ABSTRACT

DESIGN AND RESOURCE MANAGEMENT OF RECONFIGURABLE MULTIPROCESSORS FOR DATA-PARALLEL APPLICATIONS

by
Xiaofang Wang

FPGA (Field-Programmable Gate Array)-based custom reconfigurable computing machines have established themselves as low-cost and low-risk alternatives to ASIC (Application-Specific Integrated Circuit) implementations and general-purpose microprocessors in accelerating a wide range of computation-intensive applications. Most often they are *Application-Specific Programmable Circuits* (ASPCs), which are developer programmable instead of user programmable. The major disadvantages of ASPCs are minimal programmability, and significant time and energy overheads caused by required hardware reconfiguration when the problem size outnumbers the available reconfigurable resources; these problems are expected to become more serious with increases in the FPGA chip size. On the other hand, dominant high-performance computing systems, such as PC clusters and SMPs (Symmetric Multiprocessors), suffer from high communication latencies and/or scalability problems.

This research introduces low-cost, user-programmable and reconfigurable *MultiProcessor-on-a-Programmable-Chip* (*MPoPC*) systems for high-performance, low-cost computing. It also proposes a relevant resource management framework that deals with performance, power consumption and energy issues. These semi-customized systems reduce significantly runtime device reconfiguration by employing user-programmable processing elements that are reusable for different tasks in large, complex applications. For the sake of illustration, two different types of MPoPCs with hardware *FPUs* (*floating-point units*) are designed and implemented for credible performance evaluation and modeling: the coarse-grain MIMD (Multiple-Instruction,

Multiple-Data) CG-MPoPC machine based on a processor IP (Intellectual Property) core and the mixed-mode (MIMD, SIMD or M-SIMD) variant-grain HERA (*HEterogeneous Reconfigurable Architecture*) machine. In addition to alleviating the above difficulties, MPoPCs can offer several performance and energy advantages to our data-parallel applications when compared to ASPCs; they are simpler and more scalable, and have less verification time and cost. Various common computation-intensive benchmark algorithms, such as matrix-matrix multiplication (MMM) and LU factorization, are studied and their parallel solutions are shown for the two MPoPCs. The performance is evaluated with large sparse real-world matrices primarily from power engineering. We expect even further performance gains on MPoPCs in the near future by employing ever improving FPGAs. The innovative nature of this work has the potential to guide research in this arising field of high-performance, low-cost reconfigurable computing.

The largest advantage of reconfigurable logic lies in its large degree of hardware customization and reconfiguration which allows reusing the resources to match the computation and communication needs of applications. Therefore, a major effort in the presented design methodology for mixed-mode MPoPCs, like HERA, is devoted to effective resource management. A two-phase approach is applied. A mixed-mode *weighted Task Flow Graph* (*w*-TFG) is first constructed for any given application, where tasks are classified according to their most appropriate computing mode (e.g., SIMD or MIMD). At compile time, an architecture is customized and synthesized for the TFG using an Integer Linear Programming (ILP) formulation and a *parameterized hardware component library*. Various run-time scheduling schemes with different performance-energy objectives are proposed. A system-level energy model for HERA, which is based on low-level implementation data and run-time statistics, is proposed to guide performance-energy trade-off decisions. A parallel power flow analysis technique based on Newton's method is proposed and employed to verify the methodology.

DESIGN AND RESOURCE MANAGEMENT OF RECONFIGURABLE MULTIPROCESSORS FOR DATA-PARALLEL APPLICATIONS

by
Xiaofang Wang

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Engineering

Department of Electrical and Computer Engineering, NJIT

January 2006

APPROVAL PAGE

DESIGN AND RESOURCE MANAGEMENT OF RECONFIGURABLE MULTIPROCESSORS FOR DATA-PARALLEL APPLICATIONS

Xiaofang Wang

Dr. Sotirios G. Ziavras, Dissertation Advisor and Committee Chair Professor of Electrical and Computer Engineering, NJIT	Date
Dr. Alexandros V. Gerbessiotis, Committee Member Associate Professor of Computer Science, NJIT	Date
Dr. Jie Hu, Committee Member Assistant Professor of Electrical and Computer Engineering, NJIT	Date
Dr. Durgamadhab Misra, Committee Member Professor of Electrical and Computer Engineering, NJIT	Date
Dr. Roberto Rojas-Cessa, Committee Member Assistant Professor of Electrical and Computer Engineering, NJIT	Date

BIOGRAPHICAL SKETCH

Author: Xiaofang (Maggie) Wang

Degree: Doctor of Philosophy

Major: Computer Engineering

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, January 2006.

- Master of Science in Electrical Engineering,
  Beijing University of Technology (formerly Beijing Polytechnic University),
  Beijing, P. R. China, 1994.

- Bachelor of Science in Microelectronics,
  Nankai University, Tianjin, P. R. China, 1991.

Publications:

X. Wang and S. G. Ziavras,
"Exploiting Mixed-Mode Parallelism for Matrix Operations on the HERA Architecture through Reconfiguration,"
IEE Proceedings, Computers and Digital Techniques, accepted in 2005.

X. Wang and S. G. Ziavras,
"A Multiprocessor-on-a-Programmable-Chip Reconfigurable System for Matrix Operations with Power-Grid Case Studies,"
International Journal of Computational Science and Engineering, Special Issue on Parallel and Distributed Scientific and Engineering Computing, accepted in 2005.

X. Wang and S. G. Ziavras,
"Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines,"
Concurrency and Computation: Practice and Experience, Vol. 16, No. 4, pp. 319-343, 2004.

X. Wang, S. G. Ziavras, and J. Hu,
"Energy-Performance Optimization through Resource Management in Reconfigurable Mixed-Mode Single-Chip Multiprocessors," submitted in December 2005.

X. Wang and S. G. Ziavras,
"A Framework for Dynamic Resource Management and Scheduling on Reconfigurable Mixed-Mode Multiprocessors,"
IEEE International Conference on Field-Programmable Technology (FPT'05), Singapore, Dec. 11-14, 2005.

X. Wang and S. G. Ziavras,
"Adaptive Scheduling of Array-Intensive Applications on Mixed-Mode Reconfigurable Multiprocessors,"
IEEE 39th Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California, Oct. 30-Nov.2, 2005.

S. G. Ziavras, X. Wang, and M. Z. Hasan,
"Intra- and Inter-FPGA Programmable Multiprocessor Designs with Emphasis on Large-Scale Matrix Operations,"
Workshop on Architecture Research using FPGA Platforms (in conjunction with the *11th International Symposium on High-Performance Computer Architecture*), February 2005.

X. Wang and S. G. Ziavras,
"Mixed-Mode Scheduling for Parallel LU Factorization of Sparse Matrices on the Reconfigurable HERA Computer,"
International Conference on Advances in Computer Science and Technology (ACST 2004), St. Thomas, U.S. Virgin Islands, November 2004.

X. Wang and S. G. Ziavras,
"HERA: A Reconfigurable and Mixed-Mode Parallel Computing Engine on Platform FPGAs," *The 16th International Conference on Parallel and Distributed Computing and Systems* (PDCS 2004), MIT, Cambridge, MA, November 9-11, 2004.

X. Wang and S. G. Ziavras,
"A Configurable Multiprocessor and Dynamic Load Balancing for Parallel LU Factorization,"
The 5th Workshop on Parallel and Distributed Scientific and Engineering (Proc. of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS2004)), Santa Fe, New Mexico, April 2004.

X. Wang and S. G. Ziavras,
"Performance Optimization of an FPGA-Based Configurable Multiprocessor for Matrix Applications,"
IEEE International Conference on Field-Programmable Technology (FPT'03), pp. 303-306, Dec. 2003.

X. Wang and S. G. Ziavras,
"Parallel Direct Solution of Linear Equations on FPGA-Based Machines,"
The 11th Workshop on Parallel and Distributed Real-Time Systems (Proc. of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS2003)), Nice, France, pp. 113-120, April 22-26, 2003.

X. Wang, S. G. Ziavras, and J. Savir,
"Efficient LU Factorization on FPGA-Based Machines,"
The 7th International Multi-Conference on Power and Energy Systems, Palm Springs, CA, pp. 459-464, February 2003.

To my beloved parents
and husband.

ACKNOWLEDGMENT

As an international student, I have been very fortunate to get help from many people in many aspects during this important and rewarding phase of my professional life. I thank all of them.

In particular, I would like to express my deepest and sincere gratitude to my dissertation advisor, Dr. Sotirios G. Ziavras, for many things: his constant encouragement strengthened me in delving into this research and finding solutions when I was confused; his great insight and perspectives inspired me toward completing successfully this thesis; he spent countless hours on advisement, discussion, and revising our papers; he was always there when I had a problem to discuss; ... What I have learned from him will benefit me in my lifetime as a professional. Special thanks are also given to the other members of my dissertation committee, Dr. Alexandros V. Gerbessiotis, Dr. Jie Hu, Dr. Durga Misra and Dr. Roberto Rojas-Cessa, for their suggestions, comments and beneficial discussions.

I am grateful to several funding sources that supported my Ph.D work. They involve a *Teaching Assistant* position in the ECE department at NJIT, multi-year research grants for the U. S. Dept. of Energy *PowerGrid* project, and the ECE *Phonetel* and *Hashimoto Fellowships*.

The valuable and prompt support from Altera and Xilinx engineers during my implementation of the multiprocessors is highly appreciated.

My father always encouraged me to challenge my talents and to pursue the Ph.D. degree. Finally, I am very happy in fulfilling his hope and making him happy.

TABLE OF CONTENTS

Chapter	Page
1 INTRODUCTION	1
1.1 High-Performance Applications	1
1.2 Current High-Performance Computing Systems	1
1.2.1 Proprietary Supercomputers	2
1.2.2 Shared-Memory Multiprocessors	3
1.2.3 Message-Passing Multicomputers	5
1.2.4 Distributed Shared-Memory Multicomputers	5
1.2.5 Cluster-Based Computers	5
1.2.6 Grid Computing	6
1.3 Reconfigurable Computing	7
1.3.1 Current Trends in Reconfigurable Systems	8
1.3.2 New Opportunities	11
1.4 Motivations	12
1.5 Objectives and Contributions	14
1.6 Dissertation Organization	17
2 RECONFIGURABLE COMPUTING	19
2.1 Field-Programmable Gate Arrays	19
2.2 Recent Advances in FPGAs	21
2.3 Examples of Coarse-Grain Reconfigurable Architectures	23
2.4 Design Methodology for Reconfigurable Machines	24

TABLE OF CONTENTS
(Continued)

Chapter	Page
3 MULTIPROCESSORS ON A PROGRAMMABLE CHIP	28
3.1 A Coarse-Grained IP-based MPOPC (CG-MPOPC)	30
3.1.1 Multiprocessor Architecture	30
3.1.2 Processing Element	31
3.1.3 Memory Hierarchy Design	32
3.1.4 Implementation Results	34
3.2 HERA: A Reconfigurable Mixed-Mode Parallel Computer	34
3.2.1 System Organization	35
3.2.2 PE Architecture	37
3.2.3 Memory Configuration	39
3.2.4 Instruction Set	40
3.2.5 Implementation Results	43
4 APPLICATION STUDY	44
4.1 Generalized Cannon's Matrix-Matrix Multiplication Algorithm	44
4.1.1 Data Partitioning and Mapping	44
4.1.2 Dynamic Mixed-Mode Scheduling on HERA	45
4.2 Parallel LU Factorization of Large Sparse Matrices	47
4.2.1 Overview of LU Factorization	47
4.2.2 Near-Optimal Ordering Selection	50
4.2.3 Minimum Degree Ordering	53

TABLE OF CONTENTS
(Continued)

Chapter **Page**

4.2.4 Dynamic Task Scheduling..... 53

4.2.4.1 Task Definition..... 54

4.2.4.2 State Information..... 55

4.2.4.3 Dynamic Scheduling Procedure..... 56

4.2.4.4 Theoretical Performance Analysis..... 58

4.2.5 Dynamic Mixed-Mode Scheduling on HERA..... 63

4.3 Parallel Direct Solution of Sparse Linear Equations..... 65

4.4 Parallel Solution of Newton's Power Flow Equations..... 67

4.4.1 Newton's Solution to the Power Flow Problem..... 69

4.4.2 Parallel LU Factorization of Jacobian Matrices..... 72

4.4.3 Parallel Solution of Newton's Power Flow Equations..... 77

4.4.4 Relevance to Other Work..... 78

5 PERFORMANCE RESULTS AND ANALYSIS..... 80

5.1 Mixed-Mode Scheduling of MMM on HERA..... 80

5.2 Parallel LU Factorization of Sparse Matrices on CG-MPoPC..... 82

5.2.1 MPoPC Customization and Configuration..... 83

5.2.2 Experiments and Analysis..... 86

5.3 Parallel LU Factorization of Sparse Matrices on HERA..... 93

5.4 Parallel Power Flow Analysis on CG-MPoPC..... 97

6 SYSTEM-LEVEL ENERGY MODELING..... 100

TABLE OF CONTENTS
(Continued)

Chapter	Page
6.1 Related Work	101
6.2 Power Characterization of Library Function Units	103
6.3 HERA System-Level Energy Model	109
7 A FRAMEWORK FOR RESOURCE MANAGEMENT ON MPOPCs	113
7.1 Related Work	114
7.2 Problem Definition and Objectives	115
7.3 Framework Overview	116
7.4 Application Model	117
7.4.1 Task Flow Graph	117
7.4.2 IF-THEN-ELSE	119
7.4.3 Loops	120
7.5 Architecture Synthesis and Reconfiguration	122
7.5.1 Parameterized Hardware Component Library	123
7.5.2 Application-Specific System Synthesis	124
7.6 Dynamic Resource Scheduling for Performance-Energy Optimization	131
7.6.1 Related Work	131
7.6.2 Loop Partitioning.,	134
7.6.3 PE Search	135
7.6.4 Dynamic Resource Scheduling Schemes	137
7.6.4.1 Optimize the Performance without Energy Constraints	139

TABLE OF CONTENTS
(Continued)

Chapter **Page**

7.6.4.2 Optimize the Performance with an Energy Constraint..... 141

7.6.4.3 Optimize the Energy Cost under an Allowable Performance Loss 142

7.7 Experimental Results..... 143

7.7.1 Singular Value Decomposition..... 143

7.7.2 Parallel Power Flow Analysis 147

8 CONCLUSIONS AND FUTURE WORK..... 153

8.1 Conclusions..... 153

8.2 Future Work..... 156

BIBLIOGRAPHY..... 158

LIST OF TABLES

Table		Page
2.1	Comparison of Previous Coarse-Grain Reconfigurable Systems	25
3.1	The Instruction Set of HERA	41
4.1	Sparsity of Benchmark Power Matrices	48
4.2	Sparsity of the Benchmark Matrices for Power Flow Analysis	72
4.3	The Sizes of the Blocks in the Jacobian Matrix	74
5.1	HERA Execution Times for Irregular Matrices under Different Execution Modes	82
5.2	Characteristics of the Test Matrices ordered into the DBBD Form	88
5.3	Execution times (seconds) for the benchmark matrices on the two MPoPCs	90
5.4	IEEE Single-Precision Floating-Point Performance and Resource Utilization	94
5.5	Latency Comparison also Involving a DSP Processor	97
5.6	Optimal Partitioning of the Y_{bus} Matrices of the Benchmark Systems	98
5.7	Execution Times (msec) to Solve the Linear Equations for the Benchmark Systems on our Configurable Multiprocessor	99
5.8	Execution Times (sec) for Newton's Power Flow Equations with Seven Processors	99
6.1	Resource Usage (in slices) of Floating-Point FUs on XC2V6000-5	105
6.2	Total power consumption (mW) of the IEEE-754 Single- and Double-Precision FP FUs	105
7.1	Major Parameters of an FP FU in PHCL	124
7.2	SVD Task Information	145
7.3	Task Information of the Parallel DBBD Power Flow Algorithm	150
7.4	Optimal Partitioning of the Y_{bus} Matrices for the Benchmark Systems	151

LIST OF TABLES
(Continued)

Table		Page
7.5	The Parallelism Profile during the Execution.....	151
7.6	Execution Times for the Benchmark Matrices.....	151
7.7	Comparison between the Modeled and XPower-Reported Energy Consumption..	152
7.8	Performance-Energy Optimization for the 7917-Bus System.....	152

LIST OF FIGURES

Figure		Page
1.1	Temporal computing vs. spatial computing	8
1.2	Conventional methodology in reconfigurable computing	10
2.1	A CLB in Virtex II FPGAs	21
2.2	Slice configuration in Virtex FPGAs	21
2.3	Virtex 4 FPGA [Xilinx]	22
2.4	Conventional development flow for FPGA-based systems	24
3.1	The CG-MPoPC architecture	31
3.2	CG-MPoPC memory configuration	33
3.3	HERA system architecture	36
3.4	A HERA PE	38
3.5	HERA memory interface	40
3.6	HERA general instruction format	41
4.1	A partitioning example for matrices A and B ($q = 3, p1 = 2, p2 = 3, p3 = 3$)	45
4.2	Sparse DBBD matrix format	49
4.3	DBBD ordering for a matrix of size 10279 x 10279	52
4.4	The non-zero elements in the 10279-Y_{bus} and the corresponding DBBD matrices	52
4.5	Typical PE mode assignment for large DBBD matrices	65
4.6	Sparse DBBD Y_{bus} matrix	72
4.7	The Jacobian matrix produced from the DBBD Y_{bus} matrix	74
4.8	Nonzero elements for the 7917-bus system	75
5.1	Performance comparison of MMM on HERA and two Dell PCs	81

LIST OF FIGURES
(Continued)

Figure		Page
5.2	HERA speedup of parallel over uni-PE execution	82
5.3	PE and SC connectivity	84
5.4	MPoPC configurations for the tasks in DBBD-based parallel LU factorization	85
5.5	Interconnecting on-chip data memories for the MPoPC configurations of Fig.5.4	86
5.6	Impact of network partitioning on the execution time of parallel LU factorization for a DBBD matrix of 2582 x 2582	89
5.7	Speedup comparison of the run-time and static scheduling policies on the customized MPoPC	90
5.8	Speedup (over the uni-processor) of the static and dynamic scheduling policies for the 10279-Y_{bus} matrix. No hardware FPUs	91
5.9	Percentage of time needed to factor the last block in the 10279-Y_{bus} matrix	92
5.10	Comparing the predicted and real performance for the 7917-matrix	92
5.11	Execution time for the 10279-Y_{bus} matrix affected by pre-fetching	93
5.12	Execution times on HERA under the SIMD, MIMD and mixed modes (HERA system frequency: 125MHz)	95
6.1	Dynamic power consumption (per slice) of the single- and double-precision FP FUs	105
6.2	Impact of the average input activity rate on the core dynamic power consumption	107
6.3	Relationship between the core dynamic power consumption and the clock frequency	109
7.1	Design methodology overview/flowchart	117
7.2	A typical task flow graph	118
7.3	SIMD, MIMD and mixed-mode mapping of conditional blocks	120

LIST OF FIGURES
(Continued)

Figure		Page
7.4	Special examples of FOR loops	122
7.5	An example of function selection for PEs	131
7.6	Cross-iteration dependence	135
7.7	PE search path	136
7.8	Execution times with and without partial runtime reconfiguration (RTR)	146
7.9	Normalized execution times for our strategy and naive dynamic scheduling	147

CHAPTER 1

INTRODUCTION

1.1 High-Performance Applications

Many large-scale scientific and engineering problems appearing in areas such as bioinformatics, power engineering, astrophysics, high-energy physics and chemistry, structural analysis, circuit simulation, traffic simulation, and fluid dynamics can be formulated as the recurring solution of a system of equations [Bailey, 1998; Fox, et. al., 1988]. The corresponding matrix-based algorithmic solutions are often computation intensive and present major challenges to current computing systems. For example, the complexity of two common algorithmic cores in the above applications, namely LU factorization and matrix multiplication, require $O(N^3)$ time for an N x N is the matrix. Besides these classic high-end applications, many algorithms in newly emerging areas, such as wireless communications, data-intensive internet applications also present greedy demands for computing power in order to provide real-time services. Parallel computing has been recognized as an effective and viable solution to accelerate such problems and significant research has been ongoing for decades. New exciting frontiers in bioinformatics in the past few years, such as the sequencing of the human genome, rely heavily on parallel computers [Grama, et al., 2003].

1.2 Current High-Performance Computing Systems

After tremendous investment and decades of experimentation, clusters of Cray-like vector supercomputers, distributed shared-memory multicomputers employing crossbar

or multistage interconnection networks, and clusters of scalar uni- and multi-processor systems dominate the high-performance computing field [Bell, et al., 2002; Simon, 2003; Kuck, 1996]. Steady advances in related technologies provide the possibility and flexibility to mix features found in these systems, so many hybrid computing systems have been developed. Our taxonomy of parallel architectures is based on the programmer's, or more specifically, the compiler's view.

1.2.1 Proprietary Supercomputers

Supercomputers typically follow custom designs and fall into one of these computing architectures: vector supercomputers, and shared-memory (SM) SIMD, distributed-memory (DM) SIMD, SM-MIMD and DM-MIMD machines [Hwang, 2003]. The performance of these machines largely depends on their architecture and proprietary compilers. Traditional supercomputers have accomplished a great deal of success in solving computation-intensive problems and represent the top end of stand alone computing systems in terms of high computing power, high bandwidth and low latency interconnects, very fast memories and high I/O rates. Representative supercomputers, some of them still in use, include the Earth Simulator from NEC, the T3D and T3E from Cray, the SX-4/5 from NEC, the Challenge XL and Origin 2000 from Silicon Graphics, and the CM-5 from Thinking Machines Corporation. The performance of the best performing 500 (TOP500 list) supercomputers in the world can be found at *http://www.top500.org*, where the term "supercomputer" is used in a broader scope. Most state-of-the-art custom supercomputers are vector based or contain a cluster of vector components and off-the-shelf RISC processors, such as the Opteron, PowerPC or

PA-RISC. The traditional supercomputer industry has languished in recent years [Bell, et al., 2002; Vaughan-Nichols, et al., 2004]; besides reduced government and industry spending on supercomputer technology, the high price, the long design and development cycles, the difficulty of programming them, the high cost of maintaining them and their huge power consumption, limit the application of supercomputers to many diverse fields. Although PC clusters have demonstrated increased performance (as shown in the TOP500 list), their long interconnect latencies still require custom supercomputers in numerous capacity and mission-critical problems or problems characterized by fine-grain parallelism. Some areas often requiring custom supercomputers are weather forecasting, climate research, molecular modeling (computing the structures and properties of chemical compounds, biological macromolecules, polymers, and crystals), physical simulations (such as simulation of airplanes in wind tunnels, simulation of the detonation of nuclear weapons, and research into nuclear fusion), and cryptanalysis.

1.2.2 Shared-Memory Multiprocessors

The most common architecture employed in current shared-memory multiprocessors is non-uniform memory access (NUMA) symmetrical multiprocessing (SMP) [Tosic, 2004]. Multiprocessors used to be present in high-end mainframes and servers and they appear now in many kinds of systems, including high-end PCs and workstations. Examples include the Sun Enterprise 6000, the SGI Challenge and the Intel SystemPro. The SMP systems are usually small due to their nature of shared memory.

Recent advances in integrated circuit technology have fueled another opportunity: multiprocessor-on-a-chip. Single-chip multiprocessors based on fixed logic have recently emerged as the result of major hurdles in superscalar microprocessor design [Ronen, et al., 2001]. Two major categories of multiprocessors have attracted intensive interest in the academic and industrial settings. The first category utilizes advanced superscalar cores with a shared memory [Krashinsky, et al., 2004; Hammond, et al., 2000; Barroso, et al., 2000] whereas the other integrates a large number of simple, pipelined cores, like MPSoCs (MultiProcessor-Systems-on-a-Chip) [Power4; Hofstee, et al., 2005; Wolf, 2004; Stolberg, et al., 2005; Henkel, et al., 2004; Jerraya, et al., 2004]. Recent research category in the first group include, among others, Hydra [Olukotun, et al., 1996], SCMP [Baker, et al., 2002] and SCALE [Krashinsky, et al., 2004]. Hydra is designed around complex superscalar processors. SCALE combines vector processing and multithreading. SCMP is a multiprocessor organized in a 2-D mesh without global communication channels. Current MPSoC implementations have been optimized for real-time applications in networking, multimedia and communications using heterogeneous processors and custom function units [Jerraya, et al., 2004]. For chip multiprocessors based on custom logic, a high volume is required to amortize the high development and NRE (nonrecurring engineering) costs, especially for deep sub-micron designs. Also, the ever-shortening product cycles and the high design complexity of such solutions limit their viability [Bergamaschi, et al., 2001]. We have also seen some reconfigurable single-chip multiprocessors based on custom reconfigurable logic instead of commercial FPGAs, e.g., PACT XPP [Becker, et al., 2003].

1.2.3 Message-Passing Multicomputers

Message-passing multipcomputers are normally implemented with a distributed-memory architecture. They consist of multiple computers, often called *nodes*, interconnected by a uniform point-to-point network [Hwang, 2003]. Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals. The boundary between multiprocessors and multicomputers has become blurred in recent years. Examples falling into this category include the Intel Paragon and iPSC/2, Transputer-based systems and nCube machines. Multicomputer design and implementation have been declining since the mid-1990s with the increasing popularity of cluster-based systems and distributed shared-memory systems.

1.2.4 Distributed Shared-Memory Multicomputers

These are systems normally implemented with a point-to-point interconnection network but there is often both hardware and software support to implement shared memory [Hwang, 1993].

1.2.5 Cluster-Based Computers

A computer cluster is viewed as a single computing system comprising interconnected stand-alone computers that communicate with one another either via message passing or shared memory [Bell, et al., 2002]. Taking advantage of exponential advances in commercial off-the-shelf (COST) components since the mid-1990s, such as general-purpose microprocessors and Ethernet technologies, clusters of open architecture systems quickly entered the mainstream of the high-performance computing (HPC) world as the specialist supercomputer market shrank. They have much lower cost than

the latter and are also rather scalable in hardware. More than half of the TOP500 supercomputers released in November 2004 are labeled as clusters [TOP500], making them the most common architecture on the list. They bring the benefits of parallel processing at reduced cost to a broader scope, and provide an easy-to-use and accessible parallel processing alternative to the majority of high-performance applications. The ease of implementing standard programming models on them is a tremendous advantage. However, although it is easy to scale up and upgrade the hardware configuration of clusters, the performance of many parallel algorithms does not scale well on these machines primarily due to high communication latencies [Lan, et al., 2003]. They are more effective for loosely-coupled tasks lacking frequent communications [Vaughan-Nichols, et al., 2004].

1.2.6 Grid Computing

While computer clusters are often groups of dedicated homogeneous computers administrated as a single system, grid systems focus on integrating, virtualizing and coordinating computing resources and services within distributed heterogeneous systems that are in separate administrative domains [OGSA-WG]. Grids share advantages with cluster-based systems, such as low-cost and stand-alone nodes easy to maintain. They also share exaggerated disadvantages, such as very high communication costs. TeraGrid [Reed, 2003] is the largest research grid in this category.

