's idea. A flurry of activities based on the idea of using FPGA as the "variable" or "reconfigurable" structure soon followed. Three most significant efforts, where actual systems were built, applications were developed, and performances were evaluated, are the PAM [15, 66], the Splash [14, 18, 67], and the PRISM project [60, 68].

These early systems were called custom computing machines (CCM[27]) because the FPGAs could be "customized" to the target applications. A typical CCM consisted of a general-purpose computer and a FPGA-based subsystem. The "customizable" FPGA subsystems were attached to either a high-speed I/O (or peripheral) bus [14, 15, 18, 66, 67], or a host processor's local bus [60]. These systems all reported performance increases from single digit to thousands folds. The reference systems ranged from single-issue processor based computers to massively parallel supercomputers.

Table 9-1 tabulates some information about these system and performance claims. This list is not comprehensive to provide reciprocal evaluations of the sources of performance gains. Some relevant pieces of information are not reported in literature, such as the host system's CPU and its clock rate, memory bandwidth, size of cache, the reconfigurable subsystem's local memory, clock rate, etc.

A.2 Multimedia Reconfigurable Computing

Driven by a need for a fast acquisition, processing and display machine for research in digital video and model-based representations of moving scenes, Bove et al. started building a prototype programmable video processing system - Cheops in 1989 [58]. Cheops abstracted a fixed set of computationally intensive kernel functions from common video processing algorithms and embodied them in specialized hardware. It sped up overall performance of applications using these kernel functions. Though Cheops performed well for its initial intended applications, its performance edge could not be extended over to other algorithms not using these kernels. A reconfigurable

[27] We define CCM as a machine with a general-purpose subsystem and a dynamically reconfigurable subsystem.

subsystem, State Machine, to replace the specialized hardware was later designed [69, 70]. The assumption was that reconfigurable hardware could provide both programmability and significant speedups over general-purpose processor based architectures for multimedia applications.

State Machine was partially functional and could be reconfigured. However, it was built for the Cheops Imaging system as an attached subsystem. By that time, it was clear that Cheops no longer had the performance advantage over then a state-of-the-art general-purpose system. In addition, it was very difficult to debug, maintain, and develop new applications. Cheops was a prototype system thus it was not completely bug-free, stable and reliable[28]. A PCI based reconfigurable subsystem, CHIDI, was thus built to replace State Machine [59]. CHIDI had design several problems and was not functional[29].

A.3 Myth of Performance Claims

Leveraging on existing general-purpose systems' infrastructure (system architecture and organization, microprocessor architecture, memory subsystem, bus standards, programming tools etc.) and commercial FPGA devices, early CCMs were built as add-on subsystems. This allowed researchers to avoid enormous amount of time and expertise required if they were to re-think and re-build a GPRS[30] from scratch. However, this ad hoc approach also limited their ability to explore better architectures and organizations not bound by existing technology. As a result, very little experience was gained in exploring new architectures and organizations of future reconfigurable systems.

The last row of Table 9-1 shows speedups reported in [14, 45, 60, 68]. Though these performance claims seemed to suggest reconfigurable hardware's viability as a performance enhancing computing substrate, careful examination raises questions on the specifics of reference platforms and the magnitude of the speedups.

We found that these claims were not based on well-defined reference platforms and were very loosely documented in the literature. None of these reports included a detailed information on all performance-related parameters. Typically, some of the following information is missing

1. Reference system processors, their operating frequency, and semiconductor processing technology.
2. Organization and operating frequencies of the reference systems' cache and memory subsystems, as well as the semiconductor manufacturing process.

---

[28] A few Cheops main boards were built, each one in different states due to manufacturing, usage, and engineering change orders.

[29] See [59].

3. Reference systems' local bus and peripheral (I/O) bus performance characteristics.
4. Reconfigurable logic devices' (FPGAs) manufacturing technology (for comparison with the host processor). This information is only obtainable from the vendors.

The aforementioned points suggest that it is likely that the system components used in the reconfigurable subsystems were based on newer manufacturing technology with smaller feature sizes and thus faster switching speeds. These components include the FPGAs, the memory, peripheral components, and glue logic.

Our interests is in FPGA's advantages of being logically and structurally reconfigurable, allowing it to match task characteristics. We are not interested in the advantages of the newer manufacturing technology, which can be neutralized by process upgrades.

In addition, these reconfigurable subsystems had more degree of freedom in organizing memory and I/O devices. Typically, widening the memory word and coupling it with I/O specifics could increase data transfer rates not achievable in the reference platforms. However, this freedom of organization was not necessarily an intrinsic deficiency of the reference systems itself as they must maintain compatibility of components from different technology generations. It was even less a deficiency of the host processor for the same reason. In fact, strong coupling among the FPGA, the memory, and the I/Os makes the subsystem more application-specific less adaptable to other applications. This undermines reconfigurable systems' viability as programmable systems.

Our concern is that performance gains were greatly exaggerated when we compare squarely a microprocessor with RLA for a target task. A naïve interpretation of these numbers may mislead novice system designers or application developers into expecting the need and magnitude of performance benefits from reconfigurable hardware. It can cause them setting unrealistic performance goals and consequently significant loss in time and financial resources when design decisions are based on such misinformation. The following section discuss these points in greater details.

A.3.1 System Organization

PAM and Splash were simply "attached" to an I/O bus of a general-purpose host machine. There was a clear physical and logical boundary that separated the overall system into a general-purpose host system and a reconfigurable subsystem. One such typical reconfigurable subsystem could be physically and functionally separated into a data processing part, performed by the reconfigurable hardware, and a data transfer part, performed by three subcomponents: local high-speed I/O, fast local memory, and host system interface (see Figure 9-1). These components involved moving data in and out of the reconfigurable hardware and/or the reconfigurable subsystem.

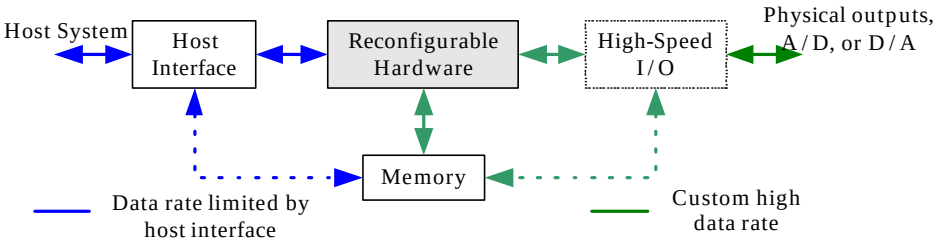


Figure 9-1: A typical functional datapath block diagram of a reconfigurable subsystem on early custom computing machines.

Though CCM designers could improve the speed of each data transfer component, ultimately it was the access to the host system memory (if the application dictates) from reconfigurable subsystem, or the access by host CPU to the reconfigurable subsystem's local memory that became the bottleneck of data transfer. This bottleneck is fixated at the speed of the host bus connecting to the reconfigurable subsystem. For most early CCMs, it was the host system's I/O bus. Whether I/O bus speed can scale with semiconductor technology upgrade is an open question and beyond the scope of this thesis. Matter-of-factly, an I/O bus has been the last part of a system that will be brought up with the capability of current technology. Over the years, the speed gap between I/O bus and processor local bus has grown wider with technology generation [53].

Memory

Local memory was considered a must to reduce access latency exactly due to the inefficiency of this attached architecture [15]. SRAM memories were used for these subsystems because of their faster access time compared to slower DRAM used in general-purpose systems. However, the makeshift organization blindsided the fact that performance of a general-purpose system would also improve if the same amount of SRAM had been used in its memory system as cache or as main memory. DRAM was the choice of memory because of its cost (and the infrastructure built around it), not its performance. Given the fact that the state-of-the-art microprocessor at that time typically had no more than 8k to 16k cache and 1M to 4M DRAM, the amount of SRAM memory used in these systems could also be used for cache and main memory.

We do not know some of the host systems' memory bandwidths. Memory bandwidth is determined by the width of a memory word and the inverse of the memory cycle time. Typically, the memory cycle was much longer than CPU's clock cycle. Even if it ran as fast as the CPU, the bandwidth for a 32-bit bus would be around 100 MB/s (25 MHz memory cycle time, DecPeRle-1). This was only ¼ of the local system memory bandwidth on DecPeRle-1. Therefore, the performance claims possibly included a significant contribution from the local memory alone[31]. However, the same amount of

[31] It depends on the applications as well.

SRAM could also improve the host system when used as cache and/or system memory.

I/O

The local memory could also help regulate differences in data processing rates among the I/O devices, the reconfigurable hardware, and the host system. Most CCMs built in high-speed I/O ports to connect to external high-speed I/O devices. Figure 9-1 shows the reconfigurable hardware can be connected to a logic device implementing a particular fixed-function standard I/O protocol, or it can implement the protocol using reconfigurable logic. In the former case, the reconfigurable hardware implements some kind of upstream or downstream processing logic for a particular I/O protocol, thus limiting its scope of applications. In the later case, the reconfigurable hardware can implement any (limited by the resources) protocol and necessary processing, thus broadening its application domain.

Regardless of the actual organization of a reconfigurable subsystem, one must be careful in making performance claims, when comparing a CCM with a general-purpose system, and drawing conclusions from such claims. Component contributions to the overall performance measurement must be separated though this is very difficult if not outright impossible.

A.3.2 Fabrication Technology

DecPeRle-1 was attached to a 25 MHz CPU host system, while some applications implemented on FPGAs could run as fast as 33MHz [45]. This raises the prospect that the CPU fabrication process might have predated the FPGAs fabrication process.

The comparison of "raw" processing power of fixed-logic processor versus reconfigurable logic without the advantage/disadvantage of fabrication technology will be invariant to technology. Therefore, the comparison should be based on equal technological backdrop (Figure 9-2).

Comparing devices made from different technology generations confuses this issue. It is not clear what host system was used for performance comparison in DecPeRle and Splash CCMs. In fact, none of them separated the system's contribution to the speedup from the FPGA's effect. In particular, the semiconductor process generation effect was not accounted for.