1.3 Reconfigurable Computing

At the physical level, two primary approaches have been employed in the implementation of applications: programmable microprocessors and customized hardware utilizing ASIC chips. Programmable microprocessors have a general-purpose, fixed architecture that implements applications temporally via atomic operations dictated by machine instructions (*temporal computing*). They can also support very limited spatial execution of operations with multiple functional units. However, the price of the programming flexibility is rather low performance, which can be far below that of an ASIC design. Also, microprocessors consume more power than ASICs. In contrast, ASICs are designed and manufactured explicitly for specific applications by spatially decomposing operations that can be implemented directly by dedicated functional units like adders or multipliers (*spatial computing*) and modification requires re-design and re-fabrication of the chip, which is an expensive process, especially with multi-million-gate chips in sub-micron processes. ASICs are designed to perform a specific algorithm quickly and efficiently, but cannot be altered after fabrication. Figure 1.1 illustrates the two computing approaches for the execution of a small loop.

Reconfigurable computing [Compton, et al., 2002] sits between the extremes of general-purpose microprocessors and specialized ASICs, and allows a high degree of both spatial and temporal execution of the operations. A reconfigurable system usually employs reconfigurable devices, such as FPGAs, and works closely with one or more general-purpose processors to accelerate computation-intensive or highly parallel applications. The reconfigurable logic can be adapted (reprogrammed) for different application. Hence, reconfigurable systems are flexible due to field programmability

after fabrication and are much less expensive than ASIC designs; but they are less efficient in terms of power and resource consumption, and are usually slower than the latter. On the other hand, they can offer much better performance due to their (semi-) customization for a wide range of applications as compared to general-purpose microprocessors. However, their adaptation requires hardware expertise.

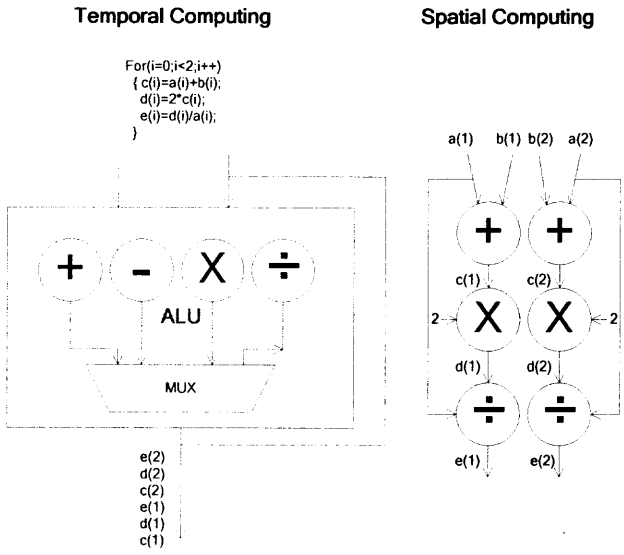


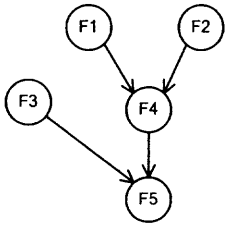
Figure 1.1 Temporal computing vs. spatial computing.

1.3.1 Current Trends in Reconfigurable Systems

FPGA-based computing machines have recently demonstrated considerable performance gains over general-purpose microprocessors for many computation-intensive applications [Compton, et al., 2002; Bondalapati, et al., 2002]. Most of them take advantage of the fine-grain architecture in earlier FPGAs and are fully customized for a specific class of applications, like ASIC designs, but with much lower costs and more flexibility than ASIC designs. Most machines target bit-level multimedia and DSP applications where floating-point operations are not often necessary. Because *floating-*

point units (FPUs) consume a very large portion of the resources in earlier FPGAs, very few such machines support floating-point arithmetic. Due to the limited resources in prior FPGAs, the fine-grain functional units in such machines most often are not program accessible and their overall processing capabilities are rather limited. These FPGAs are developer-, rather than user-programmable. A small change in the algorithm requires full reconfiguration of the hardware, which takes significant time. Moreover, full hardware reconfiguration is required when the problem size exceeds the available resources on the FPGAs, which is a major overhead in terms of time and energy during the application execution. Each reconfiguration consumes tens to hundreds of milliseconds. Figure 1.2 shows the general idea of such approaches. Figure 1.2 (a) is the application data flow graph, and the resource requirements and execution times of the tasks are shown in Figure 1.2 (b). Task mapping and scheduling on the FPGA is shown in Figure 1.2 (c). Let the configuration time of the FPGA be 1 unit of time. It is clear that the required hardware configuration time is rather significant compared to the computation time. Also, many resources are wasted during execution, as the figure shows. In an application shown for the Dynamic Instruction Set Computer (DISC), the configuration overhead contributes more than 25% of the total execution time [Wirthlin, et al., 1996].

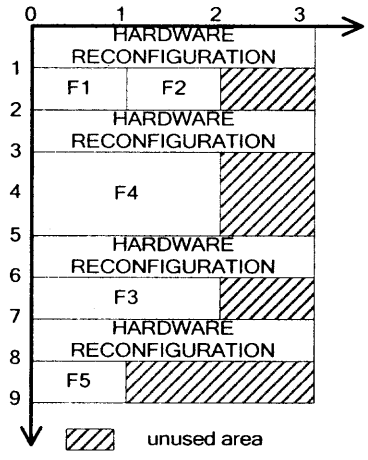
In contrast to the ever increasing speed of logic resources, the configuration overhead for SRAM-based FPGAs becomes more serious with increases in the chip size [Pan, et al., 2004] since the size of the configuration data is proportional to the total number of on-chip resources. For example, the configuration time of the device we use,



(a)

Task	Latency	Area
F1	1	1
F2	1	1
F3	1	2
F4	2	2
F5	1	1

(b)



Assume the total number of resources in the FPGA is 3.

(c)

Figure 1.2 Conventional methodology in reconfigurable computing.

the XC2V6000-5, is at least 50 msec [Xilinx Virtex II]. In comparison, we can multiply two matrices of size 1000 x 1000 in about 5 msec and perform the LU factorization a matrix of size 1000 x 1000 in about 50 msec on the same FPGA chip [Wang, et al., 2005]. The power required to reconfigure the device is another serious issue that cannot be ignored. The aforementioned device requires at least 29.7 W (800 mA * 3.3 V + 100 mA * 1.65 V + 100 mA * 1.65 V) [Xilinx Virtex II] power consumption during each configuration, which results in a total 1485 mJ of energy consumption. Research efforts trying to alleviate both problems include reducing the number of reconfigurations [Ghiasi, et al., 2004], increasing the sharing of function units [Cardoso, 2003], compression of the configuration bits [Pan, et al., 2004; Li, et al., 2001] and alternative architectures, such as multi-context FPGAs [DeHon, 1997]. The reconfiguration of such devices is carried out by switching from one configuration (context) to another one stored in the device by replicating the configuration memory; this is known as *context switching* [Scalera,et al., 1998]. Additional storage resources are needed in such devices

to store multiple configurations and intermediate results between reconfigurations. This causes serious power issues and these choices do not appeal to most FPGA vendors.

Further discussion about the conventional design methodology and its disadvantages is presented in detail in Chapter 2. However, we expect this approach to continue playing a major role in application areas where field programmability by the user is not required or is needed rarely and the problem size is small enough and full reconfiguration is not required.

1.3.2 New Opportunities

With the recent achievement of multi-million-gate platform FPGAs to contain richer embedded feature sets, such as plenty of on-chip memory, DSP blocks and embedded microprocessor IP cores, FPGA-based reconfigurable computing is going through a revolution. New FPGAs employ coarse-grain architectures to facilitate more powerful coarse-grain datapaths. The peak floating-point performance of FPGAs has outnumbered in the last two years that of modern microprocessors and is growing much faster than the latter [Underwood, 2004]. Recent research efforts in the design and implementation of FPUs [Zhuo, et al., 2004; Liang, et al., 2003] and computation-intensive algorithms on state-of-the-art FPGAs provide evidence to this effect. However, they follow the traditional approach where the circuitry is only applicable to the specific algorithms invented by the developer and the studied problems were of very small size. It is now viable for FPGAs to accommodate some high-performance applications. Even supercomputer manufacturers have recently incorporated FPGAs in their designs. For example, Cray incorporates six Xilinx Virtex 4 FPGAs per chassis in its XD1™

supercomputers that can be used as coprocessors to accelerate computation-intensive applications. The advent of soft IP configurable processors from FPGA vendors, such as Microblaze from Xilinx and Nios from Altera, have inspired some multiprocessor implementations on FPGAs [Hung, et al., 2005; Salminen, et al., 2005; Hoare, et al., 2004; Ravindran, et al. 2005]. However, we have not seen FPGA-based single-chip multiprocessors that incorporate hardware FPUs.

1.4 Motivations

From the above discussion we can see that PC-based cluster systems and SMP multiprocessors are the dominant high-performance platforms for the majority of computation-intensive applications. Nevertheless, their shared-memory nature limits the size of SMP systems and the high communication latencies in cluster systems make them more effective for loosely-coupled tasks lacking frequent communications. Both of them are based on general-purpose COTS components and are only effective on certain classes of applications. Due to the different characteristics of general-purpose and high-performance computing, we cannot rely solely on COTS components to improve the latter. Moreover, conventional (micro)architectures are fast approaching a performance limit due to the limited ILP (Instruction Level Parallelism) in real programs [Ronen, et al., 2001]; their large power dissipation is a major problem as well. Also, wire delays decrease much slower than transistor switching times for deep sub-micron processes. As a result, a major shift from ILP to TLP (Thread Level Parallelism) is present in the industry and research communities. To this extent, AMD, Intel, Sun, and IBM, among others, have recently introduced multicore chips.

The author is among the very few who observed very early that FPGAs provide a new opportunity to the high-performance computing field. State-of-the-art FPGAs have made it feasible to build high performance computing systems at affordable costs with hardware support for floating-point operations. High-performance applications often involve complex matrix-based algorithms where software programmability and standard FP representation are indispensable. Scalability and portability are also essential to performance due to the variant size of matrices and the ever changing parameters of various applications. These systems can leverage system level concepts from high-performance computing and can dynamically tune their architecture to fit the applications. They are also accessible to applications due to their low cost. However, this new approach requires extensive expertise in computer architecture, parallel processing, and digital and FPGA-based designs in order to yield high performance. The majority of the FPGA community still follows the conventional approach of designing and implementing acceleration circuitry for specific algorithms, as discussed in Section 1.3. To the best of our knowledge, we have not seen yet major research efforts in the new MPoPC direction and very few FPGA-based computing systems incorporating FPUs have been published.

The programming of reconfigurable systems for high performance can be quite challenging as it essentially involves hardware design. Although several groups have recognized that the success of such systems will highly depend on high-level design tools to efficiently map applications onto the hardware, they focus their efforts on developing more general, software-oriented approaches that resemble traditional compilers for general-purpose microprocessors; they assume simplified and regular

models for reconfigurable systems, or no specific architecture at all. Due to their difficulty in implementing hardware, most of the published results are based on simulation only. This is a major drawback as it is indeed important to implement such systems in order to evaluate the performance accurately. Reconfigurable systems are more diverse than conventional high-performance computing systems due to their reconfiguration flexibility and the eventual (semi-)customization of hardware to run application code. It is hence very important for the mapping tools to be hardware-oriented and take into account the specific idiosyncrasies, features and constraints of the target systems in order to achieve the high performance they are designed for; this is often accomplished by fully utilizing the hardware resources.

1.5 Objectives and Contributions

The first objective of this research is to propose a design methodology for high-performance, low-cost reconfigurable systems targeting large data-parallel applications [Hills, et. al., 1986] and utilizing new-generation FPGAs. The focus here is on high-performance and reconfigurable MPoPCs implemented with state-of-the-art platform FPGAs. A major contribution is the pioneering nature of reconfigurable MPoPCs, and the system-oriented approach to design and implement them for data-parallel applications. No related major efforts have been published.

Two different types of MPoPCs with hardware FPUs were designed and implemented to provide a base for further study: (a) a coarse-grain MPoPC (CG-MPoPC) based on a configurable IP processor core from Altera (i.e. Nios) that was implemented on the Altera SoPC FPGA board, and (b) the HERA mixed-mode variant-

grain machine that was implemented on Xilinx FPGAs. CG-MPoPC is designed to run in the MIMD mode while HERA can be reconfigured at runtime to support a variety of independent or cooperating computing modes, such as SIMD, MIMD and M-SIMD. Therefore, HERA can potentially match better in the time spectrum all subtask characteristics of a given application. The PEs in both systems are equipped with large data and instruction on-chip memories. Platform FPGAs also provide substantial flexibility to integrate many features found in conventional high-performance computing systems. In contrast to previous FPGA-based custom computing machines, these systems are also user-programmable by general-purpose instructions. To save on reconfiguration time, full hardware reconfiguration during execution is eliminated by employing user-programmable PEs. Parallel solutions for two computation-intensive benchmark applications, namely matrix-matrix multiplication (MMM) and LU factorization, which require $O(N^3)$ floating point operations (N x N is the matrix size), are studied and implemented on the two MPoPCs. Large sparse real-world matrices from power engineering, with size of up to 10279 x 10279, are employed in the evaluation process. A large, complex real-world application, namely power flow analysis based on Newton's method [Tinney, et al., 1967] was parallelized and mapped onto the two MPoPCs. Its real-time solution is of critical importance to the security of any power grid and current solutions on cluster systems suffer many limitations [IEEE, 1992]. Efficient application mapping, dynamic task scheduling and load balancing techniques are proposed and analyzed on my MPoPCs. The innovative nature of this work has the potential to guide research in this arising field of high-performance reconfigurable computing.

MPoPCs sit between the two categories of chip multiprocessors (Section 1.2.2) by taking advantage of the field reprogramability of FPGAs: they are similar to the first category in that all PEs share the same microarchitecture and ISA (Instruction Set Architecture); it also shares MPSoC features since the PEs are simple and yet highly (but not fully) optimized for target applications. HERA targets data-intensive, matrix-based applications in general; however, the end user can choose certain features for the PEs as shown later. A distinct advantage of MPoPCs is that it can be customized in the field by the end user due to the presence of reconfigurable logic; in addition, this can be done at a very low cost and risk of design, implementation and verification. The PE configuration is closely customized and reconfigured to match application's characteristics and, hence, increase the resource utilization for high-performance. The author also emphasizes the importance of on-chip local memory due to the ever increasing memory-processor latency gap. This is similar to the recently announced Cell processor, where PEs are interconnected by a bus [Hofstee, 2005].

On the software side, programming reconfigurable MPoPCs, especially heterogeneous systems like HERA, is very challenging given the tremendous flexibility provided by MPoPCs. The performance of computing systems highly depends on a good match of the hardware system with the application. Efficient resource management is essentially the key to achieve high performance for designs based on reconfigurable logic. Based on the HERA design, a resource-oriented and architecture-conscious framework for mapping data-parallel applications (described at a high level) is proposed, in addition to dynamic resource management and reconfiguration schemes. The applications are profiled and then expressed using weighted task flow graphs

(wTFGs) consisting of SIMD and MIMD tasks associated; several parameters denote the complexity of each task. At static time, an application-specific HERA configuration based on an in-house designed *parameterized hardware component library* (PHCL) is synthesized for various performance-energy objectives. The architecture can be reconfigured at runtime as needed by the tasks. Then, a proposed runtime management approach takes advantage of HERA's mixed-mode parallelism in order to increase the resource utilization while at the same time optimizing the performance and/or consumed energy. During the execution of an application, this approach may dynamically repartition and redistribute active SIMD tasks among the available PEs (Processing Elements) in the system. Experiments with the parallel power flow analysis algorithm and singular value decomposition (SVD), which requires at least 20 times more FP operations than LU factorization, are performed to test the proposed framework. A HERA system-level energy model which is based on physical-level implementation data and run-time application statistics is proposed to guide the run-time scheduling decisions.

1.6 Dissertation Organization

Chapter 2 provides a technical background on FPGA devices and reconfigurable computing. Chapter 3 presents the design and implementation details of the two MPoPCs. The development of MMM, parallel LU factorization of large sparse matrices, parallel direct solution of sparse linear equations and parallel processing for power flow analysis on the MPoPCs, as well as related issues for mapping and scheduling are discussed in Chapter 4. Chapter 5 contains experiments and performance analysis.

Power characterization and system-level energy modeling are presented in Chapter 6. The resource management framework for HERA and experimental results are presented in Chapter 7. Finally, conclusions of this research and suggestions for future work are presented in Chapter 8.

CHAPTER 2

RECONFIGURABLE COMPUTING

FPGAs are the most common devices employed in reconfigurable computing. This chapter provides a technical background on FPGAs and discusses the most recent advances in FPGA architectures. It also contains some examples of coarse-grain reconfigurable systems. The current FPGA development approaches are also discussed in order to provide an introduction for our compilation methodology in Chapter 7.

2.1 Field-Programmable Gate Arrays

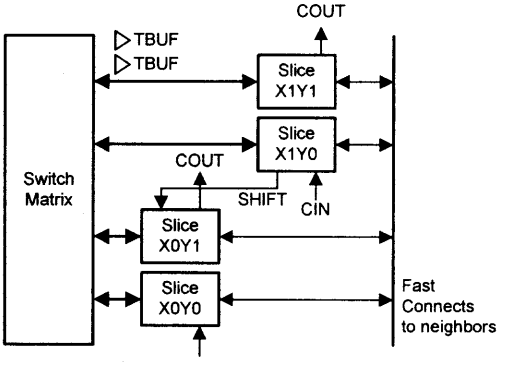
FPGAs are a class of integrated circuits (ICs) that contain arrays of pre-fabricated logic and interconnection modules whose functions are electrically configurable to meet specific design requirements by the user; this is done by using system development software after the ICs have been manufactured and delivered. FPGAs were introduced in the mid-1980s as alternatives to custom-designed MPGAs (Mask-Programmable Gate Arrays) in order to reduce dramatically the high NRE costs, long design cycles, and inherent risks associated with the latter, and provide the benefits of customized ASIC designs. Most modern FPGAs employ SRAM (Static Random Access Memory) technology to achieve programmability and comprise a matrix of configurable components, such as logic blocks, distributed and/or block memories, hierarchical fast routing resources and/or microprocessor(s) [Altera; Xilinx]. Both of the functions performed in the logic blocks and the routing of signals in the interconnection fabric are programmable by the SRAM bits connected to them; programming the SRAM bits

configures the FPGA. Most applications often require that FPGAs be configured only once. This is known as *static reconfiguration*. *Runtime reconfiguration* allows applications to dynamically change the configuration of FPGAs at runtime. An important feature in modern FPGAs is the support of partial runtime reconfiguration. The penalty resulting for this flexibility of FPGAs is larger signal delay and a lower system frequency compared to ASIC designs implemented with similar silicon processes.

The basic computational cell in a Xilinx Virtex II FPGA is a Configurable Logic Block (CLB) [Xilinx], shown in Figure 2.1, which is made up of four similar slices tied to a switch matrix for accessing the general routing fabric, with fast local feedback within the CLB. The output from the function generator in each slice drives both the slice output and the D input of the storage element. Figure 2.2 shows a detailed view of a single slice. Each slice includes two 4-input lookup tables, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements. As the diagram illustrates, the lookup tables can be configured and accessed in three different ways, including: 4-input LUT, 16 bits of distributed SelectRAM+ memory, or a 16-bit variable-tap shift register element. These LUTs are essentially 16 x 1 (with four inputs) or 32 x 1 (with five inputs) memory blocks used as universal function generators capable of serving as truth tables for the implementation of any arbitrary 4- or 5-input logic function. The extra multiplexers (MUXFx and MUXF5 in Figure 2.2) can be used to combine LUTs to realize functions with up to eight inputs.

When reconfigurable computing (RC) was introduced in the late 1980's, the largest FPGAs had only 2K gates of reconfigurable logic, far from enough real estate to

build computing systems. By the mid-90's, the size of reconfigurable devices increased to 50K gates of reconfigurable logic; but the continued low gate count, poor programming architectures, lack of partial reconfigurablity and high cost of these devices restricted the use of RC architectures for research and experimentation purposes. They were mainly used as highly integrated glue logic tying together the intelligent parts of systems or emulation engines for ASIC designs before they were fabricated.



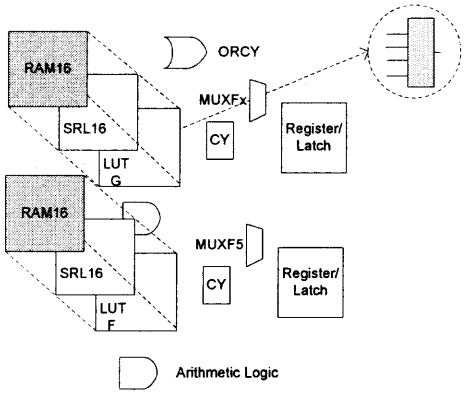


Figure 2.1 A CLB in Virtex II FPGAs [Xilinx]. **Figure 2.2** Slice configuration in Virtex FPGAs [Xilinx].

2.2 Recent Advances in FPGAs

With the arrival of million plus gates of reconfigurable logic on a chip in 2001 and the addition of high-performance RISC CPUs, block RAM, multi-gigabit high-speed serial I/Os, dedicated DSP logic, and other system enhancements, FPGAs have increasingly become system oriented (Systems-On-a-Programmable-Chip, SOPC) [Xilinx; Altera]. They have quickly taken over innumerable ASIC SoC designs with their flexible device integration capability, programmable I/O, very capable clock speed, and significantly lower overall design cost.

To give an idea of the state-of-the-art in FPGAs, let us consider the recently released Xilinx Virtex 4 FPGAs [Xilinx] shown in Figure 2.3. Virtex 4 employs a highly modularized architecture called the application-specific modular block (ASMBL), where the reconfigurable logic is structured into long, narrow stripes. Each stripe can be defined during the silicon manufacturing stage to contain either standard configurable logic elements or a function-specific block with specialized elements to handle DSP operations, memory, high-speed I/O, mixed-signal functions, or some other generic, yet application-optimized function. Its logic fabric and fixed blocks can all operate at 500-MHz clock rates. The largest available Virtex 4 device, XC4VFX140, is embedded with 63,168 slices and 9,936 Kbits of BlockRAM. Designers now have additional axis of flexibility to choose parts with varying mixures of special features more appropriate to their application.

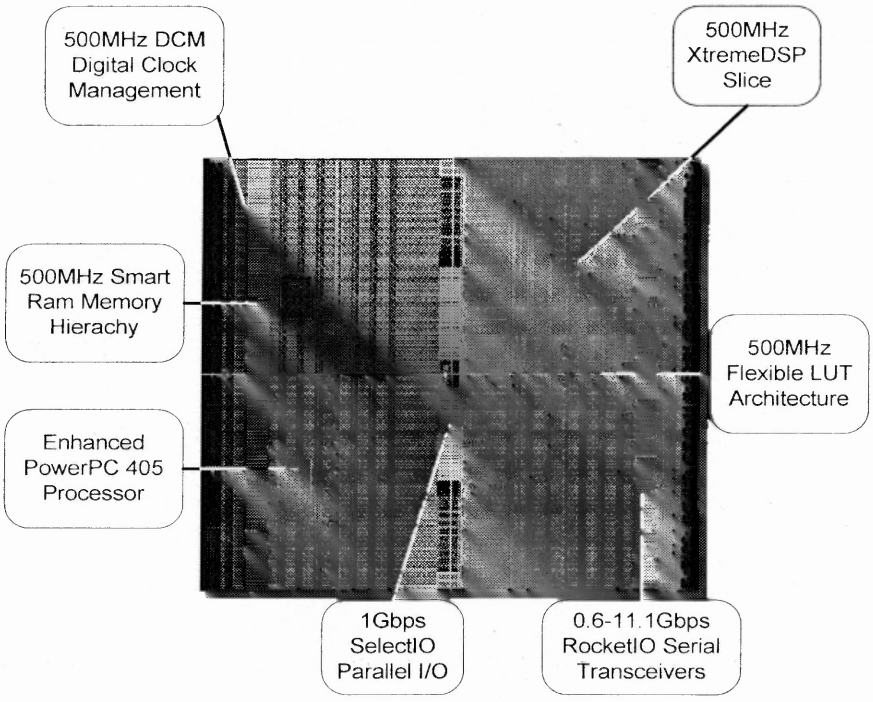


Figure 2.3 Virtex 4 FPGA [Xilinx].

2.3 Examples of Coarse-Grain Reconfigurable Architectures

Earlier FPGA-based reconfigurable computing systems were mostly *fine-grain* systems, where *processing elements* (PEs) typically comprised logic gates, flip-flops and LUTs operating at the bit level [Compton, et al., 2002; Prasanna, et al., 2002]. Fine-grain systems are difficult to program, inefficient in application mapping and take significant time to compile and reconfigure [Venkataramani, et al., 2003], which is required in these approaches for applications oversizing the available hardware. The most important reason was insufficient resources in the FPGAs at that time. On the other hand, as more and more resources were allowed on a single die, *coarse-grain* systems (where the PEs contain complete functional units like ALUs and/or multipliers operating upon multiple-bit words), have become more common [Singh, et al., 2000]. While overcoming the disadvantages of fine-grain systems, coarse-grain systems tend to have fewer long-distance control signals and more regular localized modules; these features favor multi-million-gate devices where wire delay is more of a limiting factor to the system frequency than gate delay. New FPGA architectures also favor coarse-grain designs. We are only interested in coarse-grain designs in this work. Table 2.1 shows a comparison of available coarse-grain reconfigurable systems.

Most of these coarse-grain systems appeared as coprocessors to offload the main processor of computation intensive cores, mostly from signal and image processing. Only Raw included a 4-stage pipelined FPU in its PEs. As the table shows, only a very small amount of memory was included and no general-purpose instructions were provided in these systems. *Finally, all of them were implemented by ASIC processes, although some of them were initially designed for FPGAs.* Thus they are not flexible

enough like FPGA-based reconfigurable systems to support significant resource management. We did not find comparable systems on FPGAs.

2.4 Design Methodology for Reconfigurable Machines

Traditionally, FPGA-based designs follow a very similar flow as that for ASIC designs, as shown in Figure 2.4. Hence, mapping applications to FPGAs have mostly considered a hardware expertise.

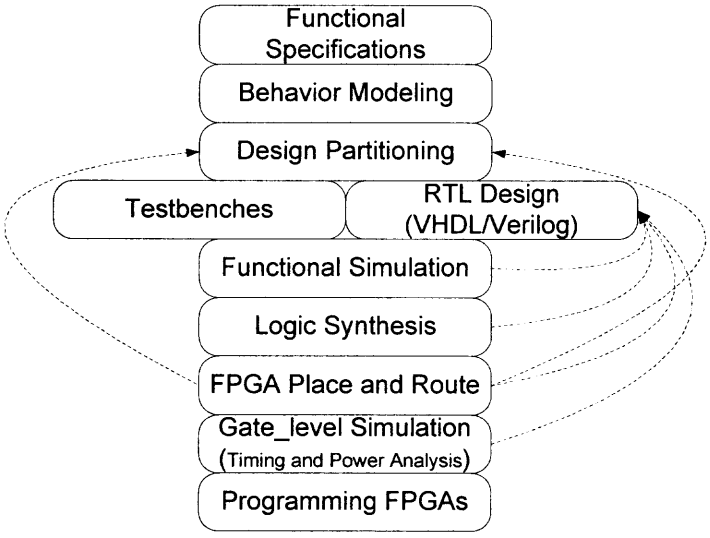


Figure 2.4 Conventional development flow for FPGA-based systems.

The entire procedure can be very time-consuming for multi-million-gate devices and the resulting configuration data can be used only for a fixed-size, specific device. The entire FPGA implementation procedure (from the *design partitioning*) is repeated if the target device changes. Runtime reconfiguration has made possible the concept of "Virtual Hardware" [Ling, et al., 1993], where the FPGA resources are assumed unlimited and applications are partitioned into function blocks that are then executed by time-sharing the same hardware in a specific order. Hardware virtualization allows to

Table 2.1 Comparison of Previous Coarse-Grain Reconfigurable Systems

	Comput. model	DataPath	Interconnect	Local on-chip memory	Program.	Application Domain
RaPiD[1]	Pipelined	16-bit	1-D Linear array	Small data memory	Rapid-C	Signal and image processing
RAW[2]	MIMD	32-bit, Pipelined	X routers	32KB Instruction memory and 32KB Data Cache	RAWCC	General-purpose and embedded computing
REMARC[3]	SIMD	16-bit	2-D mesh	16-entry data memory	N/A	Multimedia
MATRIX[4]	SIMD/MIMD	8-bit	2-D mesh, hierarchical buses	256 x 8-bit	N/A	General-purpose
MorphoSys[5]	SIMD	8- or 16-bit	2-D mesh, configurable segmented buses	No	SA-C	Data-parallel, compute–intensive applications
Chimaera[6]	SIMD	32-bit	Dynamic buses, crossbar	32bit memory	C-SIDE (Mixed C/VHDL)	Wireless communication, multimedia

1: Ebeling, et al., 1996; 2: Taylor, et al., 2002; 3: Miyamori, et al., 1999;
4: Mirsky, et al., 1996; 5: Singh, et al., 2000; 6: Ye, et al., 2000;

implement applications that are too large to fit on an FPGA. The major obstacle to its practical application is the significant overhead of reprogramming the hardware, which is typically on the order of tens to hundreds of milliseconds for current FPGAs [Xilinx; Altera]; this overhead may be larger than the actual computation time for small function blocks.

As expected advances in technology (Moore's Law) increase the resources on single programmable chips, the above design procedure becomes more and more time-consuming and cumbersome, and requires extensive expertise in both hardware and software. Motivated by this problem, the past few years have seen an increasing interest in developing tools to compile applications written in high-level programming languages for target FPGAs; such languages are C/C++ and Java [Gokhale, et al., 2000; Cardoso, et al., 2003; Najjar, et al., 2003, Venkataramani, et al., 2003]. These tools typically take the user application code and produce corresponding VHDL code (RTL level) or a circuit netlist (the output of *Logic Synthesis* in Figure 2.4), and then FPGA place-and-route tools map the design to FPGAs. They still follow the same design philosophy of APSCs, as discussed in Section 1.3. The major problem with most of these approaches that try to mimic conventional compilers in allocating and configuring silicon resources is the extreme difficulty in identifying required components and their interconnectivity. It is crucial to take into account the idiosyncrasies of the underlying reconfigurable system while these approaches often apply generic techniques. Moreover, their time-consuming procedure has to be to be repeated every time a change is made to the source code. Most of them implicitly or explicitly assume the concept of virtual hardware which requires full or partial run-time reconfiguration. While these

approaches can bring FPGAs closer to more users who are not familiar with hardware design methodology, their performance in terms of area and speed are still unsatisfactory compared to the VHDL-based manual designs.

CHAPTER 3

MULTIPROCESSORS ON A PROGRAMMABLE CHIP

Although the customization of hardware can lead to high-performance, it also limits the use of such systems due to the lack of elasticity in reusing and reprogramming functional units for various applications. Increasing the reusability of functional units is an effective way in reducing the number of required reconfigurations [Ghiasi, et.al., 2004; Cardoso, 2003]. Also, the continuous success of processor-based temporal computing platforms, including most current high-performance parallel systems, owes a great deal to their standard general-purpose and backward compatible architectures, and their standard programming environments; they protect and encourage long-term efforts and investments. It gives us a hint that in order to make reconfigurable computing machines mainstream computing platforms, standard architectures and microarchitectures, and corresponding development methodologies like those for microprocessors are absolutely essential.

This chapter discusses two approaches to FPGA-based MPoPC designs that I have implemented: CG-MPoPC, a reconfigurable IP-based MIMD MPoPC based on Altera FPGA devices, and HERA, a mixed-mode MPoPC machine based on the Xilinx Virtex II devices. Our target applications are matrix-based data-parallel and stem from the high-performance engineering and scientific fields. A pipelined IEEE-754 standard FPU was designed and implemented, and employed on both systems. The first MPoPC employs a configurable processor IP core from Altera optimized for platform FPGAs. Such RISC configurable soft cores have recently become available to greatly empower FPGA-based system implementations. Conventional processors gain in performance by

increasing the clock frequency; this results in intolerable high power consumption and the physical limits are often reached. IP configurable processors, on the other hand, provide extra opportunities in lower power consumption, higher transistor utilization, programmability and flexibility. The processor can be tailored to better meet the requirements of the application. The instruction set architecture (ISA), register file, software development APIs (Application Programming Interfaces), memory hierarchy and size, and communication channels can all be configured and extended as deemed appropriate. Also, standard and user customized logic engines can be easily added, modified or extended, as needed. We can identify critical instructions in the application code that affect performance the most and implement them in hardware. Configurable processor cores also provide us with substantial flexibility in SOPC integration. Such configurable IP cores are designed with a general-purpose microarchitecture and instruction set to achieve good performance for a large range of possible applications. Hence, the first system is intended for many diverse applications. However, the generality of such systems to provide the provisions for many scenarios leads to a rather low utilization of hardware resources and lower performance than a custom designed solution to any particular application. In contrast, a fully-customized and reconfigurable PE is designed and implemented for HERA in order to meet more stringent performance requirements. Moreover, HERA can be reconfigured dynamically at runtime to support a variety of independent or cooperating computing modes, such as SIMD, MIMD and M-SIMD, to best match in the time spectrum all subtask characteristics of a given single application. More discussion about mixed-mode computing follows in Section 3.2. In order to evaluate and compare the performance of

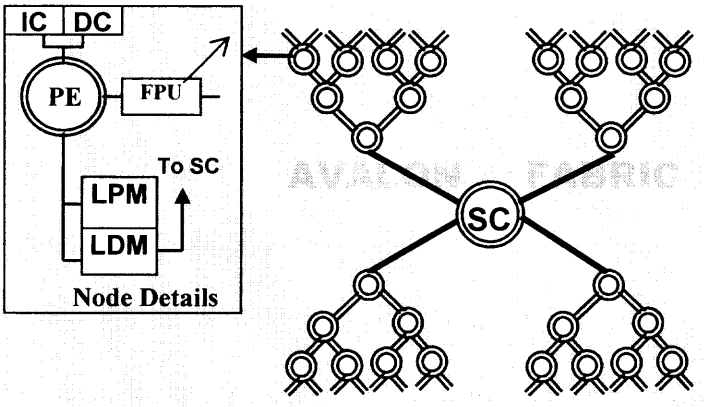
different interconnection networks, the PEs in the first MPoPC are interconnected via an X-tree network where HERA employs a 2-D mesh organization. The first approach takes much less time to develop and implement, and is easier to program than the HERA custom approach. We also employ FPGAs from the two major vendors, i.e., Altera and Xilinx, in order to compare the architecture capabilities and performance of different devices.

3.1 A Coarse-Grained IP-based MPoPC (CG-MPoPC)

3.1.1 Multiprocessor Architecture

We customize the MPoPC configuration to better match applications. Figure 3.1 shows one configuration of the CG-MPoPC. The PEs form multiple binary trees to support communication patterns in a matrix-based algorithm (details follow in Chapter 4). Each PE is guided by the SC (system controller) that utilizes the boot up code stored in the PE's private memory. An interrupt-driven control channel in a star configuration connects the SC to every PE. There is also a direct communication channel between the SC and the root of every binary tree. As the feature size of silicon processes enters the submicron range, wire delay becomes significant compared to logic delay. The routing of chip-level and clock signals tends to become more cumbersome in complex multi-million gate SOPC designs. In contrast, our binary tree network for data communications eliminates global transfers and is also scalable in size. The serial and TCP connections were implemented between the multiprocessor and the host PC. TCP

provides a flexible, quick and efficient communication channel in our parallel system, which can be accessed by all other hosts in the network.



SC: System Controller
IC: Instruction Cache
LDM: Local Data Memory
LPM: Local Program Memory
PE: Processing Element
DC: Data Cache
SDM: Shared on-board memory
FPU: Floating-point Unit

Figure 3.1 The CG-MPoPC architecture.

3.1.2 Processing Element

We employed a 32-bit Nios® [Altera] IP processor core from Altera to implement each PE and the SC. The Nios® RISC processor is fully configurable and its implementation yields over 200 DMIPS (Dhrystone MIPS) in the Altera Stratix II FPGA. It utilizes a 5-stage pipeline and conforms to a modified Harvard memory architecture. Configurable processors necessitate trade-offs between performance and the resources consumed. A typical Nios® processor in our machine consumes about 1600 logic elements (LEs). A pipelined IEEE 754 single-precision FPU and some trigonometric functions, such as sine and cosine, were implemented in hardware with every PE. They are needed by our target applications. These functions take considerable time if implemented in software. All these hardwired functions can be accessed by application code via custom-made instructions. The FPU runs at 128.3MHz for the 3-stage adder/subtractor, 150.8MHz for

the 5-stage multiplier and 165.4MHz for the 28-stage divider. These efficient realizations result in significant performance improvements for matrix operations [Wang, et al., 2004]. Taking advantage of the high density of new generation FPGAs, we are among the first ones to implement IEEE 754 FPUs in FPGA-based configurable parallel systems.

3.1.3 Memory Hierarchy Design

Since current configurable machines lack latency reducing software support, the memory design becomes a dominant factor in performance. Moreover, although new silicon technology and computer architecture advances facilitate faster processors, the performance gap between processors and memories tends to increase. If we rely solely on the on-board SRAM memory in our shared-memory multiprocessor, the overall speedup may drop substantially due to severe memory contention and large system synchronization. Fortunately, new generation FPGAs make available large on-chip memory with wide communication channels. Our FPGA-based multiprocessor architecture capitalizes on this advantage and forms several kinds of memories in order to maximize performance.

Every PE in our system has a local, exclusive on-chip program memory and a shared on-chip data memory. The PE shares its on-chip data memory with its sibling and parent in the binary tree, as shown in Figure 3.2. The sizes of the program and data memories for each PE are determined by the available memory capacity of FPGAs and the total number of PEs. The shared on-chip data memory improves the performance by minimizing the transfers of large blocks of data between memories. All the required

interconnection between on-chip memories and/or processors is implemented based on the multi-mastering, fully connected AVALON® bus of Altera. Thus, the communication bandwidth is quite large and the on-chip memory access time is only one clock cycle. All PEs share the on-board synchronous SRAM (SSRAM) memory, whose access takes at least four clock cycles, on the average. We implemented a controller to oversee the system's operation, and also pre-fetch instructions and data from the on-board memory into the PEs; the latter use the on-chip memory to run the application code because of its much lower latency compared to the on-board SRAM memories. On-chip data and instruction caches are also employed to reduce the memory access latency. We implemented a direct-mapping cache with the write-through policy in each PE. Our experimental results show that the speedup obtained by employing this cache can be more than 20%. The cache size and configuration can be tailored to the specific algorithm requirements due to the presence of configurable logic.

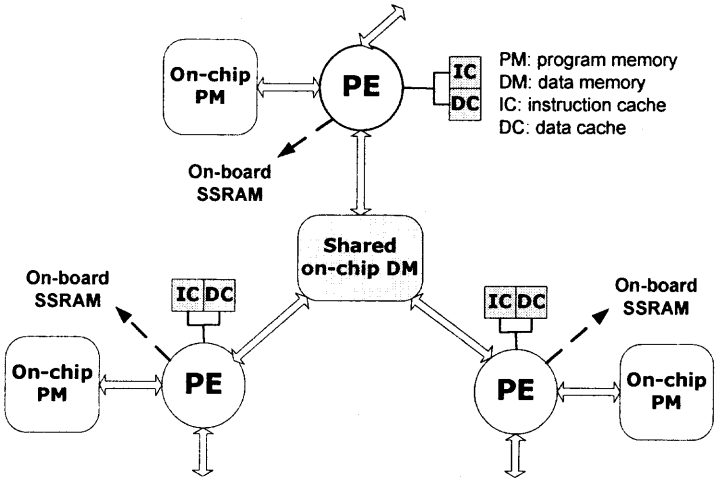


Figure 3.2 Memory configuration.

3.1.4 Implementation Results

An SOPC development board from Altera [Altera] was employed to implement the MPoPC. It is populated with the largest APEX20KE FPGA device, the EP20K1500EBC652-1x, which includes 51,840 logic elements and 442,368 bits of on-chip memory. The board also contains two banks of SSRAM chips for a total of 2 MB. Seven PEs, each with an FPU, plus the SC were fitted on the board. The system runs at 50MHz. EP20K1500E is a relatively slow device, which is built with a 0.18μm and 8-layer-metal process. It is enough to serve our purpose of an initial evaluation of the performance for real reconfigurable systems utilizing the new platform FPGAs.

3.2 HERA: A Reconfigurable Mixed-Mode Parallel Computer

From the application's point of view, the performance of general-purpose computing systems is not optimal for most subtasks due to the system's expected unsuitability; different subtasks in an application normally require different architectures for high performance. SIMD and MIMD are the two fundamental and complementary parallel modes of execution. SIMD's superior ability for data parallelism, often enhanced with low inter-PE communication and synchronization overheads, makes it superior to MIMD in performing fine-grain tasks [Parhami, 1995; Meilander, et. al., 2003]. Many numerical analysis algorithms, such as large-scale matrix multiplication and LU factorization, have a very high degree of structured, fine-grain parallelism and can benefit substantially from the SIMD mode. However, due to SIMD's implicit synchronization, SIMD machines are often under-utilized for applications involving dynamic parameters and an abundance of conditional statements. On the other hand,

MIMD machines consisting of independent PEs are good at conditional branching. Mixed-mode heterogeneous computing [Siegel, et. al., 1996], where the machine's operational mode (i.e., SIMD, MIMD, M-SIMD, etc.) changes dynamically as deemed suitable by the individual subtasks in an application, is an effective approach in alleviating such problems.

3.2.1 System Organization

Figure 3.3 shows the general diagram of our HERA machine with *m* x *n* PEs interconnected via a 2-D mesh network. Most matrix-based computations are well structured and map naturally to the 2-D mesh which is easily implemented by the FPGA place and route processes. We employ fast, direct NEWS (North, East, West and South) connections for communications between nearest neighbors. Nearest PE pairs on the same row or column can also communicate through one port of the data memory of the PEs to the west and north. Since every PE also has a *Local Control Unit* (LCU), most of the instruction decoding is carried out by the LCU. By giving the decoding work to the LCUs, we avoid broadcasting a large number of control signals to all the PEs. For debugging and system control, it is desirable for the host processor to have access to all the local program and data memories of the PEs. However, such an implementation has an adverse effect on system timing when the number of PEs increases. We designed a two-level bus scheme for global instruction distribution and communication. Every column has a Cbus and the eight Cbuses are connected to the *Column Bus*. Individual PEs or groups of PEs can be selected by their ID number(s) (address(es) on the

Cbus(es)). The bus control logic is implemented in the Sequencer. If we have more PEs, we may increase the number of bus levels.

The total number of PEs is determined by the available resources in target FPGA devices and the resource requirements of the application. The computing fabric is controlled by the system Sequencer that communicates with the host processor via the PCI bus. Interrupt logic between the Sequencer and the host processor is implemented.

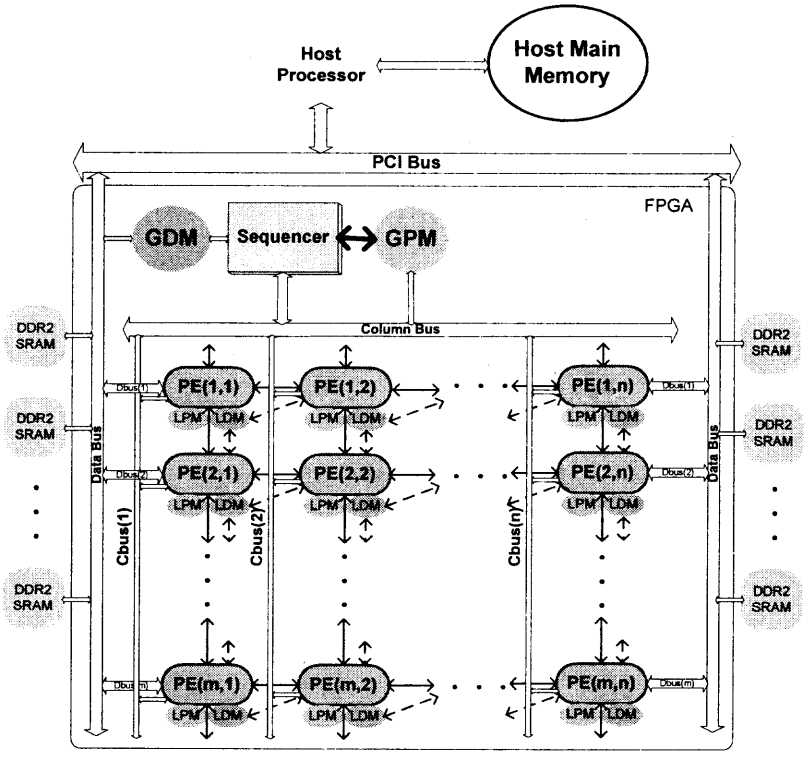


Figure 3.3 HERA system architecture.

The host can access the on-board DDR II SRAM and the on-chip memories of each PE. Each SRAM chip is owned exclusively by a group of PEs. The Global Control Unit (GCU), included in the system Sequencer, fetches instructions from the global program memory (GPM) for PEs operating in SIMD. Due to the presence of FPGAs,

besides that the system operating mode is reconfigurable at runtime, the capabilities of each PE and the number of PEs can be reconfigured based on the application's requirements. The host can load different FPGA images in the same C/C++ code to finish different subtasks at runtime. Thus, FPGAs provide another dimension of flexibility to optimize the hardware to match the specific characteristics of applications.

3.2.2 PE Architecture

In SIMD, we need to maximize the number of PEs in order to get the best possible performance. We employed a RISC or load-store architecture for our PE to save on hardware resources. Furthermore, the simplicity of the RISC architecture makes the implementation of the processor pipeline easy. Figure 3.4 shows the block diagram of the PE. All data paths are 32 bits. The PE contains several major components: a 7-stage, pipelined, 32-bit floating-point (FP) function unit (FFU), an LCU, 32-bit dual-port local program memory (LPM), 32-bit dual-port local data memory (LDM) and eight NEWS communication ports. Data in one of the NEW_IN registers can be sent to any of the four neighboring NEWS_OUT registers by using one instruction.

Our HERA design implements IEEE 754 single-precision pipelined FP operations in each PE. We employed a 3-stage pipeline in the FP adder, subtractor and multiplier, and a 28-stage pipeline in the FP divider. HERA supports both global and local PE masking. Every PE in the processor array is assigned an ID number that serves in global masking. The last seven bits of all the instructions select a particular PE or a group of PEs. Every PE holds a mask bit and computes the mask value with every instruction. A specific bit in instructions selects between global and local masking. Each

PE comprises 32 32-bit general-purpose registers (GPRs) and several system registers: local instruction register (LIR), local program counter (LPC), data memory address register (DMAR), program memory address register (PMAR), local status register (LSR), local masking register (LMR) and 1-bit operating mode register (OMR). Similar to some other RISC processors, the R0 GPR is fixed at zero.

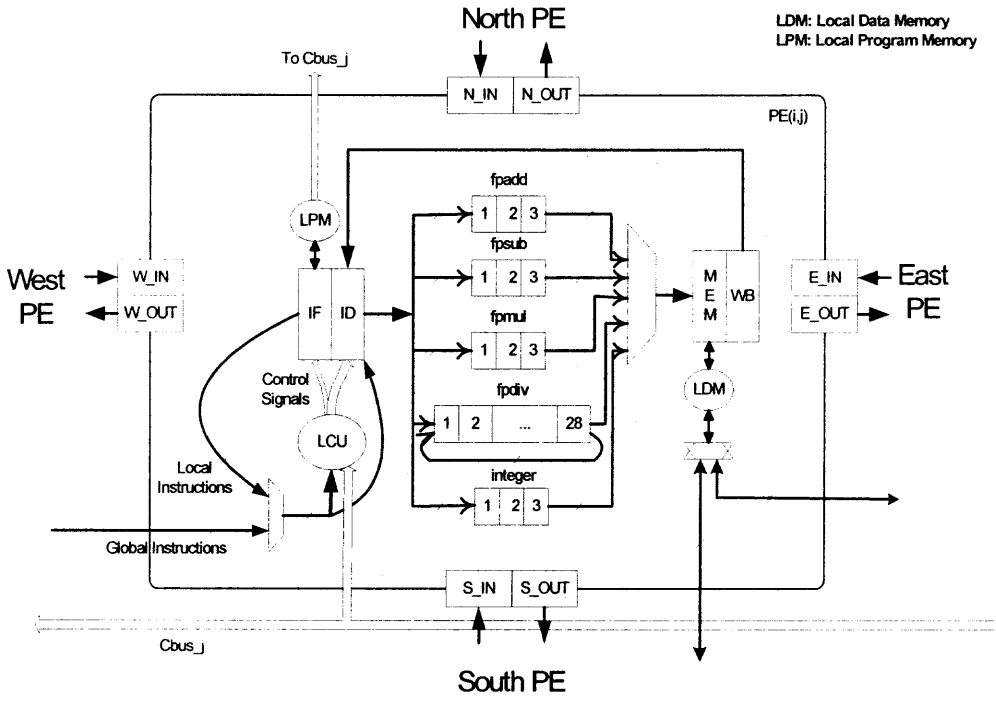


Figure 3.4 A HERA PE.

The operating mode of each PE is configured dynamically by the host processor through its OMR by using the Configure instruction: "0" indicates SIMD and "1" sets the PE into MIMD. All PEs operate in SIMD when powered up. To switch a PE to MIMD from SIMD, the sequencer first distributes the instructions to the LPM of the PE through the Column Bus and Cbus, and then sends a JumpI instruction to the PE with the starting address in the MIMD code. OMR is set to 1. To switch back to SIMD, OMR is reset to "0" and the PE then listens for the broadcasting of a global instruction.

The data in the registers and memories remain intact during switching. The instructions come from GPM in SIMD and from LPM in MIMD. The masking in the SIMD mode can use the PE's ID number and/or LMR.

3.2.3 Memory Configuration

The sizes of LPM and LDM were determined by the number of memory blocks in the FPGA device. They are configured as dual-ported 32-bit memories. Figure 3.5 shows the connections of the two memory ports of LPM and LDM. We tried to make the data memory as large as possible in order to reduce the data I/O time. The A port of LPM is connected to one Cbus, and serves as the interface to the sequencer and host processor. This way, the host processor has access to all LPMs. It sends application programs to every PE through this port from the Main Memory if the PE is to operate in MIMD. Port B of an LPM can be accessed by the local PE to get the instructions. An interesting feature of our design is the LDM interface. Our matrix algorithms usually involve frequent accesses of intermediate results in nearest neighbors. The NEWS network can handle well single word communication. However, it may take many cycles to transfer a large amount of data by using NEWS connections. Based on our experience with matrix algorithms on our IP-based multiprocessor machine, where PEs often use results from the east and north neighbors, we employed a shared dual-ported memory to address this problem. The A port of LDM is accessed by the local PE, and the B port is shared with the neighbors to the south and east. A PE can directly write to or read from the LDMs of its west and north neighbors via their B ports. Thus, we eliminate block data transfers between nearest neighbors. Another important use of the shared port is to

pipe the data assigned to each PE into its LDM at system initialization. The A port of LDM of the first PE on every row is also accessible via the data bus.

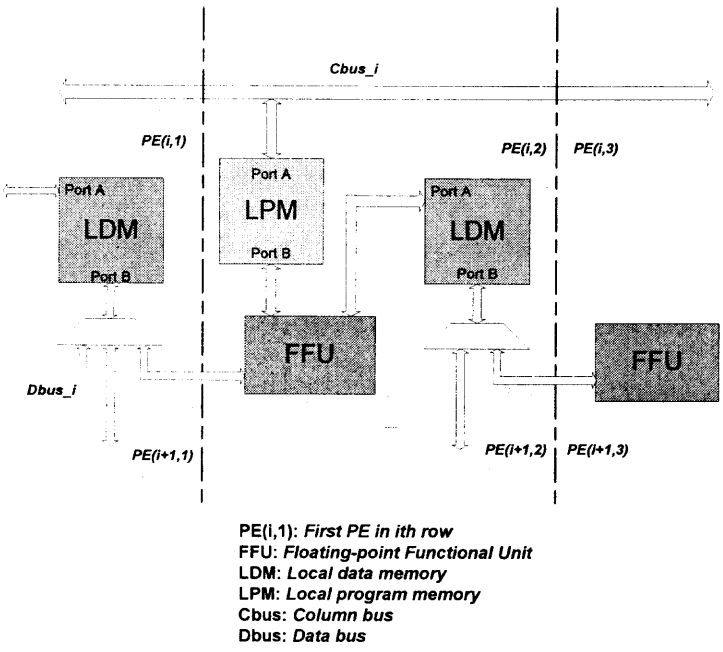
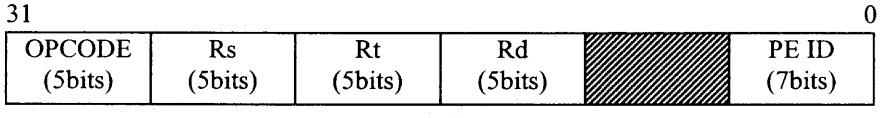


Figure 3.5 HERA memory interface.

3.2.4 Instruction Set

The efficiency of the PE greatly depends on its Instruction Set Architecture (ISA). Our design philosophy is to have a small and highly optimized instruction set for our target applications, while not losing generality for the sake of programming efficiency. The simplicity of the instructions also facilitates pipelined implementation. The instructions of HERA are classified into six major groups: integer arithmetic, FP arithmetic, memory access, jump and branch, PE communication, and system control, as shown in Table 3.1. Our target applications generally require intensive FP operations and demonstrate very limited control flow. Hence, we concentrate on optimizing the performance of FP instructions. All the instructions are 32 bits wide. We use a three-

field general format for all instructions as shown in Figure 3.6. Some fields are not used for some instructions that have one or two operands (see Table 3.1) and we still keep the alignment of the same operands in order to speed up decoding. System control instructions have special formats. An immediate FP operand is stored in the memory location immediately following the instruction. All the memory addresses are currently 10 bits long.



Rs: Source operand 1; **Rt**: Source operand 2; **Rd**: Destination operand

Figure 3.6 HERA general instruction format.