Central to this raw processing power is the maximum clock frequency as it determines the theoretical maximum throughput of each processor[32]. In chapter 5, we present some theory and argue that reconfigurable device can never achieve a higher operating clock

[32] Clock frequency is not the only factor determining the theoretical maximum achievable throughput. However, modern high-speed processors, general or application-specific, all deploy pipelined functional units to increase throughput, it is safe to say this practice will continue as transistors' switching speed becomes faster and faster.

frequency than a general-purpose processor fabricated with the same process technology. The difference in the maximum clock rate can diminish much of the reconfigurable hardware's advantages.

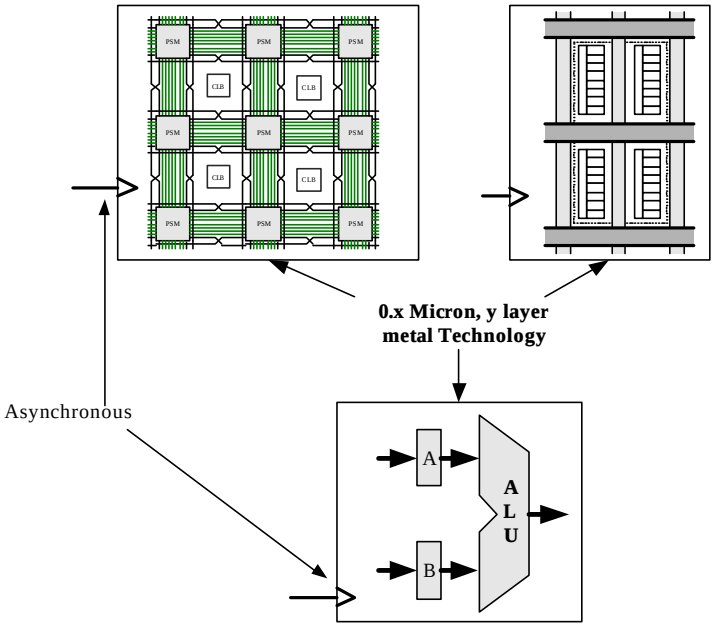


Figure 9-2: Performance comparison is more fairly judged using the same manufacturing technology.

Table 9-1 also shows that local memories used in different systems had different speed grades, possibly the result of different technology generations.

A.4 Discussions

We raised two points on previous performance claims – the effect of "local" organization and the processing technologies, on which the FPGA and the microprocessor as well as other system components were fabricated. These factors contributed to the overall performance gains. Many reports seem to misrepresent their claims in lumping all contributions from different factors.

We must understand where the performance gains come from and whether each component's contribution is invariant of the manufacturing technology. In the context of reconfigurable computing, our focus is on the advantages of a reconfigurable logic array as a system component over a microprocessor core.

A fair performance claim is based on all things equal except the ones under comparison. An ideal scenario is that we could simply replace the CPU in the reference system with a FPGA device, with everything else the same, and measure the speedups. A microprocessor based programmable add-on subsystem may compare favorably in

performance (for a set of applications) to a reconfigurable subsystem in Figure 9-1.

A even more fair comparison would be to require both devices to be fabricated using the same processes and the same silicon area. The same area requirement is based on economics. However, we feel that this is not a strong requirement since the cost of real-estate can be justified by its added value – performance. Figure 9-3 shows this "drop-in replacement" concept.

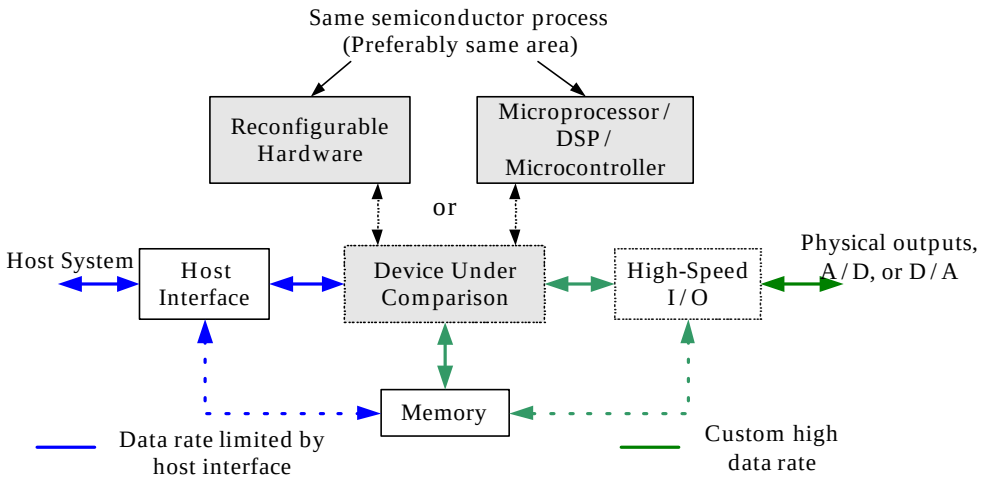


Figure 9-3: Drop-in replacement for performance comparison.

However, Figure 9-3 is most likely impossible to achieve since interfacing properly to the two candidates under test require different set of components. Nonetheless, we could try to make all things equal by requiring equal data word sizes. This requirement ensures that organizational advantage be eliminated.