Table 3.1 The Instruction Set of HERA

#xxx: Memory address of 11 bits; **#*****: Immediate operand flag; **###**: PE ID number

Types	Instructions	Descriptions
Integer Arithmetic	ADD/SUB/MUL/DIV Rs, Rt, Rd	Integer add/sub/mul/div
	ADDI/SUBI/MULI/DIVI Rs/Rt, #***, Rd	Integer add/sub/mul/div with immediate data
Floating-point Arithmetic	FP_ADD/SUB/MUL/DIV Rs, Rt, Rd	Floating-point add/sub/mul/div
	FP_ADDI/SUBI/MULI/DIVI Rs/Rt, #***, Rd	Floating-point add/sub/mul/div with immediate data
Branch and Jump	JumpI #xxx	Jump to an immediate memory address
	JumpR offset(Rx)	Jump to the memory location at ((Rx) + offset)
	BNE Rs, Rt, #xxx	If (Rs) /= (Rt), then jump to the location at #xxx
Memory	LW (x) Rd, #xxx OR offset(Rx)	Load Rd with a 32-bit data from a memory address included in the instruction or computed by ((Rx) + offset) from memory source x (LDMs, LPM, SRAM)
	SW (x) Rs, #xxx OR offset(Rx)	Store 32-bit data in Rs to a memory address included in the instruction or computed by ((Rx) + offset) from memory source x (LDMs, LPM, SRAM)
PE Communication	Send_N/E/W/S Rs	Send the content of Rs to N/E/W/S_out port
	Get_N/E/W/S Rd	Get data from a N/E/W/S_in register into Rd
System Control	Sys_call (x)	Generate a soft call from PE(x) to the sequencer and exchange information with the sequencer.
	Standby	Pause and resume the execution of a PE
	Distribute ###, size	Distribute data to the LDM of every PE
	Collect ###, size	Collect data from the LDM of every PE
	Configure	Configure the computation mode and other system functions
	NOP	No operation

The destination register of Get_N/E/W/S instructions or the source register of Send_N/E/W/S instructions can also be one of the four NEWS_OUT registers. This way, data can bypass a PE to reach the next PE because we can use shared NEWS registers between PE pairs. The instructions support immediate, register and base addressing. The calculation of the effective address in base addressing is carried out by the control unit. The memory addresses in global instructions can be modified by the local PEs because we include local control logic. This feature is really useful and provides flexibility in some applications. The Standby instruction comes in pairs. The first instruction stops the PE's execution and sets the corresponding bit in the status register to "1", and then the PE waits for another Standby instruction to resume execution. The second Standby instruction supplies a jump address. The status bit is reset to "0" by the second Standby instruction. This status bit of all the PEs is monitored by the sequencer.

HERA can be partitioned at run time into several islands, each comprising a group of PEs running in SIMD or MIMD. The partitioning is achieved by global and local PE maskings; the mask status is stored in the *Global Mask Register* (GMR) and *Local Mask Register (LMR)*, respectively. A PE in SIMD is active only when both registers are set. The LMR can be set by executing locally a comparison instruction. Every PE is assigned a distinct ID that serves in global masking. The last seven bits of an instruction in SIMD form three fields: 3 bits each for the row and column address, and 1 bit for masking. A "1" in this bit sets the GMR of all the PEs in the column and a "0" only sets the GMR of the specific PE whose address is contained in the instruction. Combined with the PE ID and appropriate masks, the system can be configured

dynamically into a mixed-mode computing system capable of supporting simultaneously SIMD, MIMD and multiple-SIMD.

3.2.5 Implementation Results

Our first implementation was carried out on the high-performance WILDSTAR II-PCI FPGA board from Annapolis Micro Systems [AnnapMicro]. The board is populated with two Xilinx XC2V6000-5 Virtex II FPGA devices and 24MB of DDRII SRAM memory (12 chips). The XC2V6000 devices are embedded with 33792 slices and 144 BlockRAM (18Kbits each). The board communicates with the host computer via the PCI bus interface. Every PE was assigned 4KB for LPM and 8KB for LDM. The interface to the PCI bus operates at 133MHz and the datapath is 64 bits. The FPU frequencies after place-and-route for the FPGAs we are using are 163.2MHz (add/sub), 172.5MHz (mul) and 172.2MHz (div). The performance of our FPU could be enhanced by adding more pipeline stages. The computing fabric is clocked at 125MHz. The system frequency is limited by the inter-FPGA communication channel speed. We could also employ a commercial IP package and a more recent FPGA to further improve the system performance. About 50 APIs are implemented to facilitate the communication between the C/C++ application running on the host and the parallel program on HERA. We removed the subtractor and divider from each PE in the case of matrix multiplication, thus allowing us to implement 64 PEs in the two FPGAs. For LU factorization with FP arithmetic, 36 PEs did fit in the two devices. Our hardware design was implemented in VHDL and can easily retarget other FPGA boards.

CHAPTER 4

APPLICATION STUDY

4.1 Generalized Cannon's Matrix-Matrix Multiplication Algorithm

Cannon's matrix-matrix multiplication (MMM) algorithm [Cannon, 1969] is for a memory efficient parallel implementation on torus-connected processor arrays, where each processor communicates directly with its immediate neighbors in the four NEWS directions. The original algorithm assumes that the input matrices and the partitioned matrix blocks are all square. In our implementation, however, matrices A and B for A x B can be of any shape and size (still, the number of rows in A and the number of columns in B should be the same).

4.1.1 Data Partitioning and Mapping

Assume PEs are organized in a q x q 2D torus. Let A and B be matrices of size $N1$ x $N2$ and $N2$ x $N3$, respectively. We assume that the on-chip memory can store $3m^2$ floating-point elements. To be able to store complete blocks from the input and output matrices, the maximum size of a matrix block should be m x m. Let $p1 = \lfloor N1/(q*m) \rfloor$, $p2 = \lfloor N2/(q*m) \rfloor$ and $p3 = \lfloor N3/(q*m) \rfloor$. In general, we first partition A and B into a 2 x 2 block-based matrix as shown in the example of Figure 4.1, in such a way that the sizes of $A(1,1)$ and $B(1,1)$ are $\{p1*(q*m)\} \times \{p2*(q*m)\}$ and $\{p2*(q*m)\} \times \{p3*(q*m)\}$, respectively. The remaining blocks $A(2,1)$, $A(1,2)$ and $A(2,2)$ of A are decomposed into blocks with maximum dimension m. B is partitioned similarly. Blocks $A(1,1)$ and $B(1,1)$ are then

partitioned into $p1$ x $p2$ and $p2$ x $p3$ blocks of size $(q*m)\mathrm{x}(q*m)$ again and are distributed into the processors in a cyclic checkerboard-like fashion.

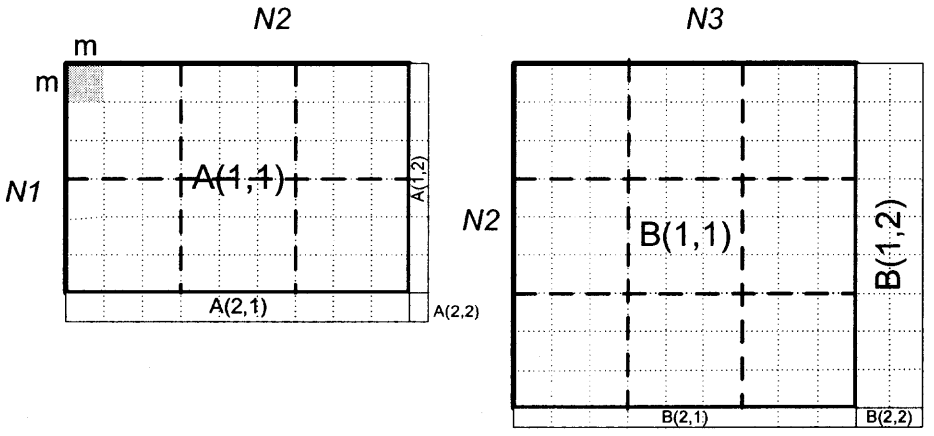


Figure 4.1 A partitioning example for matrices A and B $(q = 3, p1 = 2, p2 = 3, p3 = 3)$.

4.1.2 Dynamic Mixed-Mode Scheduling on HERA

If A and B are square, and can be partitioned into an integer multiple of q blocks, then Cannon's algorithm works best in the SIMD mode; all the PEs are then busy all the time, except during the initial alignment. If A and B are not square or cannot be partitioned in such a way that N (the matrix dimension) is a multiple integer of q * m, then the multiplication of the border blocks is not efficient in the SIMD mode since the sizes and numbers of blocks are irregular. Then, some PEs are idle while other PEs are busy at some point because SIMD is an implicitly synchronous mode. We solved this problem by changing the computation mode of the PEs. Also, we skip the initial alignment by assigning data blocks in a pre-skewed way. Because our PE is pipelined, we assume that multiplication, addition and shift operations all take one clock cycle, T_{clk}. The total execution time for Cannon's procedure on one partition is

approximately $T_c(n) = q*(2T_{shift} + T_{mul} + T_{add}) = 2(\frac{n^3}{q^2} + \frac{n^2}{q})*T_{clk}$, assuming that the size of the submatrix is n x n.

The dynamic mixed-mode scheduling procedure for our modified Cannon's algorithm on HERA is as follows:

(1) Carry out block multiplications involving $A(1,1)$ * $B(1,1)$ by using Cannon's algorithm; the total time is about $p1*p2*p3*T_c(n)$. All the PEs are configured into SIMD and take part in this step.

(2) If the size of $A(1,2)$ and/or $B(2,1)$ is larger than $\frac{1}{2}(q*m)$, then carry out $A(1,1)$ * $B(1,2)$ and/or $A(2,1)$ * $B(1,1)$ in SIMD using Cannon's procedure.

(3) We define a job as a multiplication of two blocks. Jobs are divided into two groups: SIMD and MIMD jobs. SIMD jobs are those corresponding to similar numbers of operations on the PEs. The remaining jobs go to an MIMD queue. Count the number of jobs and their associated numbers of operations in the remaining work. Determine the IDs of PEs that will work in the SIMD or MIMD mode based on the job information.

(4) Configure individual PEs in the system into either the SIMD or MIMD mode based on the decision in the previous step. The system now works in the mixed mode. Assign the SIMD jobs to the PEs running in the SIMD mode and distribute the MIMD jobs to the PEs running in the MIMD mode.

For the calculation of the quadrants in the resulting matrix, $A(1,1)$ * $B(1,1)$, $A(1,1)$ * $B(1,2)$ and $A(2,1)$ * $B(1,1)$ consume most of the execution time. In all the steps, except Step 1, data locality has priority in job assignments.

4.2 Parallel LU Factorization of Large Sparse Matrices

4.2.1 Overview of LU Factorization

Consider the solution of a system of simultaneous linear equations in the form $Ax = b$, where A is a large sparse N x N nonsingular matrix, x is a vector of N unknowns and b is a given vector of length N. A widely employed direct method is LU factorization that works as follows. We first factorize A so that $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix. Their elements can be determined by the following equations, respectively, if L has all 1's in its diagonal [Duff, et al., 1990].

$$L_{ij} = (A_{ij} - \sum_{k=1}^{j-1} L_{ik} * U_{kj}) * \frac{1}{U_{jj}}, \quad for\ j \in [1, i-1] \tag{4.1}$$

$$U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik} * U_{kj}, \quad for\ j \in [i, N] \tag{4.2}$$

Once L and U are formed, the unknown vector x can be identified by forward reduction and backward substitution, respectively, using the two equations $Ly = b$ and $Ux = y$. Since LU factorization is a computation-intensive procedure, its parallel solution has been a quite active research area. Thus, plenty of parallel techniques have appeared in the literature.

Matrices appearing in electric power applications, such as power flow, transient analysis and contingency analysis, are very large and extremely sparse [IEEE, 1992]. The average number of non-zero elements per row is normally around four, as shown in Table 4.1. The bus admittance (Y_{bus}) matrices in this table of dimension 1648, 7917 and 10279 represent Northeastern U.S. power networks. Matrix *BCSPWR10* is from the Boeing Harwell collection in the Matrix Market [Matrix Market] and represents an Eastern U.S. electrical power system. Although the LU factorization of sparse matrices

potentially has fewer operations than that of dense matrices, it can suffer tremendously from dynamic fill-ins. Many good parallel direct solvers for sparse matrices have been developed [Gupta, 2002; Duff, 1998], such as SuperLU [Demmel et al., 1999] and S+ [Fu, et al., 1998]. These packages are often optimized for proprietary parallel computers. However, the performance of such solvers has not been thoroughly analyzed for power matrices. In [Gupta, 2002], the speedup of the best solver for circuit simulation matrices, which are similar to power matrices but more dense, was shown to decrease with increases in the matrix sparsity. Some other solvers show no speedup at all for such matrices. Some research also has shown that SuperLU does not show performance gains for circuit simulation matrices [Bomhof, et al., 2000]. Coarse-grain parallel algorithms based on network partitioning have been demonstrated to be very efficient and promising for power matrices [Koester, et. al., 1994; Chen, et al., 2005].

Table 4.1 Sparsity of Benchmark Power Matrices

Dimension of the bus admittance matrix (Y_{bus})	1648	5300 (BCSPWR10)	7917	10279
NNZ*	6680	21842	32211	37755
NNZ per row	4.05	4.12	4.07	3.67

* NNZ: Number of non-zero elements

The sparse Y_{bus} matrix can be reordered into the DBBD (Doubly-Bordered Block Diagonal) form shown in Figure 4.2 by a heuristics-based algorithm [Sangiovanni-Vincentelli, et al., 1977]. The basic idea behind such a reordering algorithm is to divide an interconnected network into independent sub-networks and a collection of cutting nodes. This way, LU factorization can first be applied to completely independent sub-networks, thus speeding up the algorithm dramatically. The information corresponding to the cutting nodes is then processed at a lower rate.

$$\begin{pmatrix} A_{11} & 0 & \dots & 0 & A_{1n} \\ 0 & A_{22} & \dots & 0 & A_{2n} \\ \vdots & 0 & \dots & 0 & \vdots \\ 0 & 0 & \dots & A_{n-1n-1} & A_{n-1n} \\ A_{n1} & A_{n2} & \dots & A_{nn-1} & A_{nn} \end{pmatrix}$$

Figure 4.2 Sparse DBBD matrix format.

In the DBBD form of Figure 4.2, the A_{ik}'s represent matrix sub-blocks (sub-networks) and all the non-zero elements in the matrix appear only inside these sub-blocks. For every fixed i, the blocks A_{ii}, A_{in} and A_{ni} are said to form a *3-block group*, where $i \in [1, n\text{-}1]$ and $n \leq N$. A_{nn} is known as the last block and represents the cutting nodes. The A_{ii}'s will be referred to as *diagonal blocks*, and A_{in} and A_{ni} will be called *right border block* and *bottom border block*, respectively, where $i \in [1, n\text{-}1]$. A_{in} and A_{ni} represent the couplings (connections) between the nodes in the *i-th* independent sub-network and the cutting nodes. Since all non-border, off-diagonal blocks contain only 0's, if we apply Eq. (4.1) and (4.2) to a DBBD matrix we can find out that there will be no fill-ins in these blocks during factorization. Thus, the resulting matrix keeps the same DBBD form, as shown in Eq. (4.3).

$$\begin{Bmatrix} L_{11} & 0 & 0 & 0 & 0 \\ 0 & L_{22} & 0 & 0 & 0 \\ 0 & 0 & L_{33} & 0 & 0 \\ 0 & 0 & 0 & \dots & \dots \\ L_{n1} & L_{n2} & L_{n3} & \dots & L_{nn} \end{Bmatrix} \begin{Bmatrix} U_{11} & 0 & 0 & 0 & U_{1n} \\ 0 & U_{22} & 0 & 0 & U_{2n} \\ 0 & 0 & U_{33} & 0 & U_{3n} \\ 0 & 0 & 0 & \dots & \dots \\ 0 & 0 & 0 & \dots & U_{nn} \end{Bmatrix} \tag{4.3}$$

where

$$A_{kk} = L_{kk}U_{kk}$$

$$U_{kn} = L_{kk}^{-1}A_{kn}$$

$$L_{nk} = A_{nk}U_{kk}^{-1}, \text{for } k \in [1, n-1]$$

$$L_{nn}U_{nn} = A_{nn} - \sum_{k=1}^{n-1} L_{nk}U_{kn}$$

The calculations of L_{kk}, U_{kk}, L_{nk} and U_{kn} for different k's (i.e., 3-block groups) are independent of each other. So, we can distribute different 3-block groups to different processors to be factored in parallel, with no data exchanges until the factorization of A_{nn}. The last block, A_{nn}, requires data produced in all the right and bottom border blocks, so its factorization is the last step. To factor the last block, pairs of blocks are first multiplied in parallel to produce $A_{nn}^{k} = L_{nk}U_{kn}$, for $k \in [1, n\text{-}1]$. The summation of the n-1 products obtained for the different values of k and its addition to A_{nn} is needed to factor the last block. This summation is carried out along the binary tree in parallel by the other processors and the results are sent to the processors assigned the last diagonal block (for a highly parallel approach). We can see that the DBBD format presents great advantages for parallel implementation. A group of the three processors sharing the same data memory in a sub-tree were used to factor the last block.

4.2.2 Near-Optimal Ordering Selection

Our earlier research revealed that the total number of independent blocks and the size of the last block can have a significant impact on the entire factorization time. The sizes of the blocks after ordering largely depend on the physical characteristics of the matrices and the ordering parameters, such as the maximum number of nodes allowed in a block (sub-network). While the problem of determining an optimal ordering is NP-complete, we can select a near-optimal ordering for a given matrix based on our LU factorization algorithm and the target architecture. The best solution is also application-dependent.

We find here a near-optimal ordering by applying the following criteria in decreasing order.

1. Roughly decide a size range for the last block by setting different limits for the maximum number of nodes (*MaxNodes*) in each independent block. Increasing the limit decreases the size of the last block and the total number of independent blocks. Nonetheless, we may reach a point where further increasing the limit may not result in a big difference. If several such points exist, choose the one produced with the smallest value of *MaxNodes* in an effort to reduce the sparsity inside independent blocks. In general, the larger the block size, the more the fill-ins (i.e., more operations) produced during the factorization. Figure 4.3 shows our results as a function of *MaxNodes* in a block for the 10279-Y_{bus} system. The figure shows that a near-optimal ordering can be obtained by limiting *MaxNodes* between 150 and 200. Further increasing *MaxNodes* decreases the last block size faster than the total number of blocks. This, of course, is undesirable since the sparsity in the independent blocks increases.
2. The imbalance in the block size (n_i x n_i is the size of the *i-th* diagonal block) should be as small as possible.
3. Let p be the total number of PEs and n-1 the total number of 3-block groups. We should keep $\{(n\text{-}1) \bmod p\}$ as large as possible for good load balancing at the end.
4. The sparsity of each diagonal block should be as small as possible.

To illustrate the effect of ordering the matrix, the location of non-zero elements in the original 10279-Y_{bus} and the corresponding DBBD matrices for *MaxNodes* =180 are presented in Figure 4.4.

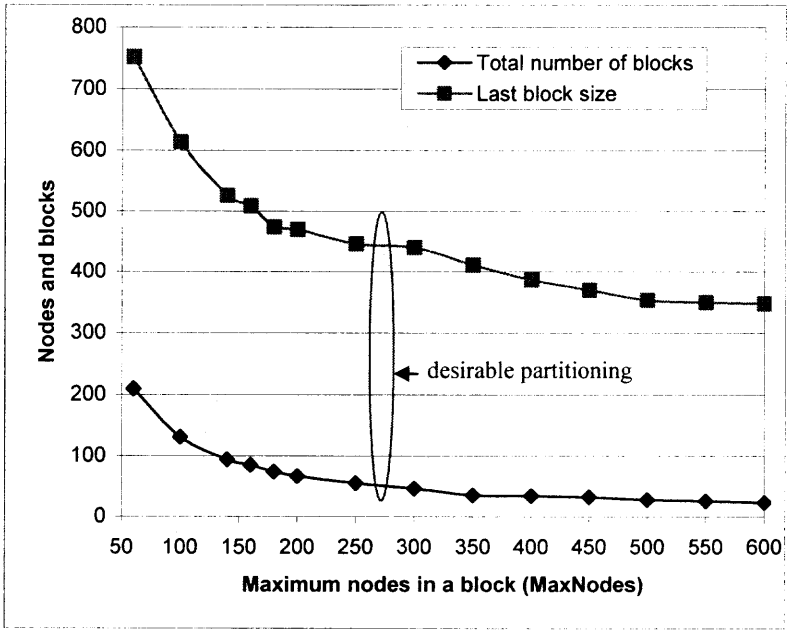


Figure 4.3 DBBD ordering for a matrix of size 10279 x 10279.

(a) Original 10279-Y_{bus} matrix.

(b) DBBD-ordered 10279-Y_{bus} matrix.

(*MaxNodes* = 180)

Figure 4.4 The non-zero elements in the 10279-Y_{bus} and the corresponding DBBD matrices.

4.2.3 Minimum Degree Ordering

For extremely sparse matrices, such as those from power electric networks, the matrix blocks in the DBBD matrices are still very sparse. Hence, efficient ordering techniques are preferred inside the matrix blocks in order to reduce the number of floating-point operations for DBBD LU factorization and further improve the performance. This is more important for very large matrices. For symmetric matrices, *minimum degree ordering* (MDO) [George, et. al., 1989] has been demonstrated to be able to significantly reduce fill-ins during LU factorization. A column-based approximate ordering algorithm [Amestoy, et. al., 1996] based on the MDO can be applied to nonsymmetric matrices.

4.2.4 Dynamic Task Scheduling

We need good scheduling to translate the hardware parallelism into high speedups for real applications. By ordering the matrices into the DBBD form, we eliminate all data dependences between different blocks during the factorization of the 3-block groups. If each processor operates on a distinct 3-block group, then this elimination of all inter-processor communications in this phase of the process is obviously of utmost importance. However, due to the very high sparsity of power matrices, the independent diagonal blocks and border blocks in the DBBD matrices are still sparse. The problem of fill-ins is still visible but to a lower extent. Moreover, the size of these blocks may have a large variance. All these factors along with memory contentions in shared-memory implementations contribute to unpredictable execution times. Therefore, a dynamic load balancing strategy is needed to reduce the effect of uncertainty during factorization. There have been some static and dynamic scheduling algorithms for fine-

grain parallel LU factorization [Fu, et al., 1996; Gupta, 2002; Duff, 1998], where the data dependences and related communication costs are the main concerns. These studies have assumed parallel systems that differ from ours in the granularity, the embedded interconnection networks and other components. Since the DBBD form eliminates data dependences between 3-block groups, our main focus is to reduce the idle time of processors due to different values for fill-ins, irregular sizes and sparsity of blocks. To take advantage of the network architecture in our system, we propose a centralized scheduling and load balancing approach. Centralized algorithms perform well for coarse-grain tasks and systems of low-to-medium size where quick and more comprehensive decisions can be made based on global information. This choice also minimizes the scheduling overhead, which is often a major disadvantage of distributed dynamic scheduling and load balancing. We employ the SC to take care of load balancing at runtime; in this approach, all processors report their load information to the controller. Configurable logic allows us to customize the hardware design at any time in order to facilitate software optimizations and to better utilize resources.

4.2.4.1 Task Definition

DBBD-based parallel LU factorization involves four types of jobs:

(1) *FAC*: Independent factorization of all the 3-block groups.

(2) *MUL*: multiplication of the factored border block pairs ($L_{nk}U_{kn}$) and (local) accumulation of the partial products inside each PE to later produce the inner product $\sum_{k=1}^{m_i} L_{nk}U_{kn}$, where m_i is the total number of 3-block groups assigned to PE_i

and $\sum_{i=1}^{p} m_i = n-1$. Every resulting product has the same size as A_{nn}. The *MUL* work on pairs of blocks cannot be scheduled until the respective *FAC* work has finished.

(3) *ADD*: Addition of two partial products from the *MUL* work.

(4) *LAST*: Parallel LU factorization in A_{nn} upon finishing all other factorization and multiplication work; this work begins with the synchronization of the involved PEs. The last block is normally dense.

The general task format is *FAC/MUL/ADD*$\{n_i, n_n, \#xxx\}$, where #xxx is the starting memory address for the *i-th* 3-block group and n_n x n_n is the size of the last block. These three classes of tasks are implemented in assembly code and stored in the local program memory of every PE.

4.2.4.2 State Information

The global information used by the SC includes at least the:

- Total number of PEs, p.
- Size of the matrix, N x N.
- Number of diagonal blocks, n-1.
- Size of the last block, n_n x n_n.
- Size of the diagonal block in every 3-block group, n_i x n_i, where $1 \leq i \leq n$-1.
- NZ numbers in the *i-th* 3-block group; they are represented by *nzd(i)*, *nzu(i)* and *nzl(i)*, where $1 \leq i \leq n$-1 and stand for NNZ in the diagonal, upper and lower block, respectively.
- Memory addresses for matrix blocks in the on-board memory.

The task pool profile includes at least the:

- Total number of tasks, N_t.
- Approximate execution time *T(k)* of task k.
- Number of remaining tasks, N_r.
- The candidate PE(k) for each remaining task k.

The SC keeps a load index $L(j)$, $1 \leq j \leq p$, for each PE that includes, among others, the:

- Task type (*FAC/MUL/ADD*) and corresponding information (size, NNZ, etc.).
- Starting time of the assigned group.
- Progress in the current task: the percentage of remaining work.
- Expected task completion time.
- Possible next group for this PE.
- List of finished tasks.
- dist(j): communication distance between the PE and the SC expressed in number of hops.

Part of this information is generated by the PEs at runtime and it is up to the SC to probe this information at the appropriate time.

4.2.4.3 Dynamic Scheduling Procedure

Dynamic task scheduling for load balancing is carried out by the SC as follows.

Step 1: Get task information from the host; the matrix blocks are assumed stored in the on-board shared memory.

Step 2: Approximate the execution time for each task based on the NNZs (including the fill-ins) and the size. Set up the task and PE load profiles. The total numbers of operations are approximated by:

FAC: $\frac{nzd(i)}{2} + nzl(i)$ (division), $\frac{4n_i + 1}{6} * nzd(i) + \frac{n_i - 1}{2} * (nzu(i) + nzl(i))$ (mul/add)

MUL: $\max\{nzl(i), nzu(i)\} * n_n$ (mul/add)

ADD: assume that the partial products are dense and the number of *mul/add* operations is n_n^2.

Step 3: Put the tasks in a queue by ordering them in descending order with respect to the number of operations.

Step 4: Assign the largest available job from the top of the task queue to an available PE and continue assigning tasks from the top of the queue until all PEs are busy.

Step 5: Reevaluate and update the load information for all PEs.

Step 6: Speculate the next task assignment for PEs and pre-fetch data for them. If $N_r > p$, schedule the largest task first to the PE with the lowest expected task completion time; if $N_r < p$, try to distribute the tasks to the PEs under different parents.

Step 7: Probe the work status of each PE and update its load information in fixed time intervals.

Step 8: If a PE becomes available for a new task, the SC should send the task scheduled in Step 6 and then should go back to Step 5. If the pre-fetching of the data for the task has not finished, the original memory address for the data should be sent to the PE. If the task queue is empty, go to next step.