We must base our comparison on the same semiconductor process generation so that architectural merits or deficiencies of general-purpose and reconfigurable architectures can be detached from the fabrication technology. We will examine whether these architectural merits or deficiencies are still valid today and whether future technology advances will change the dynamics of their relative standings. This will help system and microprocessor architects, FPGA architects, and reconfigurable application developers to optimize their efforts for their respective design goals in the future.

In addition, the performance dynamics have changed quite markedly in the process technology, microprocessor architecture, and micro-architectures. These advances will likely to make performance gains from reconfigurable logic even more difficult to obtain (i.e. requiring careful implementation).

Over the last ten years, the clock speed of a microprocessor has increased 50 folds from

33 MHz to 1.7 GHz. The estimated maximum clock speed[33] of a FPGA has increased only 6 folds from 33 MHz to 200 MHz. The rates of increase in maximum clock speed are not the same. Though these numbers are crude and depend on FPGA configurations, FPGAs' maximum clock speed appears not following the Moore's law. In other words, if we were to plot the clock speed for microprocessors and FPGAs versus technology generations, we would find the slopes of the microprocessor curve are higher than the slopes of the FPGAs. This discrepancy can change the relative performance of microprocessors and RLA over time.

One question regarding to this phenomenon is whether there is a fundamental physical underpinning causing this scaling disparity. Our argument is that the reconfigurable logic array, with its regular structures on programmable logic and routing, cannot optimize beyond the fixed placement of these resources. Whereas process upgrades can change the delay dynamics in critical paths in a custom design and make further optimization possible. This observation is backed up by the fact that only 25% of improvement is due to process improvement and another 25% is due to design innovations, such as circuit design and physical organization [46, 71]. For FPGAs, such low-level innovations are very limited due to its reconfigurable logic and routing requirements (for more details, see Chapter 5).

The difference in the rates of change of this underpinning suggests three things. First, the magnitudes of performance benefits reported in early systems would probably decrease if we simply scale the same (architecturally and organizationally) systems with new process technology. Second, FPGA designs probably require more careful optimization for hard-earned speedups. Novice designs or designs using automatic compilation tools from high-level descriptions may reduce the achievable performance benefits to a marginal or even a negative level. Third, the software-hardware partitioning boundaries may require re-consideration from technology generation to generation.

On the architectural front, new microprocessor architectures have emerged with new application domains. DSPs, media processors and multimedia extensions of general-purpose microprocessors are such examples. Using these microprocessors as the general-purpose processor in a reconfigurable computer may further reduce the performance benefits from reconfigurable logic for media processing applications.

Lastly but not the least, micro-architectural improvements, such as branch prediction (BP), superscalar (SS), out-of-order execution (OOE), dynamic register renaming, exploited all forms of instruction-level parallelism (ILP). In the author's opinion, this trend leaves little room for small pieces of reconfigurable hardware, which can only accommodate a few operators, to outperform a contemporary microprocessor. This is the argument for deploying medium to large number of reconfigurable logic gates in a reconfigurable system.

---

[33] FPGA's clock rate (if not already fixed in system) depends on applications. The maximum clock rate represents a realistic maximum for all applications.

Table 9-1: System information on several reconfigurable subsystems.

Reconfigurable Subsystem	Prism-I	Prism-II	DecPeRle-0	DecPeRle-1	Splash 1	Splash 2	CHIDI
Host Workstation	Sun-3	Sparc IPC	-	DEC 5000/200	Sun-3 Sun-4	Sparc II	#
Host System CPU	68030	68040	-	MIPS R3000A			PowerPC 604
On-Board CPU	Motorola 68010	AMD AM29050	None	None			
CPU clock rate (MHz) FPGA	10 XC3090	33 XC4010	- XC3020	25 XC3090	None XC3090	None XC4010	266 Fllex10k-100
Number	4	3	25	23	34/32	17/16	1
Topology	[68]	[60]	5x5	4x4	Linear array	16x16 Crossbar	-
Aggregated Number of Gates	40k	-	50k	200k	-	160k	100k
Aggregated Number of Registers	1.28k	2.4k	3.2k	14k	12.4k	12.8k	4992
Aggregated Number of CLB	320x4	400x3	128x25	400x23	320x32	400x16	4992
FPGA clock rate	-	-	Variable	Variable	Variable		Variable
FPGA maximum clock rate (MHz)	-	-	25	10-33	32	40	-
Local Memory Size (MB)	0.5	4	0.5	4	4	8	2
Local Memory Bandwidth	150ns access time	-	200MB/s	400-640MB/s	50ns SRAM	50ns SRAM	9ns SRAM
Communication Bus to Host	-	-	VME	turbochannel	VME	S-bus	PCI
Communication Bandwidth to Host (MB/s)	-	-	8	100	8	-	264
Speedup Claims	2.9-54	6.34-86.30	-	10-1000	-	10-1500	-

B FPGA Design Issues

Several FPGA design related issues were also raised during State Machine and CHIDI's development. These issues involved how FPGAs would be used in a system, the design constraints, the design methodology, and development tools used. They are representative of the complex interactions among different requirements and constraints of the development process. Underlying these interactions is the ever-growing complexity in semiconductor devices that comes with the shrinking minimum feature size. FGPA devices not only grew in the number of logic and routing resources, but also their architectures and micro-architectures evolved in order to adapt to new application domains. (For example, Xilinx's XC4000 FPGA devices contained embedded memory (SRAM) and different logic cell and routing architectures than XC3000 devices [72] [73].)