Step 9: If a PE is idle and the task queue is empty, the SC should first check the status of the processors along the summation tree to locate its nearest busy processor. If the idle processor is one of the two direct neighbors of a busy processor, then the SC should modify the ongoing task of the working processor and the idle processor should be asked to immediately share its work via the shared memory (i.e., without any data transfer). If the nearest busy processor is not a neighbor, the SC should further decide whether it is cost effective to ask the idle processor to help the working processor. The decision should be based on the communication cost (distance and volume of the data block) of the two processors, the size of the currently executing task, the progress (i.e., the current

running time divided by the expected task completion time) and the type of the current task. If it is an *FAC* task, the idle processor should begin to multiply the pair of border blocks following factorization in the working processor. If the working processor is in the multiplication phase and the percentage of remaining work is greater than 33% (this experimental number is based on the computation/communication time ratio in our machine), then the SC should copy half of the remaining data to the idle processor and modify the working processor's load information. The multiplication results should be collected along the binary tree.

At any time during this procedure PEs can interrupt the SC for a task request or to report exceptions.

4.2.4.4 Theoretical Performance Analysis

Since there are three basic operations in parallel DBBD LU factorization, namely LU factorization of 3-block groups, multiplication of border blocks and addition of partial sums for submatrices of the same size as the last diagonal block, we first derive the execution times of these operations. The resulting equations are to be used to some extent by the SC for dynamic load balancing. We also employ them in theoretical performance analysis.

A. Execution times of basic operations

We assume that the +, - and * floating-point operations take the same amount of time, T_f, and floating-point division takes $4T_f$ time. These assumptions are reasonable for

advanced floating-point units. Given a 3-block group with a diagonal block of size n_i, the total number of operations for either multiplication or addition is:

$$\frac{2n_i^3 + 3n_i^2(2n_n - 1) - n_i(6n_n - 1)}{6} \tag{4.4}$$

and the number of divisions is:

$$\frac{n_i^2 + n_i(2n_n - 1)}{2} \tag{4.5}$$

The total execution time to factor such a 3-block group is a function of n_i and n_n, and is given by:

$$T_{lu3}(n_i, n_n) = \{\frac{2}{3}n_i^3 + (2n_n + 1)n_i^2 - \frac{5}{3}n_i\} * Tf \tag{4.6}$$

The total computation time for the multiplication of two matrix blocks, $A_{nn}^i = L_{ni} * U_{in}$, where A_{nn}^i is the n_n x n_n *i-th* partial product of A_{nn}, L_{ni} is the *i-th* n_n x n_i factored lower block and U_{in} is the *i-th* n_i x n_n factored upper block, is:

$$T_{mul}(n_i, n_n) = 2n_n^2 * n_i * Tf \tag{4.7}$$

We can see that n_i dominates the factorization time and n_n dominates the multiplication time in the processing of the 3-block group.

The time required to add two n_n x n_n matrices is:

$$T_{add}(n_n) = n_n^2 * Tf \tag{4.8}$$

With on-chip memory every access takes just one clock cycle (assuming no bus contention). If T_{clk} is the clock period, reading/writing a matrix of size n_n x n_i or n_i x n_n in floating-point representation requires time:

$$T_{mem}(n_i, n_n) = K * n_i * n_n * T_{clk} \tag{4.9}$$

where K is a constant associated with the processor and its shared memory. In our system, K is between 3 and 5 if the algorithm is coded in assembly, and between 10 and 15 if the algorithm is coded in C. The average time for the block's transfer between the on-chip memory and the on-board SSRAM in our system is around $4T_{mem}(n_i, n_n)$.

The execution time to factor the last block is given by:

$$T_{last}(n_n) = (\frac{4n_n^3 + 9n_n^2 - 13n_n}{6}) * T_f \tag{4.10}$$

B. Sequential execution time

If we assign all the DBBD matrix blocks to a single processor, then the n-1 independent 3-block groups will be processed in time:

$$T_{seq,3blocks} = \sum_{i=1}^{n-1} \{T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + T_{add}(n_n) + 2T_{mem}(n_i, n_n)\} \tag{4.11}$$

The remaining work is the factorization of the last block that takes time $T_{last}(n_n)$. If we also consider the startup time, T_{start}, taken by the host to send the matrix and application code to the memory and the time T_{end} to collect the factored data, the sequential execution time to factor the entire DBBD matrix is given by:

$$T_{seq} = T_{start} + T_{seq,3blocks} + T_{last}(nn) + T_{end} \tag{4.12}$$

C. Parallel solution with static scheduling

The worst case scenario for a parallel solution appears when all p processors finish their work on n-2 independent 3-block groups at the same time and just one 3-block group is

left at the end (the smallest block according to our scheduling policy). The time spent on the *n-2* 3-block groups is:

$$\frac{1}{p} * \{\sum_{i=1}^{n-2} \{T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + T_{add}(n_n) + 2T_{mem}(n_i, n_n)\} \tag{4.13}$$

The last 3-block group will be handled by a single processor in time:

$$T_{lu3}(n_{n-1}, n_n) + T_{mul}(n_{n-1}, n_n) + T_{add}(n_n) + 2T_{mem}(n_{n-1}, n_n) \tag{4.14}$$

After finishing with all the 3-block groups, every PE adds the partial sums of its two children and writes the result back into the memory to be accessed by its parent in the next step. The collection of the partial sums along the binary tree of height, $h = \lceil \{\log_2(p+1) - 1\} \rceil$, takes time:

$$\sum_{i=1}^{h} [2T_{add}(n_n) + 3T_{mem}(n_n)] \tag{4.15}$$

The last block is factored by three neighbors after the summation of the partial sums. Since the three neighbors share memory, we save on computation time but not on communication time: $\frac{1}{3} T_{last}(n_n) + 2T_{mem}(n_n)$.

Thus, the worst case time required with static scheduling is:

$$\begin{aligned} T_{static} = T_{start} + \frac{1}{p} * \{\sum_{i=1}^{n-2} [T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + T_{add}(n_n) + \\ 2T_{mem}(n_i, n_n)]\} + \{T_{lu3}(n_{n-1}, n_n) + T_{mul}(n_{n-1}, n_n) + \\ T_{add}(n_n) + 2T_{mem}(n_{n-1}, n_n)\} + \sum_{i=1}^{h} [2T_{add}(n_n) + 3T_{mem}(n_n)] + \\ \{\frac{1}{3} T_{last}(n_n) + 2T_{mem}(n_n)\} + T_{end} \end{aligned} \tag{4.16}$$

D. Parallel solution with dynamic task scheduling

If we employ the proposed dynamic load balancing technique, then the work on the last 3-block group in the worst case will be performed by three neighboring processors; all the other p-3 processors will be normally working on the parallel summation of partial results at that time. Eq. (4.14) is then replaced by:

$$\frac{1}{3}\{T_{lu3}(n_{n-1}, n_n) + T_{mul}(n_{n-1}, n_n) + T_{add}(n_n)\} + 2T_{mem}(n_{n-1}, n_n) \tag{4.17}$$

So, the total time is reduced to:

$$\begin{aligned} T_{dyn} = T_{start} + \frac{1}{p} * \{ & \sum_{i=1}^{n-2} [T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + T_{add}(nn) + \\ & 2T_{mem}(n_i, n_n)] + (n-2) * (T_{add}(n_n) + 2T_{mem}(nn, nn))\} + \\ & \max\{[\frac{1}{3}(T_{lu3}(n_{n-1}, n_n) + T_{mul}(n_{n-1}, n_n) + T_{add}(n_n)) + \\ & 2T_{mem}(n_i, n_n)], \sum_{i=1}^{h} [2T_{add}(n_n) + 3T_{mem}(n_n)]\} + \{\frac{1}{3}T_{last}(nn) + 2T_{mem}(nn)\} + T_{end} \end{aligned} \tag{4.18}$$

The upper bound on the speedup, $\frac{T_{seq}}{T_{dyn}}$, compared to the sequential solution, is a complex function of $\{n_i, n_n, p, T_f, T_{clk}\}$. Since the factorization and multiplication have complexity $O(n^3)$, this algorithm will have good performance with a large number of processors, as shown by Eq. (4.18). The last block in the DBBD matrix is usually dense after combining all the partial products. The factorization of the last block dominates the execution time. The size of the last block also has a big impact on the factorization and multiplication times, as demonstrated by Eqs. (4.6) and (4.7). Therefore, we should try to make the last block as small as possible. On the other hand, the minimum size of

the last block largely depends on the physical characteristics of the original matrix. With more independent 3-block groups, we may have fewer floating-point operations and increased times for the summation of partial products and data communication. Also, the size of the last block increases with more independent blocks. With fewer such blocks, however, the total number of fill-ins is normally larger and this increases the total factorization time.

4.2.5 Dynamic Mixed-Mode Scheduling on HERA

The parallel LU factorization of sparse DBBD matrices involves irregular computation patterns and blocks of various sizes, as the result of the physical characteristics of the original matrices. However, many parts in the algorithm could still benefit from an SIMD implementation. As a natural consequence, a combination of appropriate parallel execution modes should give better results.

For this application, our HERA machine comprises 36 PEs mapped to a 6 x 6 mesh. To map an application algorithm onto a mixed-mode system, the main focus is on identifying the optimal mode of parallelism for each subtask. We should also take into account the costs incurred when switching between different pairs of modes: SIMD/MIMD, SIMD/M-SIMD and MIMD/M-SIMD. The following is the general scheduling procedure to carry out the parallel LU factorization of DBBD matrices on our mixed-mode machine.

Step 1 (SIMD & M-SIMD) Identify 3-block groups of comparable size and put them into different task queues. Divide and configure the system into M-SIMD based on the task information. Assign 3-block groups from each queue to the PEs working in the

same SIMD group, and perform the *FAC* and *MAC* work on these groups until the number of remaining 3-block groups is less than the number of PEs (i.e., 36).

Step 2 (M-SIMD) Assign the remaining 3-block groups so that groups of comparable size go to the same column of PEs (see Figure 3.3) and every PE has the largest possible number of idle nearest neighbors. This is an effort to facilitate the subsequent *PAC* work. If necessary, reconfigure the system into a different M-SIMD layout.

Step 3 (M-SIMD & MIMD) A PE is reconfigured into MIMD as soon as it finishes its work and no more 3-block group is waiting in the task queue.

Step 4 (M-SIMD & MIMD) Assign each PE in MIMD to the multiplication of a pair of (row and column) factored border blocks. Since the LDM has a shared port with its east and south neighbors, every idle PE will help its neighbors after it finishes its own work; no data transfer incurs in this process.

Step 5 (SIMD) After the factorization of all the 3-block groups and the multiplication of factored border blocks, reconfigure all the PEs again into the SIMD mode to carry out the *PAC* work.

Step 6 (SIMD) Factor the last block in the SIMD mode.

Figure 4.5 shows a typical PE mode assignment in the above procedure for large DBBD matrices. When the number of tasks in one or more task queues is larger than 36, we begin with one or more single SIMD configurations, which is a special case of M-SIMD in Step 1.

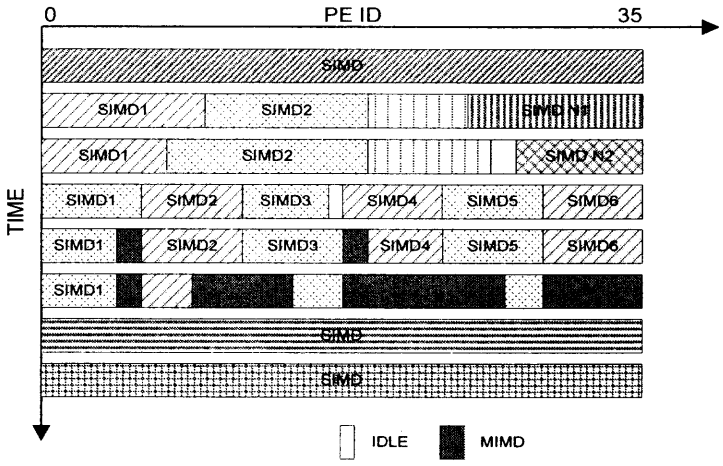


Figure 4.5 Typical PE mode assignment for large DBBD matrices.

4.3 Parallel Direct Solution of Sparse Linear Equations

Once L and U are determined, then the equations in the form $Ax = b$ can be written as two triangular systems, $Ly = b$ and $Ux = y$, whose solutions can be obtained by forward reduction and backward substitution, respectively.

The factored LU matrix produced by this algorithm is in the DBBD format. This format shows inherent parallelism in the forward reduction and backward substitution phases. In forward reduction, the following equation is used:

$$y_i = b_i - \sum_{j=1}^{i-1} (l_{ij} * y_j) \qquad \text{for } i = 1, \ldots, N \tag{4.19}$$

where l_{ij} stands for L_{ij}. If the matrix blocks are distributed among the processors in the increasing processor-address, row-number orders, communication is required to transfer the results in the y vector to the processor with the next higher address before the latter begins its work. However, except for the diagonal blocks in the sparse DBBD matrix, all matrix blocks in L of Eq. (4.2) contain all zeros (see Eq. 4.3), so no communication is required between processors. Therefore, solving for the values in the y vector

corresponding to the independent diagonal blocks can be carried out in parallel, except for the last block that requires all the solved data of L and the values in the y vector from all the processors with lower addresses. We let every processor generate the partial sums after it finds the unknowns in y, which are then accumulated for the last processor by employing a binary tree of processors configuration. The procedure is as follows. (1) All processors operate in parallel to solve the part of the y vector assigned to them, using their assigned diagonal blocks in matrix L and vector B. (2) All processors perform matrix-by-vector operations in parallel involving their lower border block and the corresponding solved block in the y vector. (3) Partial results are accumulated in parallel by all processors so they can be used in the next step to obtain the solutions in the last diagonal block. (4) Finally, forward reduction is carried out in the last diagonal block by the processor with the highest address. (Parallel processing could be applied in this stage as well).

The equation for backward substitution is

$$x_i = y_i - \sum_{j=i+1}^{N} (u_{ij} * x_j) \qquad \text{for } i = N, \ldots, 1 \tag{4.20}$$

where u_{ij} stands for U_{ij}. In our DBBD parallel algorithm, we start backward substitution in the last block involving the processor with the highest address. After the solutions are obtained for the last block, this processor broadcasts its solved block for x to all the other processors. Finally, all the processors find the solutions in parallel for their assigned block in the x vector.

4.4 Parallel Solution of Newton's Power Flow Equations

The real-time solution of the AC power flow problem is a critical and fundamental task in power system planning and operations. An efficient power flow solution can also improve the performance of other relevant problems, such as those associated with transient stability. Among the vast number of power flow solutions, two major categories of solvers have been thoroughly investigated and widely employed by the power research and industry communities: Newton's method [Tinney, et al., 1967] and the Fast Decoupled Power Flow (FDPF) method [Stott, et al., 1974]. Newton's method solves repeatedly simultaneous, large, sparse, linear systems of equations. The LU factorization of the Jacobian matrix at each iteration in the direct solution of the linear equations has been a great challenge to the computation capability and memory capacity of available computing platforms [IEEE, 1992]; this is the main reason that the employment of an exact Newton's method is often avoided, especially when the computation is carried out in real time and/or involves very large power systems. On the other hand, FDPF algorithms require LU factorization only once and the result is repeatedly used throughout the entire power flow analysis process by employing a fixed and smaller coefficient matrix. Thus, the solution time can be reduced dramatically. Although many improvements can make FDPF more robust, in some cases where the coefficient matrices are ill-conditioned FDPF has convergence difficulties even with the application of good pre-conditioners. The conventional Newton's method is still commonly employed by the power industry.

To overcome the heavy computation demands caused by LU factorization in Newton's method, parallel computing techniques may be applied [IEEE, 1992; Falcao,

et al., 1996]. More specifically, researchers have realized the importance of selecting appropriate parallel architectures for high performance. However, parallel computing in power engineering has not been widely accepted by the industry due to the scarce availability of low-cost, high-performance parallel machines suitable for these tasks [IEEE, 1992; Falcao, et al., 1996]. Although parallel computers have been successful in solving several computation-intensive problems, their high price and long design and development cycles, and the high cost of maintaining them often make their long term availability unpredictable [Bell, et al., 2002]. Moreover, the efficient parallelization of the exact Newton's method has proved to be a Herculean task and few good speedups have been reported in the literature. PC clusters have emerged in recent years as a parallel-computing alternative to take advantage of the ever-increasing computing power of commercial-off-the-shelf (COTS) general-purpose microprocessors. Parallel implementations of FDPF methods on PC clusters are popular in solving the AC power flow problem due to their reduced computation and memory requirements and the high-availability of these computing platforms [Tu, et al., 2002; Chen, et al., 2005]. However, the high communication overheads present in these platforms quickly diminish the performance when increasing the system size; therefore, these implementations suffer in terms of scalability and efficiency. Details of relevant work will be presented in Section 4.3.6 after we present the details of our proposed algorithm.

Based on our parallel DBBD LU factorization algorithm introduced in the last section, we propose a novel partitioning technique for nonsymmetric Jacobian matrices and use the DBBD algorithm to solve the power flow problem in parallel with Newton's method.

4.4.1 Newton's Solution to the Power Flow Problem

The main objective of power flow analysis is to determine precise steady-state voltages (magnitudes and angles) on all buses in a given network, and then to derive from them the real and reactive power flows into every line and transformer; the network topology and all information about the generation and load lines are known. For most network buses, the active and reactive powers are specified; they can be evaluated by the following equations for a network with N buses [Grainger, et al., 1994]:

$$P_i = \sum_{k=1}^{N} | y_{ik} V_i V_k | \cos(\theta_{ik} + \delta_k - \delta_i) \tag{4.21}$$

$$Q_i = \sum_{k=1}^{N} | y_{ik} V_i V_k | \sin(\delta_i - \delta_k - \theta_{ik}) \tag{4.22}$$

where P_i, Q_i and V_i, are the active power, reactive power and complex voltage at bus i, respectively, with $V_i = |V_i| \angle \delta_i$, $V_k = |V_k| \angle \delta_k$, $y_{ik} = |y_{ik}| \angle \theta_{ik} = g_{ik} + jb_{ik}$, for $i, k \in [1, N]$; y_{ik} is an element of the bus admittance matrix (Y_{bus} matrix). If the number of voltage-controlled buses in the system is N_g, then we need to solve $(2N\text{-}N_g\text{-}2)$ equations.

Newton's method expands these equations into a Taylor series and incorporates the first-derivative information when updating the voltages. Thus, the following linear equations are produced to be solved iteratively until the mismatches $\Delta\delta$ and ΔV are smaller than a pre-specified tolerance:

$$\begin{bmatrix} J^{11} & J^{12} \\ J^{21} & J^{22} \end{bmatrix} \begin{bmatrix} \Delta\delta \\ \frac{\Delta V}{|V|} \end{bmatrix} = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} \tag{4.23}$$

The Jacobian matrix $J = \{J^{11}, J^{12}, J^{21}, J^{22}\}$ is reevaluated at each iteration by the following equations that use updated voltages:

$$\frac{\partial P_i}{\partial \delta_j} = |V_i||V_j|(g_{ij}\sin\delta_{ij} - b_{ij}\cos\delta_{ij}) \qquad j \neq i \tag{4.24}$$

$$\mathbf{J^{11}}: \quad \frac{\partial P_i}{\partial \delta_i} = -|V_i|\sum_{\substack{i=1 \\ i\neq j}}^{N}|V_j|(g_{ij}\sin\delta_{ij} - b_{ij}\cos\delta_{ij})$$

$$|V_j|\frac{\partial Pi}{\partial |Vj|} = |Vi||Vj|(gij\cos\delta ij + bij\sin\delta ij) \qquad j \neq i \tag{4.25}$$

$$\mathbf{J^{12}}: \quad |Vi|\frac{\partial Pi}{\partial |Vi|} = 2|V_i|^2\, gii + |Vi|\sum_{\substack{i=1 \\ i\neq j}}^{N}|Vj|(gij\cos\delta ij + bij\sin\delta ij)$$

$$\frac{\partial Q_i}{\partial \delta_j} = -|V_i||V_j|(g_{ij}\cos\delta_{ij} + b_{ij}\sin\delta_{ij}) \qquad j \neq i \tag{4.26}$$

$$\mathbf{J^{21}}: \quad \frac{\partial Q_i}{\partial \delta_i} = |Vi|\sum_{\substack{i=1 \\ i\neq j}}^{N}|V_j|(g_{ij}\cos\delta_{ij} + b_{ij}\sin\delta_{ij})$$

$$|V_i|\frac{\partial Q_i}{\partial |V_j|} = |V_i||V_j|(g_{ij}\sin\delta_{ij} - b_{ij}\cos\delta_{ij}) \qquad j \neq i \tag{4.27}$$

$$\mathbf{J^{22}}: \quad |V_i|\frac{\partial Q_i}{\partial |V_i|} = |V_i|\sum_{\substack{i=1 \\ i\neq j}}^{N}|V_j|(g_{ij}\sin\delta_{ij} - b_{ij}\cos\delta_{ij}) - 2V_i^2 b_{ii}$$

In order to derive the mismatches at each iteration from the above linear equations, two kinds of methods are usually employed: direct and iterative methods [Grainger, et al., 1994]. LU factorization followed by forward reduction and backward substitution [Duff, 1998] is one of the most widely used direct methods to solve the linear systems. Then, Eq. (4.23) can be solved by the following two sets of equations:

$$Lw = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} \tag{4.28}$$

$$U\begin{bmatrix} \Delta\delta \\ \dfrac{\Delta V}{|V|} \end{bmatrix} = w \tag{4.29}$$

Direct methods are usually more robust and efficient for larger matrices, but it may be difficult to extract substantial parallelism while maintaining a low inter-

processor communication overhead. Moreover, the typical computation complexity of LU factorization is $O(M^3)$, where M x M is the matrix size (i.e., the size of the Jacobian matrix in our case). In Newton's method, the LU factorization is applied on a new Jacobian matrix at each iteration. The repetitive solution of the linear equations in Newton's method is very time-consuming for large networks, if the problem is solved sequentially.

There have been many attempts to develop parallel algorithms optimized for different parallel architectures for efficiently solving the power flow problem. Excellent reviews of high-performance computing efforts in power engineering appeared in [IEEE, 1992; Falcao, et al., 1996]. Our parallel approach stems from the fact that the Y_{bus} and Jacobian matrices are usually very sparse, especially for large networks. Table 4.2 shows the sparsity (percentage of non-zero elements in a matrix) in the benchmark matrices used in experiement. For networks with thousands of buses, the typical number of nonzero elements per row is less than four. Although the LU factorization of sparse matrices potentially has fewer operations than that of dense matrices, it can suffer tremendously from dynamic fill-ins. As we discussed in the Introduction, network partitioning is a promising approach to reduce the number of operations applied to power matrices. The basic idea behind such an approach is to divide an interconnected network into independent sub-networks and a collection of cutting nodes. The DBBD form is obtained by reordering a given sparse matrix based on network partitioning. This way, LU factorization can first be applied to completely independent sub-networks in parallel, thus speeding up the algorithm dramatically. The information corresponding to the cutting nodes is then processed at a lower rate. Details follow in Section 4.4.3.

Table 4.2 Sparsity of the Benchmark Matrices for Power Flow Analysis

Systems	57	118	300	1648	7917
Branches	80	186	411	2516	12147
Dimensionality of the Jacobian matrix	106	181	530	2982	14508
% of non-zeros in the Y_{bus} matrix	6.56	3.42	1.24	0.246	0.0514
% of non-zeros in the J matrix	6.40	3.21	1.33	0.244	0.0513

4.4.2 Parallel LU Factorization of Jacobian Matrices

A. Network Partitioning

The sparse Y_{bus} matrix can be reordered into the DBBD form shown in Figure 4.6 by the node tearing technique or other similar heuristics-based algorithm. $\{Y_{ii}, Y_{in}, Y_{ni}\}$, *for* $i \in [1, n\text{-}1]$, are matrix blocks representing a sub-network; for a given value of i, this will be referred to as a 3-block group. Y_{nn} represents cutting nodes and is referred as the last block. Let N_c be the number of cutting nodes, i.e., the size of Y_{nn} is N_c x N_c. All other blocks contain all zeros. The maximum number of nodes in each diagonal block and, hence, the sparsity of the blocks can be tuned by a user parameter, *MaxNodes,* during network partitioning. Generally speaking, the more diagonal blocks (sub-networks) in the DBBD matrix, the denser the blocks are and the larger the last block Y_{nn} is. Different orderings may result in big differences in the total solution time of the equations. In our implementation of the reordering technique, we seek an ordering with a large number of diagonal blocks but not a too large Y_{nn}. This objective is justified in Subsection 4.4.3.

$$\begin{pmatrix} Y_{11} & 0 & \dots & 0 & Y_{1n} \\ 0 & Y_{22} & \dots & 0 & Y_{2n} \\ \vdots & 0 & \dots & 0 & \vdots \\ 0 & 0 & \dots & Y_{n-1,n-1} & Y_{n-1,n} \\ Y_{n1} & Y_{n2} & \dots & Y_{n,n-1} & Y_{nn} \end{pmatrix}$$

Figure 4.6 Sparse DBBD Y_{bus} matrix.

The node tearing technique assumes that the matrix is symmetric whereas Newton's method requires employs a nonsymmetric Jacobian matrix as the coefficient matrix. To obtain the DBBD form for the Jacobian matrix, we first examine Eqs. (4.24)-(4.27) to produce J^{ik} (i, k = 1, 2). We observe that every element in J^{ik} is directly related to the corresponding element in the Y_{bus} matrix; the zero elements in the Y_{bus} matrix cause the corresponding elements in each of the four quadrants in the Jacobian matrix to be zero. This reveals a structural similarity involving non-zero elements in the J^{ik} and Y_{bus} matrices. After we order the Y_{bus} matrix into the DBBD form, the corresponding Jacobian matrix should have the form shown in Figure 4.7 (a).

In this figure, $\{J_{i\_d11}, J_{i\_u11}, J_{i\_b11}$, for $i \in [1, n\text{-}1]\}$, J_{n_d11} and along with all the other zero elements in the corresponding quadrant constitute J^{11} in Eq. (4.23), whereas $\{J_{i\_d12}, J_{i\_u12}, J_{i\_b12}\}$ and J_{n_d12} are in J^{12}, and so on. The sizes of the diagonal blocks in the four quadrants are shown in Table 4.3. S_{yi} is the size of the i^{th} diagonal block in the Y_{bus} matrix and N_{gi} is the number of PV buses that appear in the i^{th} diagonal block of the Y_{bus} matrix. $N_g = \sum_{i=1}^{n} N_{gi}$ is the total number of PV buses in the system.