Architectural changes require new CAD tools (new tools to generate or synthesize memory control logic, new optimization and technology mapping tools for different logic implementation, new place and route tools for different routing structures). For high-performance designs or designs under hard constraints, FPGA designers often must understand the FPGA architectures and CAD tools' capability to meet their design constraints[34]. State Machine and CHIDI's (as well as other FPGA-based reconfigurable systems) development issues manifest the dynamic nature of a fast-changing technology base, which in turn seeds new applications, which then sets new requirements, which needs a new development methodology and the tools to help reap the benefits of technological advances.

From the experience of State Machine and CHIDI as well as other efforts in the literature, we examine how FPGA devices have been used in the past, and the development process and methodology supporting reconfigurable computing applications. We also conjecture possible reconfigurable hardware's use in the future – in a general-purpose environment and in an embedded environment. Our conjecture is based on the strength and weakness of reconfigurable hardware relative to those of general-purpose architectures.

B.1 Design Issues

Normally in its traditional use, FPGAs are used in fast prototype systems or low quantity production systems as a low cost substitute for ASIC. The target applications are known at the beginning of the actual design of a system. Designers know and specify the exact functionality to be implemented on FPGAs. They then make an initial

[34] An analogy to high-performance software development, one must know the instruction set details of the processor and how compiler works so that he can write codes in a style that exposes parallelism (that compiler can detect) to the compiler, or one must write assembly codes. In a FPGA design, unlike software programmers, some vendor's CAD tools do not give designers control of low-level details (or not all details).

attempt at implementing the functionality on the FPGAs. That is, they implement either a complete or a partial (often the potential trouble part) design; estimate the size and performance of the design with the help of FPGA vendor's software; then select the FPGA devices that can meet these requirements. The actual board fabrication is normally started after going through these steps[35]. At that time the designer should have much deeper understanding in the implementation details and thus should have higher confidence in making implementation decisions.

This design methodology is a result of accumulation of past experience shared by many experts. It varies with the difficulty of the application itself, the devices used, the experience of the designer, and the CAD tools. In some cases, the FPGA design should be almost completely designed and verified before selecting a particular FPGA device for board fabrication. In other cases, one can bypass this stage if the application is simple[36] or he is very experienced and is confident in his decisions.

One common mistake is for designers, who have done SPLD and CPLD designs, to commit to fabricate an FPGA-based system board without going through some of the abovementioned steps. Unlike PLD devices, a FPGA design was most noticeably different in its unpredictability of routing delays. For SPLD and CPLD devices, delays are predictable [74]. One can design and fabricate a system board with a CPLD device without actually finishing the logic design first since its minimum clock period can be estimated from the beginning[37]. This is much more difficult to do with FPGA, especially under performance constraints.

In the case of CHIDI, there were two logical FPGA devices[38]. One contained fixed functionalities such as bus interface, memory controller, and data organizer (or data shuffler, data packing/unpacking). The second logical FPGA device was to be used as a reconfigurable computing substrate, whose function was to be developed after board fabrication on a per application basis. The bus interface imposed certain requirements on the fixed-function FPGA device (bus protocol and signaling speed, etc). It must meet a set of hard performance constraints set by the operating environment of the hosting system.

These two different usage scenarios represent different timing of functionality (the actual FPGA design is in the form of a configuration) and performance specification. For the fixed-function FPGA, the binding of its specification and performance happens before the hosting system board is fabricated. This means one is free to choose a FPGA

---

[35] Of course, if cost is not a factor, but time-to-market is. Then one can always select the fastest and largest possible device as a way of "defensive" design.

[36] For example, performance is not important.

[37] Assuming the number of logic cells are more than necessary to implement the application.

[38] Here the distinction is based on whether the device is to be reconfigurable for different application or not. One logical device may contain multiple physical devices.

device with appropriate resources (number of logic cells, routing capacity, number of I/O pins, etc.) to meet this functionality and performance specification. If one particular FPGA device lacks in some kind of resources, one can "bump up" his selection to one that has more such resources. Of course, this is limited by the availability of different FPGA devices offered by the vendors. The development methodology of fixed-function FPGA was relatively well established[39].

For the reconfigurable FPGA, the binding of its functionality and performance occurs after the hosting system is fabricated. This means a FPGA device is chosen somewhat randomly for unknown functionality and performance characteristics. For whatever reasoning behind the choice of this reconfigurable FPGA, it's architecture, resources, and performance characteristics become constraints or limitations for its applications. For reconfigurable computing application development, there was no well-established methodology to support it.

Two constraints almost always come up in hardware design - speed and area constraints. For system designs using FPGAs as components, these constraints must be carefully considered against selected FPGA devices if board fabrication time is to precede or parallel the FPGA design.

In particular, for high-performance designs, where we may want to exploit every possible bit of performance out of a FPGA, we may want to know what is the maximum achievable clock rate for realistic designs. This question is difficult to answer due to FPGA's unpredictable routing delays. It is often compounded by the vendor's proclaimed maximum clock speed using a few pedagogical designs. In addition, high-speed designs translate into area trade-off. CHIDI encountered both problems.