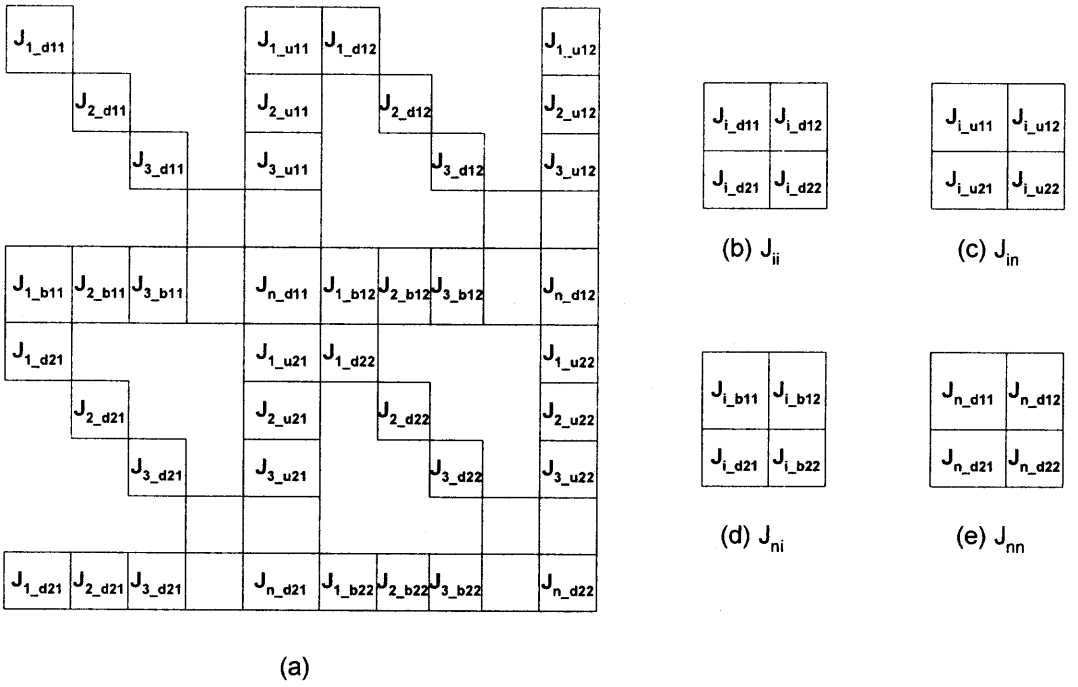
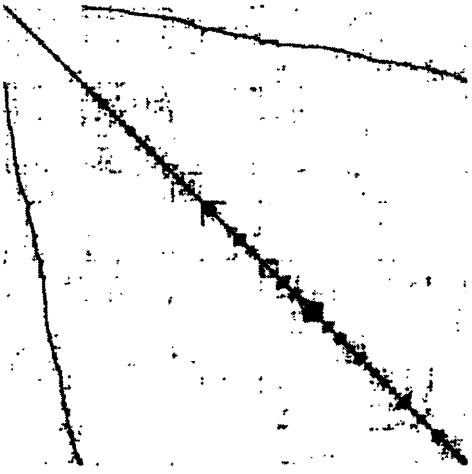


Figure 4.7 The Jacobian matrix produced from the DBBD Y_{bus} matrix.

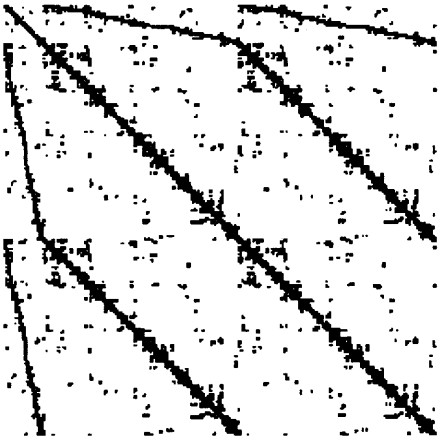
If we permute the Jacobian matrix shown in Figure 4.7 (a) in such a way that the four blocks related to the same i^{th} diagonal block in the Y_{bus} matrix are grouped together, we can find that the Jacobian matrix can also be represented in the DBBD form, with the i^{th} diagonal block being of size $(2S_{yi} - N_{gi})$. The blocks $\{J_{i\_d11}, J_{i\_d12}, J_{i\_d21}, J_{i\_d22}\}$, $\{J_{i\_u11}, J_{i\_u12}, J_{i\_u21}, J_{i\_u22}\}$, and $\{J_{i\_b11}, J_{i\_b12}, J_{i\_b21}, J_{i\_b22}\}$, for $i \in [1, n\text{-}1]$, form the new 3-block groups $\{J_{ii}, J_{in}, J_{ni}\}$ in the DBBD Jacobian matrix (shown in Figs. 4.7 (b), (c) and (d)), and $J_{nn} = \{J_{n\_d11}, J_{n\_d12}, J_{n\_d21}, J_{n\_d22}\}$ (shown in Figure 4.7 (e)) becomes the new last block in the DBBD Jacobian matrix. Figure 4.8 shows the nonzero elements in different matrices for the 7917-bus system.

Table 4.3 The Sizes of the Blocks in the Jacobian Matrix

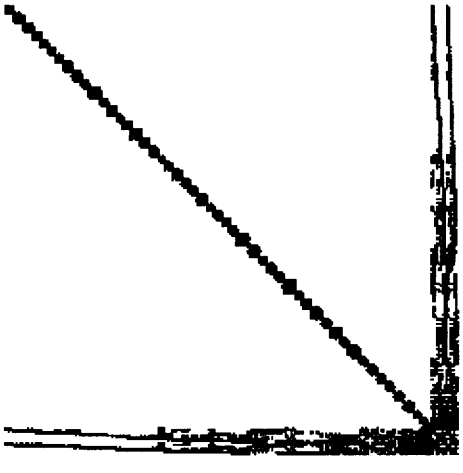
Block	J_{i_d11}	J_{i_d12}	J_{i_d21}	J_{i_d22}
Rows	S_{yi}	S_{yi}	S_{yi}-N_{gi}	S_{yi}-N_{gi}
Columns	S_{yi}	S_{yi}-N_{gi}	S_{yi}	S_{yi}-N_{gi}



(a) Original Y_{bus} matrix



(b) Original Jacobian matrix



(c) DBBD Jacobian matrix

Figure 4.8 Nonzero elements for the 7917-bus system.

B. Parallel LU Factorization of the DBBD Jacobian Matrix

After we order the Jacobian matrix into the DBBD form, Eq. (4.23) can be solved by the parallel DBBD LU factorization algorithm described in Section 4.2, which is then followed by parallel forward reduction and backward substitution described in Section 4.3. The calculations of L_{kk}, U_{kk}, L_{nk} and U_{kn} for different k's (i.e., 3-block groups) are independent of each other. So we can distribute different 3-block groups to different processors to be factored in parallel, with no data exchanges until the factorization of J_{nn}. This is the reason that we try to maximize the number of diagonal blocks in the matrix reordering procedure. The last block, J_{nn}, requires data produced in all the right and bottom border blocks, so its factorization is the last step. Before factoring the last block, pairs of the already factored border blocks are first multiplied in parallel to produce $J_{nn}^{k} = L_{nk}U_{kn}$, for $k \in [1, n\text{-}1]$. The summation of the n-1 products obtained for the different values of k and the addition of its results to J_{nn} is then carried out in a binary tree fashion in parallel and the results are sent to the processors assigned to the factorization of the last diagonal block (for a highly parallel approach). The last block becomes denser after all the partial results are applied to it and its factorization require a significant amount of time. Thus, the most computation-intensive part of the entire problem can be solved efficiently in parallel. Another advantage of the DBBD ordering is that the results can be used repeatedly for different values of the right hand side in the linear system and the 3-block groups assigned to each processor remain the same as long as the network topology does not change.

4.4.3 Parallel Solution of Newton's Power Flow Equations

We summarize here our parallel DBBD Newton algorithm for power flow analysis based on the above parallel LU factorization approach.

Step 1. Use the heuristics-based node tearing algorithm to reorder the Y_{bus} matrix into the DBBD form and sort the 3-block groups $\{Y_{ii}, Y_{in}$ and Y_{ni}, for $i \in [1, n-1]\}$according to their computation cost (by estimating the number of floating-point operations based on the number of nonzero elements); try to get the best partitioning results according to our rules discussed earlier. Then, renumber the buses in the original network file according to the selected partitions so that the ordered DBBD matrix has continuous bus numbering; the purpose of the renumbering is to speedup the remaining steps. This preprocessing step is performed on a PC.

Step 2. Assign the relevant bus data along with the 3-block group(s) in the Y_{bus} matrix to the processors. The bus data for the cutting nodes is copied into every processor in order to subsequently minimize the number of data collisions during processor communication.

Step 3. Initialize in parallel the voltages to a flat voltage start.

Step 4. Evaluate Eqs (4.21) and (4.22) and calculate $\triangle P$and $\triangle Q$ in (4.23) in parallel using 3-block groups, bus data and voltages (they are updated at every iteration). Then, every processor checks to see if all $\triangle P$and $\triangle Q$ in its assigned bus range are sufficiently small (we set the tolerance at 0.001p.u.); it sends its decision to a control processor, which may stop the iterative procedure based on the information reported by all the computing processors.

Step 5. Form in parallel the 3-block groups $\{J_{ii}, J_{in}$ and $J_{ni}\}$ in the DBBD Jacobian matrix. In this step, every processor uses the data assigned to it in Step 1 and performs calculations on its assigned 3-block groups. J_{nn} is processed by the control processor.

Step 6. Apply in parallel LU factorization to the 3-block groups $\{J_{ii}, J_{in}$ and $J_{ni}\}$; use the procedure described earlier. Also apply our dynamic load balancing techniques during this procedure in order to increase the efficiency. We employ the control processor to monitor the load information of every processor and predict and assign jobs based on its capabilities, job costs related to computation and communication.

Step 7. Solve Eq. (4.23) in parallel by forward reduction and backward substitutions in order to derive the voltage corrections $\Delta\delta$ and ΔV. Then, apply these corrections to the current voltage values.

Step 8. Go back to Step 4.

From this description, we can see that the most time-consuming steps are carried out in parallel and little inter-processor communication is needed.

4.4.4 Relevance to Other Work

We focus here on a comparison with other DBBD-related parallel algorithms. Several parallel implementations of the power flow analysis problem by FDPF based on DBBD partitioning have been reported, where the target matrices for LU factorization are symmetric and smaller than the corresponding Jacobian matrices [Tu, et al., 2002; Chen, et al., 2005; Koester, et al., 1994]; In contrast, the Jacobian matrices in our work

are nonsymmetric. To the best of our knowledge, we have not found any important literature about the parallel solution of exact Newton's method with parallel DBBD LU factorization. In the former papers, the total number of independent blocks in the DBBD Y_{bus} bus matrix (i.e., n-1) is limited by the number of available processors, which is normally small. For large power systems with thousands of buses, the resulting 3-block groups are still very sparse. It is also very difficult in this situation to balance the processor work loads. Since these approaches often target PC clusters or other loosely coupled systems with high communication latencies, more independent blocks will incur more communication overhead at the end of the procedure, thus limiting performance. Good performance is only observed for a small number of processors.

Since all the processors in our system are embedded into a single chip and we also employ an architecture optimized for the application, our system presents very low communication costs to our algorithm. The system architecture can even be changed at run time, as needed, to match the dynamically changing characteristics of the application by taking advantage of the reconfigurability of FPGAs. Another major contribution of our approach is the application of our dynamic load balancing techniques in the parallel LU factorization of large DBBD Jacobian matrices where load imbalance is a major bottleneck that can inadvertently affect the performance [Chai, et al., 1993].

CHAPTER 5

PERFORMANCE RESULTS AND ANALYSIS

5.1 Mixed-Mode Scheduling of Matrix-Matrix Multiplication on HERA

We first implemented on HERA with 64 PEs our mixed-mode MMM scheduling for square matrices of size up to 1000 x 1000. The execution times on HERA are presented in Figure 5.1. Previous MMM work on FPGAs targeted fixed-point data and the floating-point performance on platform FPGAs in [Zhuo, et al., 2004] was shown for a small 8 x 8 matrix, therefore we cannot compare HERA's performance with other related work on FPGAs. We implemented instead block-based MMM in C code on two commercial PCs; comparative results are also shown in Figure 5.1. The block-based MMM code for the Dell PCs was optimized by several techniques, such as using best block sizes for the L1 and L2 caches, compiler flags and copy optimization. We can see that our results on HERA are better than those on the dual-Xeon 2.66GHz and the uni-Pentium IV 2GHz systems despite HERA's much lower clock frequency (i.e. 125MHz). In fact, the relative speedup on HERA improves further for larger matrices. The performance of HERA shown here is not significantly faster than that of a conventional microprocessor because we used entry-level FPGA devices in our current implementation. However, we expect dramatic performance gains in the near future by employing more advanced FPGAs. A testimony to this effect appeared in [Underwood, 2005] that shows the performance of FPGAs in floating-point operations to be growing at a much faster rate than that of microprocessors; it surpassed the latter in 2003-2004, which agrees with our results.

The speedup of the parallel implementation on the 64-PE HERA over the sequential one on the 1-PE HERA is shown in Figure 5.2. The speedup for the 100 x 100 case is much lower because the ratio of computation to communication times is much lower than for the other cases. We could improve the speedup further in all cases by using a bigger local memory since the complexity of multiplication on a single PE for a pair of blocks is $O(N^3)$ and that of communication (i.e. shifting) is $O(N^2)$, where N x N is the block size. With increases in the problem size, the speedup and, of course, the efficiency stabilizes in a very narrow range. We also evaluated the performance of our mixed-mode scheduling for a variety of non-square matrices. The multiplication of irregular matrices is required in the parallel LU factorization of sparse DBBD matrices. SIMD mappings, where all the PEs work in the SIMD mode all the time were also implemented for these matrices; the results are shown in Table 5.1. From this table, we can see that dynamic mixed-mode scheduling can greatly boost performance.

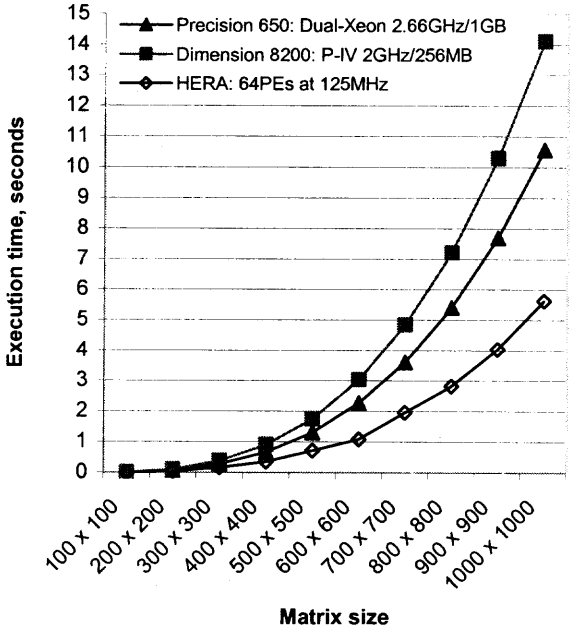


Figure 5.1 Performance comparison of MMM on HERA and two Dell PCs (optimized code was run on all the machines).

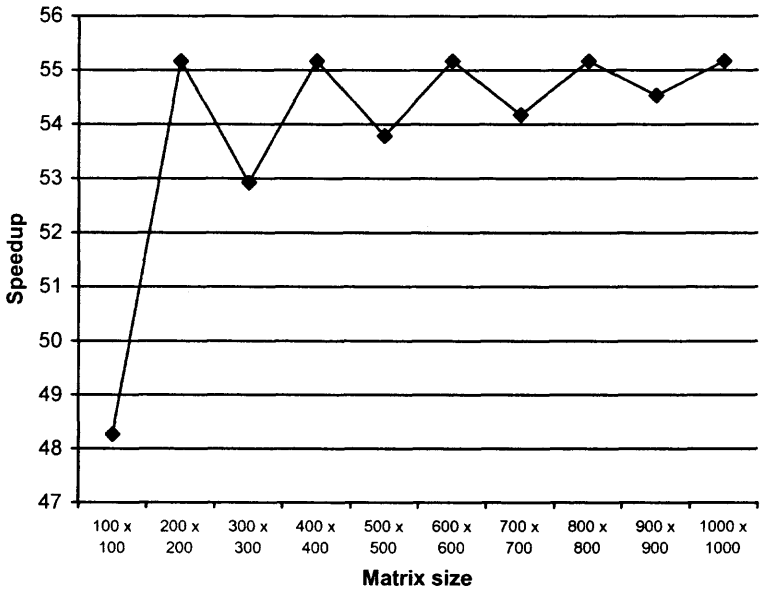


Figure 5.2 HERA speedup of parallel over uni-PE execution.

Table 5.1 HERA Execution Times for Irregular Matrices under Different Execution Modes
(Clock frequency: 125MHz)

Matrix Dimensions			HERA in SIMD mode, sec	HERA in mixed-mode, sec	Improvement %
N1	N2	N3			
105	101	113	0.0115	0.0103	10.1
201	215	323	0.0864	0.0723	16.3
324	599	315	0.3699	0.3366	9.8
05	611	613	0.9254	0.7853	15.1
509	301	201	0.2163	0.1894	12.4
677	202	677	0.5037	0.4749	5.7
711	713	403	1.3344	1.1584	13.2
955	957	976	5.6077	5.1872	7.5

5.2 Parallel LU Factorization of Sparse Matrices on CG-MPoPC

The main objective of this set of experiments was to evaluate any performance gains when customizing the architecture, interconnection network and memory hierarchy of

the MPoPC. Several benchmark matrices from the Harwell-Boeing Collection in the Matrix Market [MATRIX] and the U. S. northeastern power grid were used.

5.2.1 MPoPC Customization and Configuration

Given an FPGA device, we first analyze and profile the application tasks and evaluate different system configurations, including PE functionality and interconnection network, in order to find the best solution for each task. Different hardware configurations may result in different numbers of PEs in the MPoPC system. An FPGA configuration image is then generated at static time for each task and is loaded at runtime as needed. Since we utilize configurable and extensible processors instead of custom processors, different system configurations can be designed, built, and evaluated very quickly.

A System Controller (SC) is implemented with each MPoPC configuration since we are currently using a PC to download MPoPC configuration and application data. The SC has access to the local memories of every PE, as shown in Figure 5.3. Avalon [ALTERA] is a multi-mastering non-shared bus which is used to connect all the local data memories of the PEs to the SC. All the PEs share the on-board SDRAM memory and a memory controller is included in the SC. The on-board memory keeps all the intermediate results when reconfiguring the FPGA for each task. As per Section 4.2, DBBD-based parallel LU factorization consists of four categories of mega-tasks having different computation operations and communication patterns. For each task the MPoPC configuration is determined using two rules: (1) Choose the smallest base Nios processor for the PEs since FP operations are implemented in hardware. (2) Maximize

the local instruction and data memories of each PE as allowed by the available on-chip FPGA memory resources.

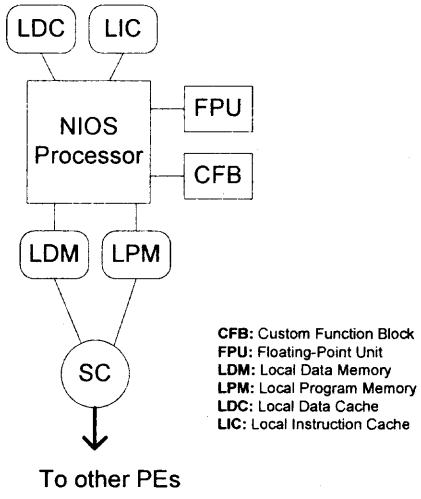


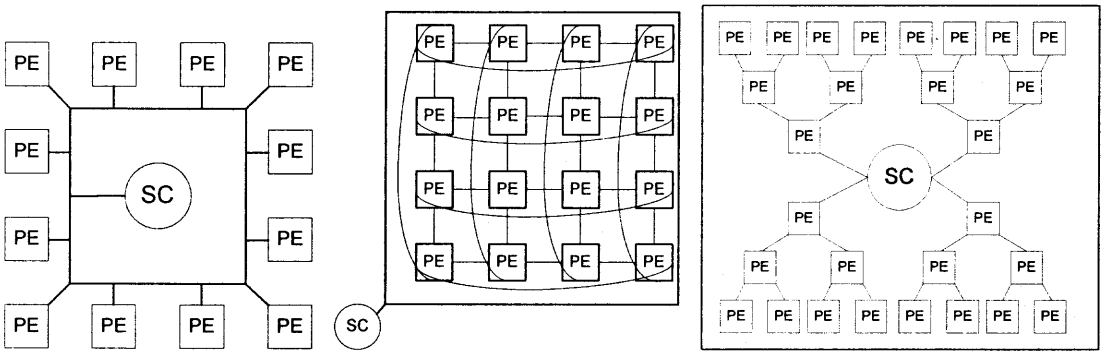
Figure 5.3 PE and SC connectivity.

The chosen configuration for each mega-task is as follows.

- ***FAC:*** All four FP operations (i.e., +, -, * and /) are required. Hence, each PE contains a complete FPU. Since no communication is required between the PEs, the interconnection is dramatically simplified. The SC takes care of data I/O and communications with the host processor. The system is shown in Figure 5.4.a. The local data memory of each PE is shared with its three neighbors, as shown in Figure 5.5.a, employing different access priorities.
- ***MAC:*** Only addition and multiplication are needed. So no hardware FP support for the other two operations is implemented in the PEs. This saves dramatically on hardware resources and results in an increased number of PEs since typically a FP divider takes more than twice the resources needed by an FP adder. A torus network is chosen in this phase as shown in Figure 5.4.b. The local data memory of each PE

is shared with its three neighbors, as shown in Figure 5.5.b, employing different access priorities.

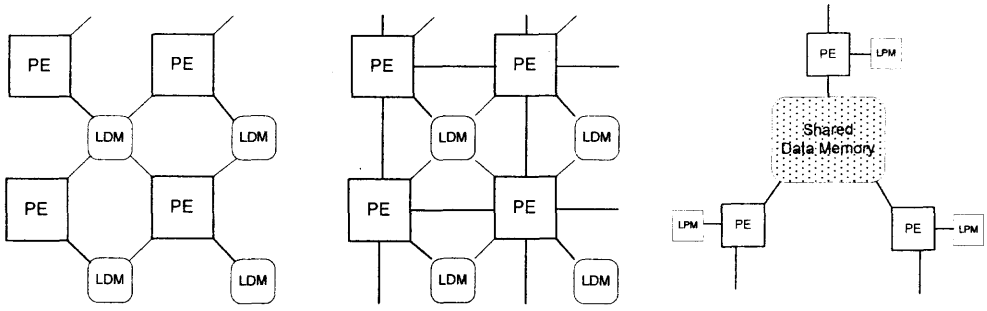
- ***PAC:*** Only an FP adder is included in each PE and a multi-tree network (shown in Figure 5.4.c) is efficient for this mega-task. Three neighboring PEs share the same data memory as shown in Figure 5.5.c.



(a) *FAC* (b) *MAC & LAST* (c) *PAC*

Figure 5.4 MPoPC configurations for the tasks in DBBD-based parallel LU factorization.
Note: The actual number of PEs in an MPoPC depends on the given application and the FPGA device.

- ***LAST:*** This is the bottleneck of the entire application, especially for large matrices usually having a large last block (A_{nn}); it becomes much denser just before LU decomposition is applied to it. We still use a torus-connected MPoPC (Figure 5.4.b and Figure 5.5.b) for this mega-task. A full FPU is included in each PE. A block-based parallel algorithm is used for large matrices with a big last block [Grama, et al., 2003]. The sharing of the data memory among neighbors reduces the communication overhead.



(a) *FAC* (b) *MAC* and *LAST* (c) *PAC*

Figure 5.5 Interconnecting on-chip data memories for the MPoPC configurations of Figure 5.4

5.2.2 Experiments and Analysis

The Altera SOPC development board with an EP20K1500EBC652-1x APEX20KE FPGA was used in our experiments. Although this is a relatively old FPGA with limited resources and speed (it was released in 2000), it can serve our purpose here. The Stratix II EP2S180device [Altera] with 9,383,040 bits of on-chip memory and 384 hardware multipliers can accommodate 23 copies of our processor with a system frequency of more than 135MHz. Our design clocks the board at 50MHz. A single-precision (32-bit) IEEE 754 pipelined FPU was developed and implemented; it runs at 128.3MHz for the 3-stage adder/subtractor of 671 LEs, 150.8MHz for the 5-stage multiplier of 785 LEs and 165.4MHz for the 28-stage divider of 2508 LEs. All the programs are implemented in assembly language and are stored entirely in the on-chip program memory.

Since the chosen matrix partitioning approach can have a tremendous impact on the execution time of parallel LU factorization, we first simulated the parallel LU factorization of DBBD matrices for a wide range of matrix partitioning results; this way, we produced a near-optimal partitioning for each power system. For example, Figure 5.6(a) shows the general trend in the number of cutting nodes (N_c) and number of

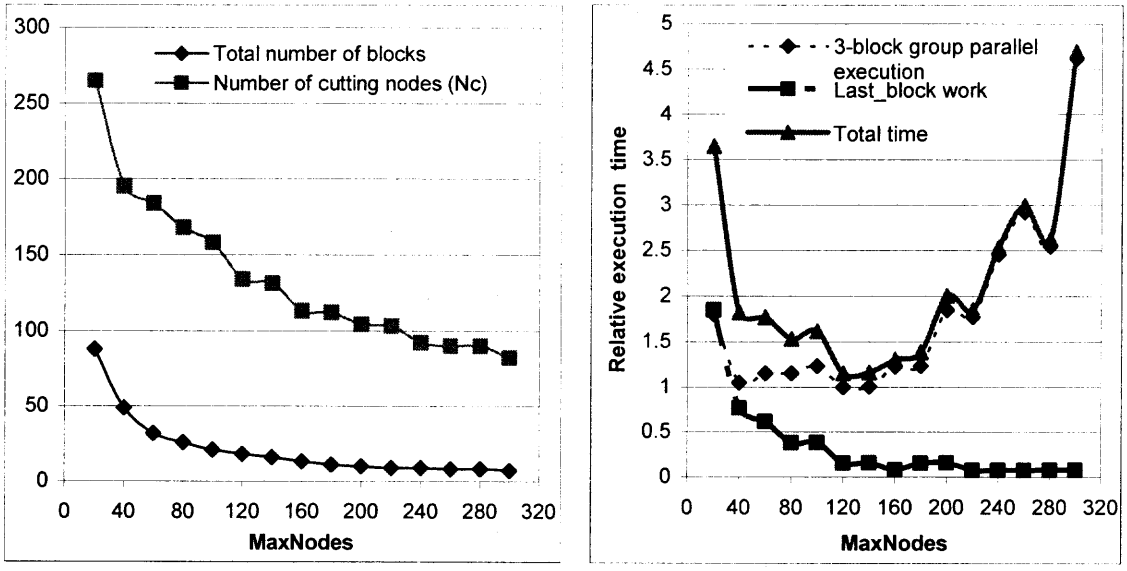
independent diagonal blocks (n-1) produced when increasing *MaxNodes* (the maximum number of nodes allowed in each sub-network) for a 2852 x 2852 matrix (a Jacobian matrix of the 1648-bus system). The relative execution time of parallel LU factorization based on different partitioning results in Figure 5.6(a) is shown in Figure 5.6(b). It is clear that the choice made in the network partitioning phase can have a big impact on the LU factorization and equation solution times. Table 5.2 shows the results of near-optimal partitioning for the benchmark systems. The last matrix corresponds to a power network in North America. The near-optimal partitioning results highly depend on the individual characteristics of the power system. This also shows the necessity for load balancing, especially for large power systems, such as the 7917- and 10279-bus systems; they produce very irregular blocks and nonzero patterns. For most large-scale applications, such as in power and circuit simulation, the matrix sparsity normally increases with increases in the matrix size; this favors the choice of more diagonal blocks in the partitioned DBBD matrix. However, the size of the last block increases as a result of more independent 3-block groups.