A FPGA device (FLEX 10K50) chosen to interface with a 66 MHz local bus could not run at speed. That was even after very careful hand-optimization. This is a result of the designer's not understanding the FPGA's limitations (see Section 5.2). Though FPGAs' routing delay is very unpredictable, it cannot run as fast as one wishes. So how do we know their speed limits?

The second problem CHIDI encountered was its aggressive area estimation at the beginning of its design. A FPGA device was selected to implement a bus interface and various data organization mechanisms. An area estimate was made based on the number of sequential elements, such as flip-flops, were needed in the datapath and state machines. However, the number of logic cells needed to implement the combinational part can exceed the number of estimated registers. This situation arises when there are high fan-in Boolean functions in the design. A logic cell has a fixed number of inputs to its combinational circuit that implements a combinational function. High fan-in

---

[39] However, there were always the issues of supporting the latest architectures and features in new FPGA devices by the vendors. Typically, new FPGA devices are available before the tools that optimize for their architectures and features become available. In addition, automatic synthesis from a high-level behavioral description of the circuit faces the same problems a software compiler faces – it cannot infer all hardware features without introducing errors.

combinational functions may require more than one logic cell (and thus multi-level) to realize.

Combinational delays are difficult to estimate by a human designer. In addition, for high performance designs (i.e. performance requirement close to the limit of the FPGA), pipelining and signal buffering are two most often used techniques to increase throughput. Both require more logic cell resources, making the initial tight resource budget even tighter. Most of all, tight logic cell budget can force placement tools spread clustered logic into unused logic cells located in different rows and columns. This either makes routing delay longer or the design cannot be routed at all.

CHIDI's problems manifested the particular issues of "putting constraints" before applications development. This is exactly the issues reconfigurable application development must face.

B.2 Tool Issues

At the time of State Machine's development, a FPGA application development was considered as designing a piece of hardware. The capacity of a FPGA device had grown to hundreds of thousands of transistors or more. However, FPGA vendors offered only schematic or simple textual (a netlist of bit-wide register-level primitives and parameterized modules) entry methods, more suitable for SPLD (Simple Programmable Logic Device) and small CPLD (Complex Programmable Logic Device). The largest FPGA (then ORCA2C40) had equivalent gate capacity of forty thousand gates. The complexity of this scale rendered traditional design entry methods for SPLD (Simple Programmable Logic Device) and CPLD (Complex Programmable Logic Device) difficult to enter, explore, manage, and modify. Furthermore, CAD tools vendors were barely able to keep up the pace to providing solutions to their majority market - ASIC development, which was also growing exponentially in complexity. The FPGA tools was much more behind what the hardware was capable of.

The initial FPGA design methodology at that time was a two-stage, non-synergistic, makeshift process. A third party CAD tools provided high-level design entry, usually in the form a hardware description language, and logic synthesis tools with technology independent logic minimization. The FPGA vendors provided "foundry tools" for mapping a technology-independent netlist to a technology library, placement of logic functions and sequential circuits, and routing among these functions. However, traditional logic synthesis approaches were not suitable for LUT based logic functions.

Applying the traditional two-level logic synthesis and minimization techniques, which were more suitable for PAL-like logic structures, to LUT-based logic did not produce satisfactory results (this problem was amended later when CHIDI was designed). In addition, constraints entered at the front-end high-level entry tools could not be effectively translated into constraints for the low-level foundry tools. Furthermore, device specific features could only be instantiated at the back-end tools, making the front-end and back-end design descriptions inconsistent.

As a result, changes of design specification at the front-end tools could not be automatically propagated to the foundry tools. One must manually modify such changes. These problems were addressed by FPGA vendors and third party synthesis tool vendors later. The solution to logic synthesis was to have third party provide synthesis tools targeted toward LUT-based logic functions. Third party synthesis tool vendors and FPGA vendors agreed on a way to pass constraints from front-end tools to back-end tools. To address the ability to include device specific features (which HDL cannot infer) at the original design entry level, front-end tools incorporated compiler pragmas or compiler directives to encapsulate them from the high-level compiler as black boxes.

The lack of tool integration illustrates the difficulty of developing reconfigurable applications.

Bibliography

[1] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," presented at Design, Automation and Test in Europe, Conference and Exhibition, 2001.

[2] B. Radunovic, "An overview of advances in reconfigurable computing systems," presented at 32nd Annual Hawaii International Conference on Systems Sciences, HICSS-32., 1999.

[3] P. J. Fleming and J. J. Wallace, "How Not to Lie with Statistics - the Correct Way to Summarize Benchmark Results," *Communications of the ACM*, vol. 29, pp. 218-221, 1986.

[4] J. E. Smith, "Characterizing Computer-Performance with a Single Number," *Communications of the ACM*, vol. 31, pp. 1202-1206, 1988.

[5] R. Giladi and N. Ahitav, "SPEC as a Performance Evaluation Measure," in *Computer*, vol. 28, 1995, pp. 33-42.

[6] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *COMPUTER*, vol. 28-34, 2000.

[7] A. DeHon, "Reconfigurable Architectures for General-Purpose Computing," MIT A.I. Lab 1586, October 1996.