The numbers of PEs implemented for the four mega-tasks are 9, 16, 21 and 9, respectively. As a result of the customization of the MPoPC configuration, we are able to increase the hardware parallelism and the utilization of hardware resources. In order to compare the performance of the customized MPoPC with a fixed architecture for this algorithm, we implemented an MPoPC with PEs interconnected via the network shown in Figure 5.4.c. The comparison of the execution times for the benchmark matrices is shown in Table 5.3. The best performance improvement is 14.05% for matrix BCSPWR09.

Table 5.2 Characteristics of the Test Matrices Ordered into the DBBD Form

Matrix	PSADMIT	PSADMIT	BCSPWR09	BCSPWR10	7917-matrix	10279-matrix
Dimensionality of admittance matrix (Y_{bus})	494	1138	1723	5300	7917	10279
Non-Zero Elements	1666	4054	6511	21842	32211	37755
Total diagonal blocks (n)	27	67	42	125	51	74
Dimension of the largest diagonal block	20	20	50	50	198	180
Dimension of the smallest diagonal block	11	4	29	30	84	55
Dimension of the last diagonal block	45	100	134	577	517	474
Distribution of block sizes*	13(20)*, 8(15), 5(12), 1(8)	22(20) 26(15), 18 (12), 1(4)	10(50),14(40), 18(30)	30(50),40(40), 55(30)	12(180), 12(150), 26(120), 1(84)	17(170-180), 5(160), 17(130~140), 34(110), 1(55)

*13(20) stands for 13 diagonal blocks of size close to 20 x 20.



(a) The effect of *MaxNodes* on network partitioning.

(b) Relative execution time of parallel LU factorization for the DBBD Jacobian matrix.

Figure 5.6 Impact of network partitioning on the execution time of parallel LU factorization for a DBBD matrix of 2582 x 2582.

We also compared the performance of run-time scheduling with that of static scheduling. The speedups for all the test matrices under the run-time and static scheduling policies are shown in Figure 5.7. Both scheduling policies show good performance that improves with increases in the matrix size. The matrix of size 5300 x 5300 has a large last block that limits the speedup. Run-time scheduling performs better in all the cases. Static scheduling cannot handle well the effect of dynamic fill-ins and renders some PEs idle during the procedure. The performance of run-time task scheduling is better for matrices with irregular distribution of block sizes, as shown for PSADMIT and the 7917-matrix. A 10.89% speedup results for the 7917-matrix.

Table 5.3 Execution Times (seconds) for the Benchmark Matrices on the two MPoPCs (with the run-time scheduling policy)

Matrix Size	Customized MPoPC	Fixed MPoPC	Improvement (%)
494 x 494	0.089	0.099	11.24
1138 x 1138	0.857	0.944	10.12
1723 x 1723	2.437	2.779	14.05
5300 x 5300	50.11	56.38	12.51
7917 x 7917	132.8	147.2	10.81

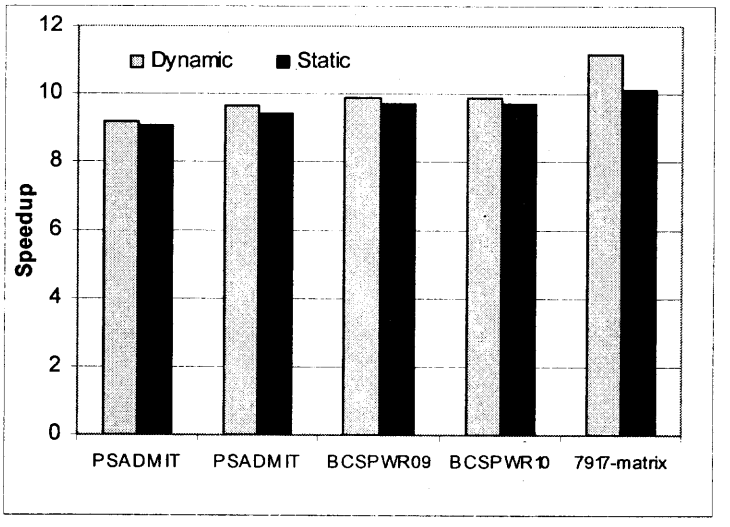


Figure 5.7 Speedup comparison of the run-time and static scheduling policies on the customized MPoPC

We further tested the performance of dynamic scheduling with that of static scheduling on systems with more PEs (without hardware FPUs for the sake of higher scalability). The speedups for the 10279-Y_{bus} matrix with up to 28 processors are shown in Figure 5.8. We chose this matrix because it is the largest one in the group and it also displays more irregularity in the location of non-zero elements. Dynamic scheduling performs better in all the cases. We observed that the performance improvement of

dynamic task scheduling further improves with increases in the number of PEs. It demonstrates a 16.4% speedup with 25 PEs.

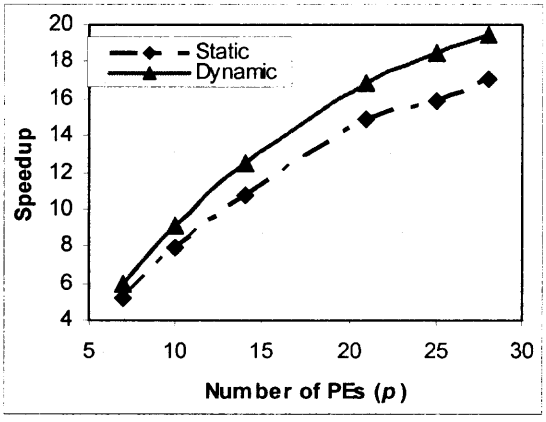


Figure 5.8 Speedup (over the uni-processor) of the static and dynamic scheduling policies for the 10279-Y_{bus} matrix. No hardware FPUs.

With an increase in p, the efficiency, $\frac{Speedup}{p}$, of the algorithm decreases because the time spent on the factorization of the last block becomes a more significant component of the entire execution time; this is shown in Eq. (4.16) and (4.18). Our experiments show that the best choice is to use three immediate neighbors sharing the same data memory to factor the last block. Figure 5.9 shows the percentage of time needed to process the last block in the 10279-Y_{bus} matrix for different numbers of PEs. The results prove that we have to make the last block as small as possible in the ordering phase. In general, however, dynamic task scheduling potentially performs better with more blocks in the DBBD matrix. For most large-scale applications, such as power applications and circuit simulations, normally the matrix sparsity increases with increases in the matrix size; this favors more diagonal blocks in the partitioned DBBD matrices. However, the size of the last block increases as a result of more independent 3-block

groups; its factorization could then diminish the speedup. So the bottleneck appears in the factorization of the last block.

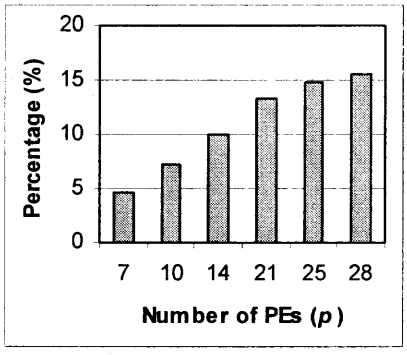


Figure 5.9 Percentage of time needed to factor the last block in the 10279-Y_{bus} matrix.

We can deduce from Table 5.2 and our theoretical analysis that the matrix of size 7917 x 7917 potentially represents the worst-case scenario. Hence, we measured the speedup for this matrix by varying the number of PEs and compared with the predicted performance from Section 4.2. We used a fixed MPoPC configuration similar to Figure 5.4.b with 9 PEs. Figure 5.10 shows the results. The measured results generally follow the predicted speedups. Any differences are due to software overheads and some simplifications made in the analysis. In general, however, run-time task scheduling potentially performs better with more blocks in the DBBD matrix.

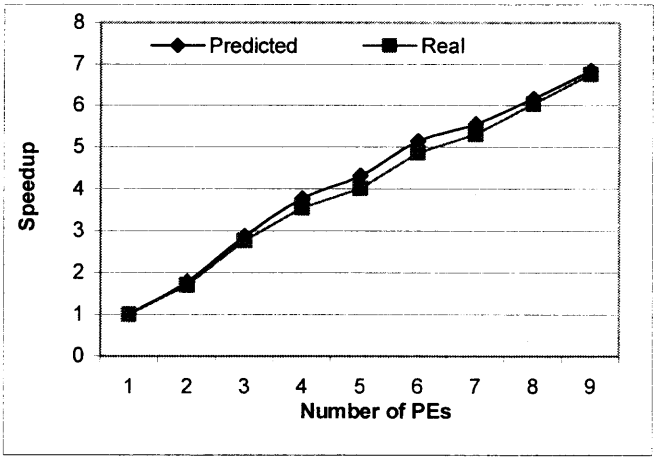


Figure 5.10 Comparing the predicted and real performance for the 7917-matrix.

Also, the transfer of matrix blocks between the on-board and on-chip memories becomes a bottleneck for a large number of PEs. We employed data pre-fetching and the relevant execution times for the 10279-Y_{bus} matrix are shown in Figure 5.11. Pre-fetching almost eliminates any contribution of the data load time to the total execution time by overlapping load operations with computations.

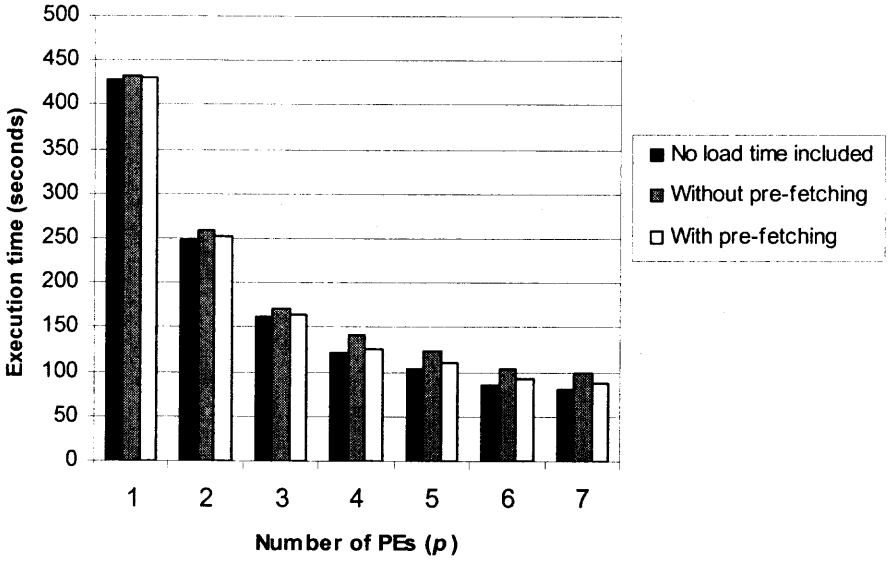


Figure 5.11 Execution time for the 10279-Y_{bus} matrix affected by pre-fetching.

5.3 Parallel LU Factorization of Sparse Matrices on HERA

Experiments implementing the SIMD, MIMD and mixed-mode scheduling schemes were performed on the 36-PE HERA machine. For the sake of comparison, similar to [Govindu, et al., 2004] we synthesized and implemented our design using the Xilinx ISE 5.2i toolset for an XC2VP125-7 FPGA. The divider in [Govindu, et al., 2004] uses a look-up table based reciprocator and a multiplier that result in precision errors. Table 5.4 shows that our FPU components generally result in higher frequency of operation; the overall latency and resource consumption are always smaller for our design. The test

matrices shown in Table 5.2 were used. The running times under these parallel execution modes are presented in Figure 5.12. It is clear that mixed-mode parallelism consumes less time for all the matrices and the advantage is more significant when the 3-block groups are highly irregular in size and shape, such as for the matrices of dimension 1723 and 7917. For the former (i.e., 1723) matrix, speed ups of 19.1% and 15.5% are obtained compared to the SIMD and MIMD implementations, respectively.

Table 5.4 IEEE single-precision floating-point performance and resource utilization

Functional Unit		Area (Slices)	Frequency (MHz)	Latency (Cycles)
Add/Sub	XC2V6000-5	349	163.2	3
	XC2VP125-7	348	184.1	3
	XC2VP125-? [G]	402/425/520*	100/150/220*	6/12/16*
Multiplier	XC2V6000-5	95	172.5	3
	XC2VP125-7	95	199.5	3
	XC2VP125-? [G]	130/201/229*	100/150/220*	4/7/10*
Division	XC2V6000-5	875	172.2	27
	XC2VP125-7	883	197.9	27

* Based on three special-purpose designs for LU factorization

Under the SIMD mode, some PEs are sometimes idle during the factorization of the 3-block groups and the multiplication of the border blocks. The total execution time is $T_{simd} = \sum_{i=1}^{m} T_{1_i}(FAC + MUL) + \lceil \log_2 p \rceil T_2 + T_{last}$, where $m = \left\lceil \frac{n-1}{36} \right\rceil$ and n is the total number of independent diagonal blocks. $T_{1_i}(FAC + MUL)$ is the maximum execution time among the PEs for the i^{th} iteration of jobs. T_2 is the time for a PE to perform one addition and one communication during the PAC work. T_{last} corresponds to the execution time for the last block. In MIMD, the *PAC* work may begin while some PEs are still working on *FAC* or *MAC* tasks. The worst case execution time is

$T_{mimd} = \max_{1 \le PE_j \le 36} \{\sum_{i=1}^{m_j} T_{1_i (FAC + MAC)}\} + \lceil \log_2 p \rceil T_2 + T_{last}$, where $T_{1_i (FAC + MAC)}$ is the execution time of the i^{th} iteration for PE_j that processes m_j 3-block groups. From the equations, it is easy to see why MIMD performs better than SIMD for the matrices of dimension 1723, 5300 and 7917, and worse than SIMD for the rest of matrices. The disadvantage of MIMD is that half of the memory is available for data. In our architecture, the communication cost in MIMD is not significantly higher than that in SIMD, so their performance is quite close. However, MIMD tends to perform better than SIMD in this algorithm for large matrices where we have a good chance that matrix blocks are more irregular and sparse. Due to insufficient work, all modes perform comparably for the 494 x 494 matrix.

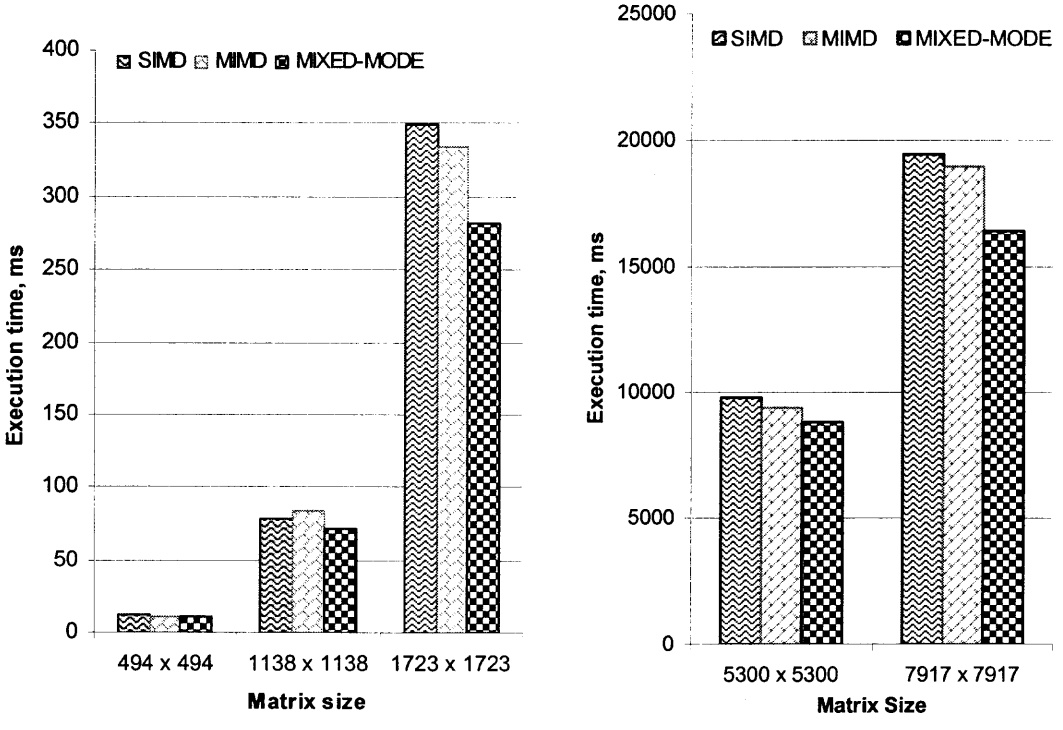


Figure 5.12 Execution times on HERA under the SIMD, MIMD and mixed modes (HERA system frequency: 125MHz).

Several previous works have presented fixed-point implementations of LU factorization. The only relevant work on floating-point LU factorization with platform FPGAs [Govindu, et al., 2004] involved just a dense 48 x 48 matrix and compared with a DSP processor. A circular linear array architecture is implemented, specifically for LU factorization. To resolve data dependencies, Govindu and et al. assume a stream of s "stacked" dense matrices; s has to be larger than the combined latency (in cycles) of the multiplier and subtractor units. The total latency is $s * n + s * n^2$ for n x n matrices and the throughput is one matrix per $n + n^2$ cycles. Also, shift registers are inserted in the datapaths that bypass the FPUs and the control logic must be able to delay the control signals. This design is inflexible since the number of processors and their storage space are specific to the matrix size.

We have to emphasize here that [Govindu, et al., 2004] attempts to maximize the throughput because it is an application-specific programmable circuit (ASPC) whereas HERA is a fully-programmable system. Results in [Govindu, et al., 2004] were reported for s = 10, 19 and 25. The latencies as shown in [Govindu, et al., 2004] for non-optimized code on the TMS320C6711 DSP processor are included in Table 5.5; matrices of various sizes were considered. Note that [Govindu, et al., 2004] only shows the "*effective* latency" which is actually the inverse of the throughput for processed matrices. It can be deduced that HERA's performance is much better than that of both systems.

There are 35 PEs in HERA running at 147MHz (for the Xilinx ISE tools) and each PE can complete three floating-point operations per cycle. Therefore, HERA has a peak performance of 15.44GFLOPs whereas the performance of the TMS320C6711x family is 600-1500MFLOPs (for 100-250MHz frequencies) [TMS320]. There are often

Table 5.5 Latency Comparison also Involving a DSP Processor (latency in msec)

Matrix Size	Design in [G] f = 100MHz		HERA f = 147MHz		TMS320C6711 [G] f = 150MHz
Number of matrices	1*	100	1**	100	1
8 x 8	0.06	6	0.052	0.156	11.5
16 x 16	0.26	26	0.431	1.293	23.0
24 x 24	0.58	58	1.210	3.630	55.2
32 x 32	1.02	102	2.920	8.760	87.5

* Average latency based on a stream of 100 matrices [[Govindu, et al., 2004]]. ** Single matrix

more disadvantages to DSP processors that rely heavily on the clock frequency for high performance. FPGAs can offer much more flexibility in memory hierarchy and configuration, system architecture and processor microarchitecture, data formats, interconnection networks, etc.; an FPGA-based design can be optimized based on various metrics such as energy, throughput, latency, area, design time, budget, etc.

5.4 Parallel Power Flow Analysis on CG-MPoPC

We implemented our parallel power flow analysis algorithm on our CG-MPoPC on the SOPC development board for the benchmark matrices shown in Table 5.6. The system frequency is 50MHz and seven processors with hardwired FPUs fit into the FPGA device. Table 5.7 shows the execution times of LU factorization, forward reduction and backward substitution (including communication times) for the chosen benchmark systems. The corresponding execution times for Newton's power flow solution are listed in Table 5.8. The uni-processor solution times in Table 5.8 are obtained by running the DBBD algorithm on a single processor. The obtained speedups are very good for our 7-processor parallel system. The speedup is also system-dependent. For example, for the

7917-bus system, the speedup is lower than those for the 300- and 1648-bus systems because of the larger number of cutting nodes; the process associated with these nodes becomes a bottleneck. From Tables 5.7 and 5.8, we can deduce that for a large system the time spent on LU factorization dominates the total execution time, which justifies our effort and time spent on ordering the Y_{bus} and Jacobian matrices.

Table 5.6 Optimal Partitioning of the Y_{bus} Matrices of the Benchmark Systems

Dimensionality of admittance matrix (Y_{bus})	57	118	300	1648	7917
Maximum nodes in a block	7	18	16	120	150
Number of independent diagonal blocks	7	7	21	18	67
Minimum dimensionality of independent diagonal blocks	4	11	6	33	15
Maximum dimensionality of independent diagonal blocks	7	18	16	120	150
Dimensionality of the last block	12	12	42	134	541
Size distribution of independent diagonal blocks*	5x7**, 6, 4	18, 18, 17, 16, 14, 12, 11	5x9, 6x16, 15, 3x14, 2x12, 3x10, 6	120, 109, 99, 3(90), 5(85)*, 79, 5(75), 33	7(150), 17(130), 10(120), 13(100), 19(90), 1(15)

*5(85) stands for 5 blocks of size close to 85 x 85.
**5 x 7 stands for 5 blocks of size 7 x 7.

Table 5.7 Execution Times (msec) to Solve the Linear Equations for the Benchmark Systems on our Configurable Multiprocessor (seven processors)

Benchmark systems	57-bus	118-bus	300-bus	1648-bus	7917-bus
Dimensionality of the Jacobian matrix	106	181	530	2982	14508
LU factorization of the Jacobian matrix	13.42	39.11	479.12	7,425	107,391
Forward reduction	0.56	1.12	6.61	102.1	3,210.7
Backward substitution	0.59	1.30	8.81	109.3	3,291.5
Total time	14.57	41.53	494.54	7,636.4	113,893.2

Table 5.8 Execution Times (sec) for Newton's Power Flow Equations with Seven Processors

Benchmark systems	57-bus	118-bus	300-bus	1648-bus	7917-bus
Iterations	4	4	5	5	6
Total time	0.069	0.198	2.582	39.21	712.4
Uni-processor	0.425	1.148	15.75	247.8	4,210.3
Speedup	6.16	5.79	6.10	6.32	5.91

Tolerance: 0.001p.u.

CHAPTER 6

HERA SYSTEM-LEVEL ENERGY MODELING

A system-level energy model for HERA is proposed based on physical-level implementation data and run-time application statistics to guide run-time scheduling decisions. As CMOS processes enter the deep submicron range, power or energy consumption is increasingly becoming one of the major challenges for most computing systems. For MPoPCs employed in embedded applications, power constraints form a critical design specification. Most of the power modeling and/or low-power design or research efforts on modern FPGAs [Shang, et al., 2001; Li, et al., 2005; Anderson, et al., 2004] involve sophisticated physical power models and assume detailed low-level design information at the gate- or register-transfer level. Continuous increases in chip density and gate count make these such low-level tools too slow and limit their applicability in architecture studies. In contrast, system-level power modeling [Brooks, et al., 2000; Ye, et al., 2000] is more practical and still reliable approaches for architects to quickly estimate power/energy consumption early in the design process. This is of utmost importance for FPGA-based systems that normally have a short turnaround time. Moreover, previous research has demonstrated that design decisions made in a very early phase of the development process, in which the design consists of a yet very abstract description (algorithmic abstraction level), have the greatest influence on power dissipation [Raghunathan, et al., 1998]. We follow this approach here. Prasanna's group has investigated extensively algorithmic-level energy modeling and optimization techniques based on application-specific architectures [Prasanna, 2005].

6.1 Related Work

Our energy modeling aims to provide a quantitative basis for performance-energy trade-offs at runtime. System-level power/energy modeling approaches for processors are generally instruction-based [Tiwari, et al., 1994; Sinha, et al., 2001] or component-based [Brooks, et al., 2000; Ye, et al., 2000]. In the former category, exhaustive energy measurements for all the instructions are performed and the total energy consumption of a program is obtained by summing up individual energy costs; the hindrance lies in estimating inter-instruction impacts [Tiwari, et al., 1994] (e.g., data dependencies) on the consumption. These processor-dependent results do not provide much information on the distribution of the consumption among individual components; this information, however, is prudent to use in architecture optimization. The second approach evaluates individual processor components, such as the ALU, controller, memory, bus, and cache, and develops a detailed model for them using physical design parameters. Component-level modeling is obviously time-consuming. A hybrid approach can be used with extensible processors [Sun, et al., 2005], where the energy consumption of the basic processor is modeled by an instruction-level technique whereas a component-based approach is applied to the custom extensions.

In contrast to significant power/energy modeling efforts and (micro)architecture-level exploration of microprocessors [Benini, et al., 1999; Benini, et al., 2000; Marculescu, et al., 2001], minuscule system-level results have appeared for on-chip multiprocessors or FPGA-based systems. An instruction-level rapid energy estimation approach for soft IP microprocessors on FPGAs is presented in [Ou, et al., 2004]; a processor is treated as a black box and the impact of inter-instruction interaction is

ignored. Moreover, due to major architectural differences between FPGA- and SoC-based designs, our MPoPC analysis cannot rely on previous results for fixed logic. [Loghi, et al., 2004] stimulates a shared-memory, bus-interconnected homogeneous ARM-based on-chip multiprocessor system to find out that the main consumers of power are the caches.

In our framework, HERA PEs are generated from an in-house developed hardware library that may contain diverse types of FP units. Our full knowledge of the HERA system provides a great advantage for accurate power modeling. Before deciding on appropriate system-level energy modeling and performance-energy optimization approaches, we performed experiments to evaluate the effect of various factors on power consumption. ISE 7.1 and XPower 7.1 from Xilinx, ModelSim SE 6.0 and Synplify Pro 8.0 are our power analysis tools. Based on physical-level measurements, we will show that the PEs are the main contributor of power and, hence, they become our focus in power modeling. The PE local memories are constructed with dedicated BlockRAM blocks in Xilinx FPGAs and exhibit different characteristics than those in [Loghi, et al., 2004]. We propose a state-based component-level power model. The activity cycles of the function units (FUs) in individual PEs are measured for a given application at run-time using dedicated monitoring hardware. Since the device-based physical power data of the FUs are evaluated only once at static time, the time spent on energy estimation for an application is quite reasonable.

6.2 Power Characterization of Library Function Units

The FUs in PHCL are the primitive components used to synthesize individual PEs for a semi-customized MPoPC. Hence, their performance and energy characteristics form the basis for system energy modeling. We use dedicated logic such as BlockRAM, 18 x 18 multipliers, and DSP blocks, rather than LUTs, as much as we can when designing the PHCL FUs. These embedded logic blocks have better performance in terms of delay and power consumption compared to LUT-based designs. The details of the PHCL FUs can be found in Chapter 7. Power dissipation in SRAM-based FPGAs can be broken down into static and dynamic parts. As feature sizes shrink, dynamic power has a decreasing trend because smaller-feature processes usually come with lower voltage and capacitance whereas static power rises dramatically. For modern FPGAs at 90nm or less, static power can exceed dynamic power.