[8] A. Dehon, "Comparing computing machines," presented at Configurable Computing: Technology and Applications, SPIE Conference, 1998.

[9] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *Computer*, vol. 30, pp. 86-93, 1997.

[10] R. S. Razdan, M.D., "A High-Performance Microarchitecture With Hardware-Programmable Functional Units," presented at Proceedings of the 27th Annual International Symposium on Microarchitecture, 1994.

[11] J. R. Hauser and J. Wawrzynek., "Garp: A MIPS Processor with a Reconfigurable Coprocessor," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, 1997.

[12] D. A. Patterson and J. L. Hennessy, *Computer architecture : a quantitative approach*. San Mateo, Calif.: Morgan Kaufman Publishers, 1990.

[13] P. Faraboschi, J. A. Fisher, and C. Young, "Instruction scheduling for instruction level parallel processors," *Proceedings of the IEEE*, vol. 89, pp. 1638-1659, 2001.

[14] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," presented at 4th annual

ACM symposium on Parallel algorithms and architectures, San Diego, CA, 1992.

[15] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Transactions on VLSI Systems*, vol. 4, pp. 56-69, 1996.

[16] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal, "The RAW Benchmark Suite: Computation Structures for General Purpose Computing," presented at *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, 1997.

[17] J. M. Arnold, "The Splash-2 Software Environment," *Journal of Supercomputing*, vol. 9, pp. 277-290, 1995.

[18] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using a Highly Parallel Programmable Logic Array," *Computer*, vol. 24, pp. 81-89, 1991.

[19] P. Bertin and H. Touati, "PAM Programming Environments: Practice and Experience," presented at IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, 1994.

[20] A. Kumar and B. Waldecker, "PAT: state machine based approach to performance modeling for PowerPC™ microprocessors," presented at IEEE International Conference on Performance, Computing, and Communications, 1997.

[21] T. Chen, "The past, present, and future of image and multidimensional signal processing," in *IEEE Signal Processing Magazine*, vol. 15, 1998, pp. 21 -58.

[22] R. V. Cox, B. G. Haskell, Y. Lecun, B. Shahraray, and L. Rabiner, "On the applications of multimedia processing to communications," *Proceedings of the IEEE*, vol. 86, pp. 755-824, 1998.

[23] H. Samueli, "The broadband revolution," *IEEE Micro*, vol. 20, pp. 16-26, 2000.

[24] P. Suetens, P. Fua, and A. J. Hanson, "Computational Strategies for Object Recognition," *ACM Computing Surveys*, vol. 24, pp. 5-61, 1992.

[25] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Computing Surveys*, vol. 31, pp. 264-323, 1999.

[26] S. C. Becker, "Vision-assisted modeling for model-based video representations," in *Program in Media Arts and Sciences*. Cambridge, Mass: MIT, 1997.

[27] E. Chalom, "Statistical image sequence segmentation using multidimensional attributes," in *EECS*. Cambridge, Mass: MIT, 1998.

[28] V. Lappalainen, "Performance analysis of Intel MMX technology for an H.263

video H.263 video encoder," presented at The sixth ACM international conference on Multimedia, 1998.

[29] R. Bhargava, L. K. John, B. L. Evans, and R. Radhakrishnan, "Evaluating MMX Technology Using DSP and Multimedia Applications," presented at 31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998.

[30] D. Talla and L. K. John, "Quantifying the effectiveness of MMX in native signal processing," presented at 42nd Midwest Symposium on Circuits and Systems, 2000, 2000.

[31] D. A. Patterson, J. L. Hennessy, and D. Goldberg, *Computer architecture : a quantitative approach*, 2nd ed. San Francisco: Morgan Kaufmann Publishers, 1996.

[32] S. Devadas, A. Ghosh, and K. W. Keutzer, *Logic synthesis*. New York: McGraw-Hill, 1994.

[33] C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," *Journal of Vlsi and Computer Systems*, vol. 1, pp. 41-67, 1983.

[34] "Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference," Intel Corp. 1999.

[35] *PowerPC 604 RISC Microprocessor User's Manual*: IBM Corporation, 1997.

[36] G. Conte, S. Tommesani, and F. Zanichelli, "The long and winding road to high-performance image processing with MMX/SSE," presented at Fifth IEEE International Workshop on Computer Architectures for Machine Perception, 2000.

[37] J. Cong and Y. Ding, "Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, pp. 145-204, 1996.

[38] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, pp. 5-35, 1991.

[39] J. Cong and C. Wu, "An efficient algorithm for performance-optimal FPGA technology mapping with retiming," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 738-748, 1998.

[40] J. Cong and C. Wu, "Optimal FPGA mapping and retiming with efficient initial state computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 1595-1607, 1999.

[41] J. Frankle, "Iterative and adaptive slack allocation for performance-driven layout and FPGA routing," presented at 29th ACM/IEEE Design Automation Conference, 1992.

[42] Y.-L. Wu, S. Tsukiyama, and M. Marek-Sadowska, "On computational complexity of a detailed routing problem in two dimensional FPGAs," presented at Fourth Great Lakes Symposium on VLSI, 1994. Design Automation of High Performance VLSI Systems, 1994.

[43] W. N. Li, "The complexity of segmented channel routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, pp. 518 -523, 1995.

[44] S. Brown, M. Khellah, and Z. Vranesic, "Minimizing FPGA interconnect delays," *IEEE Design & Test of Computers*, vol. 13, pp. 16-23, 1996.

[45] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable Active Memories: a Performance Assessment," Digital Paris Research Laboratory 24, March, 1993.

[46] D. H. Allen, S. H. Dhong, H. P. Hofstee, J. Leenstra, K. J. Nowka, D. L. Stasiak, and D. F. Wendel, "Custom circuit design as a driver of microprocessor performance," *IBM Journal of Research and Development*, vol. 44, pp. 799-822, 2000.

[47] V. Betz and J. Rose, "How much logic should go in an FPGA logic block?," *IEEE Design & Test of Computers*, vol. 15, pp. 10-15, 1998.

[48] A. DeHon, "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)," presented at International Symposium on Field Programmable Gate Arrays, 1999.

[49] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, pp. 41-+, 2000.

[50] Y.-S. Tung, C.-C. Ho, and J.-L. Wu, "MMX-based DCT and MC algorithms for real-time pure software MPEG decoding," presented at IEEE International Conference on Multimedia Computing and Systems, 1999.

[51] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, pp. 24-36, 1999.

[52] F. P. O'Connell and S. W. White, "POWER3: The next generation of PowerPC processors," *IBM Journal of Research and Development*, vol. 44, pp. 873-884, 2000.

[53] R. S. Tetrick, B. Fanning, R. Greiner, T. Huff, L. Hacking, D. Hill, S. Chennupaty, D. Koufaty, S. Palacharla, J. Rabe, and M. Derr, "A Discussion of PC Platform Balance: the Intel Pentium 4 Processor-Based Platform Solutions," *Intel Technology Journal*, vol. Q1, 2001.

[54] J. L. Hennessy and D. A. Patterson, *Computer organization and design : the hardware/software interface*, 2nd ed. San Francisco: Morgan Kaufmann, 1998.

[55] B. Burgess, N. Ullah, P. Vanoveren, and D. Ogden, "The Powerpc-603 Microprocessor," *Communications of the ACM*, vol. 37, pp. 34-42, 1994.

[56] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan, "Superscalar Instruction Execution in the 21164-Alpha Microprocessor," *IEEE Micro*, vol. 15, pp. 33-43, 1995.

[57] S. P. Song, M. Denman, and J. Chang, "The Powerpc-604 Risc Microprocessor," *IEEE Micro*, vol. 14, pp. 8-17, 1994.

[58] V. M. Bove and J. A. Watlington, "Cheops - a Reconfigurable Data-Flow System for Video Processing," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, pp. 140-149, 1995.

[59] V. M. Bove, "Chidi: The Flexible Media Processor,". *http://chidi.www.media.mit.edu/*, 1997.

[60] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh, "PRISM-II compiler and architecture," presented at IEEE Workshop on FPGAs for Custom Computing Machines, 1993.

[61] R. Razdan, "PRISC: Programmable Reduced Instruction Set Computers," in *Applied Sciences*. Cambridge: Harvard, 1994.

[62] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *Computer*, vol. 33, pp. 62-69, 2000.

[63] A. Rashid, J. Leonard, and W. H. Mangione-Smith, "Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 1998.

[64] G. Estrin, "Organization of Computer Systems: The Fixed-Plus Variable Structure Computer," presented at Western Joint Computer Conference, New York, 1960.

[65] G. Estrin and C. R. Viswanathan, "Organization of a ``Fixed-Plus-Variable'' Structure Computer for Computation of Eigenvalues and Eigenvectors of Real Symmetric Matrices," *J. of ACM*, vol. 9, pp. 41-60, 1962.

[66] P. Bertin, D. Roncin, and J. Vuillemin, "Introcuction to Programmable Active Memories," Digital Paris Research Laboratory 3, June, 1989 1989.

[67] D. A. Buell, J. M. Arnold, and W. J. kleinfelder, "Splash 2- FPGAs in a Custom Computing Machine,".: IEEE Computer Society, 1996.

[68] *P. M. S. Athanas, H.F.*, "Processor reconfiguration through instruction-set metamorphosis," in *Computer*, vol. 26, 1993, pp. 11-18.

[69] E. K. Acosta, J. V. M. Bove, J. A. Watlington, and R. A. Yu, "Reconfigurable

Processor for a Data-Flow Video Processing System," presented at *FPGAs for Fast Board Development and Reconfigurable Computing*, 1995.

[70] E. K. Acosta, "A Programmable Processor for the Cheops Image Processing System," in *EECS*. Cambridge, MA: Massachusetts Institute of Technology, 1995.

[71] R. D. Isaac, "The future of CMOS technology," *IBM Journal of Research and Development*, vol. 44, pp. 369-378, 2000.

[72] *XC3000 Series Field Programmable Gate Arrays*: Xilinx Corporation, 1998.

[73] *XC4000E and XC4000X Series (EX/XL) Field Programmable Gate Arrays*, 1.2 ed: Xilinx Corporation, 1999.

[74] S. Brown and J. Rose, "FPGA and CPLD architectures: A tutorial," *IEEE Design & Test of Computers*, vol. 13, pp. 42-57, 1996.