The static power of an FGPA highly depends on the technology, the specific device and the working conditions; it is independent of the runtime activity rates of the FUs. Hence, it can be determined at static time using vendor power analysis tools. The static power is largely determined by the design size, which can be translated in our case into a precision choice for the FP FUs. Let $FU_{j,k}$ denote the k_{th} FP FU capable of the operation type j. Currently we support five FP operation types: +, -, *, /, and $\sqrt{}$. We approximate the contribution of $FU_{j,k}$ with the following equation:

$$P_{j,k}^{static} = P_{\Sigma}^{L,static} * \frac{L_{j,k}^{FU}}{\Phi} + P_{\Sigma}^{D,static} * \frac{D_{j,k}^{FU}}{X} + P_{\Sigma}^{M,static} * \frac{M_{j,k}^{FU}}{\Psi} \quad (6.1)$$

where Φ, X, and Ψ are the total number of logic resources expressed in logic cells, on-chip memory blocks, and embedded DSP blocks, respectively, of the target FPGA device. $P_{\Sigma}^{L,static}$, $P_{\Sigma}^{D,static}$, and $P_{\Sigma}^{M,static}$ are the total static power consumption of the logic

resources, DSP blocks, and on-chip memory blocks, respectively, of the chosen FPGA device. $L_{j,k}^{FU}, D_{j,k}^{FU}$, and $M_{j,k}^{FU}$ are the usage of logic resources, DSP blocks, and on-chip memory blocks of $FU_{j,k}$, respectively. Note that our calculation of static power for individual FUs is different from related works where the static power of the entire chip is used when comparing the static power of a design with its dynamic power. Since we aim to efficiently use all available resources, it is not fair to compare the static power of all the resources with the dynamic power of the few resources that individual FUs use. Table 6.2 shows the total power of the single and double-precision FUs in our PHCL. "S_" and "D_" stand for single- and double-precision, respectively. Their resource usage is shown in Table 6.1. The total number of slices in our target device (XC2V6000) is 33,792. As shown in Table 6.2, the dynamic power of all the FUs still dominates the total power. This table also shows that double-precision FUs consume much more dynamic power than corresponding single-precision FUs. The number of PEs that can fit in an FPGA device decreases if double-precision FUs are dictated by the application. Hence, the performance will be reduced due to a smaller number of PEs as compared to single-precision systems. However, the total power may decrease if the entire chip is engaged in computing. Figure 6.1 shows the power per slice characteristics of various designs. We can see that all the double-precision FUs, except the divider, consume a smaller amount of power per slice than their single-precision counterparts. It is clear from Table 6.1 that we can have more than double the number of PEs if we choose single-precision instead of double-precision. Hence, an important conclusion is that lowering the precision of operations should normally be expected to decrease the total energy consumption for a given application.

Table 6.1 Resource Usage (in slices) of Floating-Point FUs on XC2V6000-5

FU	Single-Precision	Double-Precision
Adder	343	745
Multiplier	119	836
Divider	731	3089
Square-root	666	2757

Table 6.2 Total Power Consumption (mW) of the IEEE-754 Single- and Double-precision FP FUs
(Clock Frequency: 100MHz; Average Input Activity Rate: 20%)

FU	Dynamic	Static	Total
S_ADD	247.5	4.1	251.6
D_ADD	472.52	8.91	481.43
S_MUL	75.92	1.42	77.34
D_MUL	493.76	9.997	503.76
S_DIV	559.84	8.74	568.58
D_DIV	2526.5	36.94	2563.44
S_SQRT	435.976	7.964	443.94
D_SQRT	1409.03	32.97	1442

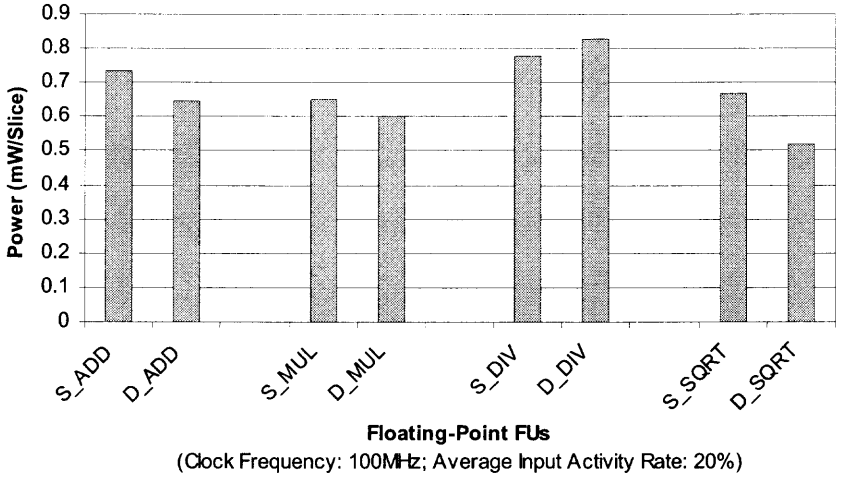


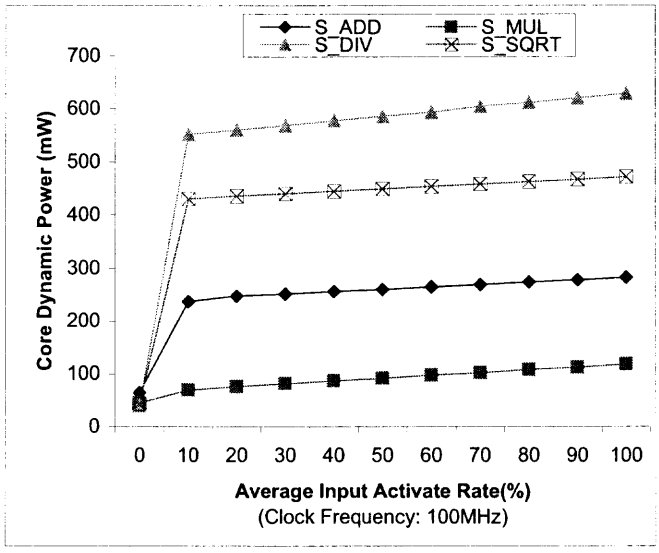
Figure 6.1 Dynamic power consumption (per slice) of the single and double-precision FP FUs.

Contributors to the dynamic energy consumption of an FPGA are the device core, and the auxiliary and I/O blocks. The latter two parts are directly related to the real board implementations, so we are only interested in the first factor. The dynamic power of an FU is determined by the following equation:

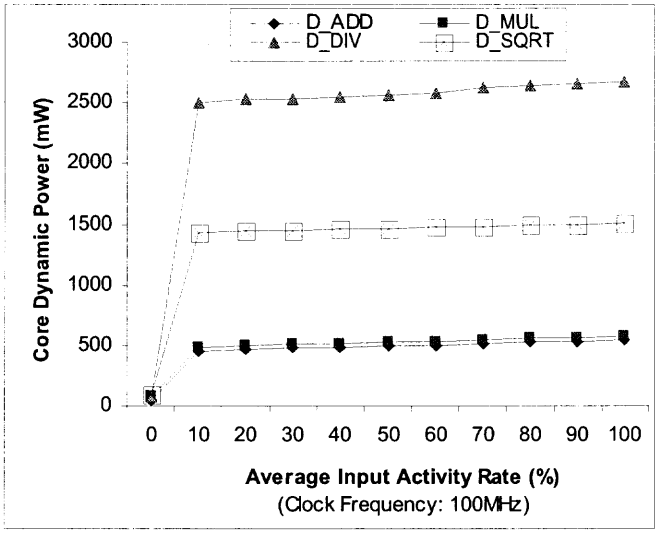
$$P_{j,k}^{dynamic}(F) = \alpha_{j,k} * F_{j,k} * C_{j,k} * V_{j,k}^2 \quad (6.2)$$

where $\alpha_{j,k}$ is the average number of activated switches per clock cycle inside the FU, $C_{j,k}$ is the switch capacitance in Farads and $V_{j,k}$ is the voltage in Volts.

We can have control the clock frequency and input activity rate of the FUs at the system level. Figure 6.2 shows the impact of the average input activity rate on the core's dynamic power. The differences due to different input activity rates are quite insignificant compared to the big gap between the idle (activity rate is 0%) and active states. Hence, we distinguish among four power states for each FU in HERA: *active, idle, standby*, and *sleep;* each FU is an indivisible block. An FU consumes both static and dynamic power in the active and idle states, and only static power in the standby state. All consumptions are eliminated by shutting down the power supply to an FU, which puts it into the sleep state. An FU enters the idle state when no instruction accesses it and its consumption is due to clock activities. An FU is put into standby by disabling its clock signal. Given the little differences in the consumption of the standby and sleep states, and the significant overhead of switching the power supply for individual FUs, we do not recommend high utilization of the sleep state. Moreover, more than 80% of the static power of our target FPGA is due to the auxiliary blocks.



(a) IEEE-754 Single-Precision FP FUs.



(b) IEEE-754 Double-Precision FP FUs.

Figure 6.2 Impact of the average input activity rate on the core dynamic power consumption.

The power consumption of $FU_{j,k}$ in the active, idle, and standby states is represented by $P_{j,k}^{idle}(F)$, $P_{j,k}^{active}(F)$, and $P_{j,k}^{stdby}$, respectively, where F is the system clock frequency and is determined after the system synthesis and implementation. $P_{j,k}^{stdby}$ is

the static power and is determined after the FU's implementation on a given FPGA device. $P_{j,k}^{idle}(F)$ is represented as a linear function of F and obtained by performing experiments with various F values. Both $P_{j,k}^{stdby}$ and $P_{j,k}^{idle}(F)$ are independent of the application. We also approximate the dynamic power part of $P_{j,k}^{active}(F)$ as a linear function of the input activity rate, as suggested by Figure 6.2. Note that Eq. (6.2) shows that the dynamic power portion of $P_{j,k}^{active}(F)$ is a linear function of $\alpha_{j,k}$ instead of the average input activity rate. $\alpha_{j,k}$ is dependent on the design as well as input activity rates. While it is impossible and impractical to perform an exhaustive simulation of input data to get the average activity rate of the design since HERA is used to solve repeatedly different sets of data produced at runtime, vendors suggest an average activity rate between 12% and 24% [Xilinx Power, 2003]. Given an application, we obtain a typical rate for each task by simulating the application using ModelSim and XPower to get $P_{j,k}^{active}(F)$. Figure 6.2 also implies that the clock consumption of different designs (when the input activity rate is 0%) is approximately the same for the same clock frequency. Figure 6.3 verifies that the core dynamic power consumption is proportional to the clock frequency.

The parameters associated for other system components, such as the BlockRAM blocks, the buses, and the Sequencer, are obtained by performing similar experiments. For the BlockRAM memory, it turns out that the power variations between read and write operations are very small. The input activity rates, including those of data and addresses, have very little impact on the consumption whereas the clock activity and the number of accesses are the main contributing factors. Hence, the clock of all the

BlockRAM blocks in HERA is controlled by a glitch-free enable signal provided with the memory blocks. Due to limited space, we do not show here the detailed experimental results for these components.

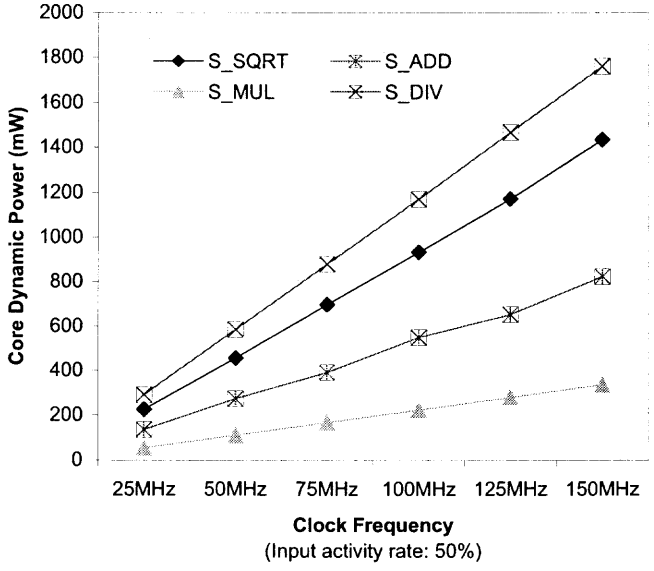


Figure 6.3 Relationship between the core dynamic power consumption and the clock frequency.

6.3 HERA Energy Estimation Model

Our energy model targets a dynamic HERA system during scheduling instead of the initial solution just after synthesis. The total number of PEs in the dynamic system changes as time evolves. Some PEs may disappear at some point and the resources will be used by new PEs, as needed. The details will be presented in Chapter 7. The total number of PEs during the entire execution is represented by p and all are assigned distinct IDs. The major components of HERA are the PEs, and their LDM and LPM, buses, NEWS interconnect, Sequencer, GDM and GPM, and the system template. The SDRAM chips are outside of the FPGAs and are not considered in this paper. The total energy consumption, E_{sys}, of HERA for a given application can be represented by:

$$
\begin{aligned}
E_{sys} &= \sum_{i=1}^{p} E_{PE(i)} + E_{sys}^{MEM} + E_{seq} + E_{bus} + E_{NEWS} + E_{sys}^{tpt} \\
&= \sum_{i=1}^{p} \{ [\sum_{j=1}^{5} \sum_{k} (E_{j,k}^{FU} * * \gamma_{i,j} * \gamma_{j,k}) + \sum_{m} E_{m}^{CFB} * \gamma_{i,m}] + \\
&\quad E_{PE}^{tpt} + E_{i}^{MEM} \} + E_{sys}^{MEM} + E_{seq} + E_{bus} + E_{NEWS} + E_{sys}^{tpt}
\end{aligned}
\tag{6.3}
$$

where E_i^{tpt} and E_{sys}^{tpt} represent the energy consumption of a PE and system template, respectively. A PE or system template includes mainly the control logic, system and pipeline registers, and decoding and issue logic. Although a PE or system template may require different numbers of resources for different configurations and, hence, may have different energy consumptions, we consider a constant energy value for different configurations because the FUs consume most of the transistors in a PE. In HERA, a PE template uses less than 10 percent of the logic resources in a PE. The template is treated as an FU and is evaluated by the same equation as for $E_{j,k}^{FU}$ except that the templates are in either the active or idle state (never in the standby or sleep state). The energy consumption $E_{j,k}^{FU}$ of FU$_{j,k}$ is determined by:

$$
\begin{aligned}
E_{i,j,k}^{FU} &= E_{j,k}^{active} * C_{i,j,k}^{active} + E_{j,k}^{idle} * C_{i,j,k}^{idle} + E_{j,k}^{sdby} * C_{i,j,k}^{stdby} \\
&= P_{j,k}^{active}(F_i) * \frac{1}{F_i} * C_{i,j,k}^{active} + P_{j,k}^{idle}(F_i) * \frac{1}{F_i} * C_{i,j,k}^{idle} + P_{j,k}^{static} * \frac{1}{F_i} * (C_{i,j,k}^{active} + C_{i,j,k}^{idle} + C_{i,j,k}^{stdby})
\end{aligned}
\tag{6.4}
$$

where F_i is the clock frequency of PE(i), and $E_{j,k}^{active}$, $E_{j,k}^{idle}$ and $E_{j,k}^{stdby}$ represent the energy consumption per clock cycle of FU$_{j,k}$ in the *idle*, *active*, and *standby* state, respectively. $C_{i,j,k}^{idle}$, $C_{i,j,k}^{active}$, and $C_{i,j,k}^{stdby}$ are the respective total clock cycles of FU$_{j,k}$ in PE(i) in the corresponding states; they are collected at runtime. If an FU has an s-stage pipeline, then $E_{j,k}$ in the three states can be determined by:

$$E_{j,k} = \frac{1}{S}\sum_{i=1}^{S} E_i \tag{6.5}$$

where E_i is the average energy consumption of stage i in the corresponding state. The energy consumption of the CFBs in a PE is determined by the same approach as for the FUs. The local memories (LDM and LPM) of PE(i) are implemented with on-chip embedded memory blocks (e.g., BlockRAM in Xilinx FPGAs). Based on our experiments in Section 7.1, we identify energy consumption for three states: *idle*, *one-port access* (*acc*_1), and *simultaneous dual-port access* (*acc*-2). Their consumption is:

$$E^{MEM} = \sum_{i=1}^{p} E_i^{MEM} = \sum_{i=1}^{p}\sum_{l=1}^{m_{i,m}} (E_{mem}^{idle} * C_{i,l}^{idle} + E_{mem}^{acc\_1} * C_{i,l}^{acc\_1} + E_{mem}^{acc\_2} * C_{i,l}^{acc\_2}) \tag{6.6}$$

where $m_{i,m}$ is the total number of memory blocks in PE(i), and E_{mem}^{idle}, $E_{mem}^{acc\_1}$ and $E_{mem}^{acc\_2}$ are the energy consumption per clock cycle of a memory block in respective state. E_{sys}^{MEM} is treated similarly. Similarly, we can find the energy consumption of the Sequencer by:

$$E_{seq} = E_{seq}^{idle} * C_{seq}^{idle} + E_{seq}^{active} * C_{seq}^{active} \tag{6.7}$$

The NEWS interconnect and the buses are implemented mainly with global routing fabric. It has been shown that a large part of FPGA power is due to the routing resources [Shang, et al., 2002; Li, et al., 2005]. Local routing resources are mainly used by PEs and counted in the PEs' power. We distinguish between two power states for them: *idle* and *active*, represented by P_{bus}^{idle}, P_{bus}^{active}, P_{NEWS}^{idle}, and P_{NEWS}^{active}, respectively. The total energy consumption due to the NEWS interconnect can be found by:

$$E_{NEWS} = \sum_{i=1}^{p} (E_{i,NEWS}^{idle} * C_{i,NEWS}^{idle} + E_{i,NEWS}^{active} * C_{i,NEWS}^{active}) \tag{6.8}$$

where $C_{i,NEWS}^{idle}$ and $C_{i,NEWS}^{active}$ are the total numbers of clock cycles in the respective states.

For the buses, the following equation is used:

$$E_{bus} = \sum_{i=1}^{m_b} E_i^{bus} = \sum_{i=1}^{m_b} (E_{i,bus}^{active} * C_{i,bus}^{active} + E_{i,bus}^{idle} * C_{i,bus}^{idle}) \tag{6.9}$$

where m_b is the total number of buses.

All the clock counts needed by the above equations are collected at runtime as each component is equipped with appropriate hardware. The counters can be read and reset by the host processor by using the *Configure* instruction. The bus activity information is monitored by the bus controller in the Sequencer. Each PE counts its own NEWS requests and memory accesses.

CHAPTER 7

A FRAMEWORK FOR RESOURCE-EFFICIENT MAPPING ON MPOPCs

As the VLSI technology continues to allow more resources on a single chip and promises billion-transistor chips in a few years, we expect FPGAs to evolve into coarse-grain architectures and reconfigurable MPoPCs to become more appealing, thus entering the mainstream of high-performance computing. The good performance results of CG-MPoPC and HERA provide strong evidence in this new research direction. However, as discussed in the Introduction and also from experience gained with the two MPoPCs, designing, implementing, and programming MPoPCs are very time-consuming processes that require a lot of expertise in the software, hardware, and system design areas. Moreover, the design complexity increases with increases in the chip size and thus this approach becomes more error-prone. As shown in previous chapters, current compilation tools based on the spatial computing concept are not applicable to MPoPCs. The functional units in our MPoPCs are reusable for different tasks by supporting general-purpose instructions and they are closer to temporal computing platforms from the programmer's point of view while still providing spatial parallelism. A new methodology is needed to take over the hardware expertise from the user and efficiently exploit the advantages provided by MPoPCs.

In this chapter, a framework is proposed to map data-parallel applications to coarse-grain reconfigurable MPoPCs like HERA without diving into the pains of VHDL-based hardware design. The focus here is to maximize the performance through customization of the PEs and efficient resource management at runtime based on the application's characteristics. This framework does not assume any specific device

characteristics and thus can be applied to any current or future FPGAs. The performance of the target system is predictable since the general system organization is fixed and a library of place-and-routed function units is employed to generate the system based on information from the mapping results. Also the requirement of full hardware reconfiguration of FPGA chips is eliminated during execution and advantage is still taken of partial runtime reconfiguration that overlaps the computations; this limited reconfiguration is used to change the functionality of PEs when required before further computation proceeds at the affected locations. This way, we can maximize the utilization of the available resources at rates unimaginable for conventional microprocessors.

7.1 Related Work

Resource management is a broad area and can be investigated from many perspectives. Ref. [Jin, et al., 2005] presents a methodology to find an optimal multiprocessor configuration that maximizes the throughput for IP packet forwarding; the configuration of each processor is fixed. [Sun, et al., 2005] proposes a novel method to synthesize a heterogeneous MPSoC for a given application by customizing the instruction set of extensible processors. Cesario et. al present a component-based MPSoC design approach in [Cesario, et al., 2002]. In contrast to our bottom-up approach in architecture synthesis that utilizes the in-house developed PHCL, a top-down flow based on a virtual architecture model is employed in the latter approach. Also, our approach emphasizes semi-customizing the architecture to a specific application. The benefits of [Cesario, et al., 2002] are high-level abstraction, high adaptability, and ease of integration for

software and hardware components. [Nollet, et al., 2005] proposes a resource management heuristic for NoCs (Networks-on-Chip) that involve reconfigurable logic tiles. Most of these approaches target real-time applications and assume full knowledge of the task load information at static time. HERA is a mixed computation mode machine and tasks are decomposable and will be mapped to multiple PEs at runtime. Hence, it is impossible to know their execution times at static time.

7.2 Problem Definition and Objectives

The starting point for our design exploration is a matrix-based data-parallel FP application and an FPGA chip that supports run-time reconfiguration. The latter has the following available resource populations: Φ (logic resources expressed in logic cells), X (on-chip memory blocks), and Ψ (embedded DSP blocks). Logic cells are the basic building blocks consisting of one or more look-up tables (LUTs) and storage elements (e.g., Configurable Logic Blocks (CLBs) in Xilinx FPGAs and Logic Elements in Altera FPGAs). The memory blocks are dedicated on-chip memory resources; e.g., BlockRAM or TriMatrix memory for Xilinx or Altera FPGAs, respectively. The DSP blocks, if available, can implement math functions. Our focus is the efficient management of the available resources based on HERA with three common performance-energy optimization objectives: (1) optimize the performance with no energy constraints; (2) optimize the performance with energy constraints; and (3) reduce the energy cost for a given performance loss.

7.3 Framework Overview

Figure 7.1 shows our general process flow targeting heterogeneous MPoPCs like HERA. There are five major phases in this framework: *task profiling*, *system synthesis*, *task coding* using the target system's instruction set, *system implementation* on FPGAs and *dynamic, adaptive resource-efficient task decomposition, mapping, and scheduling*. The implementation on FPGAs follows the same procedure as any VHDL-based design methodology. Due to limited space, we focus on task profiling, system synthesis, and dynamic resource management and application mapping and scheduling. Resource management is applied in two stages: (a) static-time application-specific system synthesis and (b) run-time adaptive scheduling of tasks.

The target architecture should have the following key features that can be found in HERA and support a variety of independent computing models (SIMD, MIMD, and M-SIMD):

- PEs are programmable and customizable from a library of hardware components;
- 2-D mesh layout;
- NEWS nearest neighbor connection;
- Each column of PEs has a shared bus;
- The dual-ported data memory of each PE is also directly accessible by its immediate south and west neighbors;
- All the local memories form a global data and program memory accessible to the sequencer;
- Every PE is selectable by the sequencer by its ID and mask;
- General-purpose instruction set.

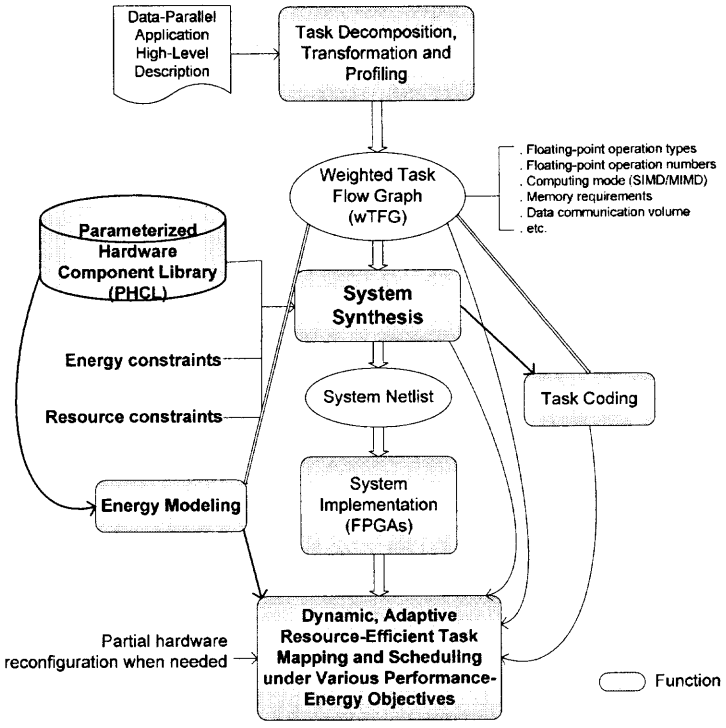


Figure 7.1 Design methodology overview/flowchart.

7.4 Application Model

We start from an application described in a high-level language, such as C/C++, Java, FORTRAN, or just a piece of behavioral pseudo-code. We target floating-point (FP) data-parallel and computation intensive algorithms, where a few blocks of code, such as nested loops, consume most of the overall execution time; these loops are controlled by conditional statements.

7.4.1 Task Flow Graph

In our framework, the behavioral description of the application is first analyzed to construct a *Task Flow Graph* (TFG), $G = (S, D)$; it is a *weighted, directed acyclic graph*

(wDAG). S and D represent the sets of nodes and edges, respectively. Figure 7.2 shows a typical TFG. Each node in this graph represents a task $S_i \in S$, where $i \in [1, s]$ is inclusive of all the tasks. There are two types of tasks: SIMD tasks and MIMD tasks. Associated with each task S_i are its computing mode (SIMD represented by a circle or MIMD represented by an octagon), any FP operation types (+, -, *, /, $\sqrt{}$), and an FP computation number (total FP operations of all the types), represented by $\varepsilon(S_i)$, $\pi(S_i)$, and $O(S_i)$, respectively. The memory requirements in bits of each task are represented by two parameters, $\sigma_c(S_i)$ and $\sigma_d(S_i)$, for the instructions and data, respectively. A directed edge between two tasks S_i and S_j represents a data dependence between them and its weight $D_{i,j} \in D$ represents the volume of data in bits that goes from task S_i to S_j. An *entry task* is defined as a node with no incoming edges (e.g., S_1 in Figure 7.2) and an *exit task* is defined as a node with no outgoing edges (e.g., S_8).

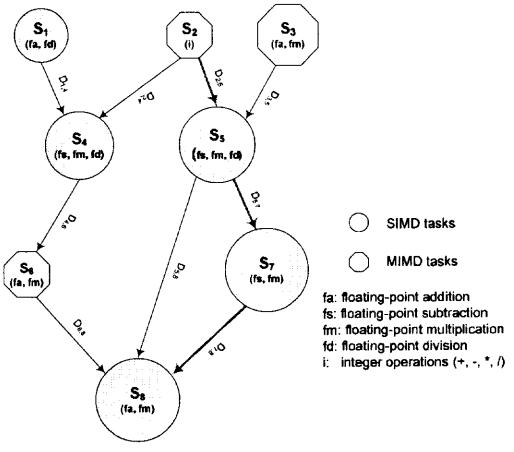


Figure 7.2 A typical task flow graph.

A typical data-parallel application in engineering and science consists of blocks of conditional statements and nested loops. We concentrate on coarse-grain partitioning of applications and the sizes of tasks, illustrated by their covered areas in the TFG (for the sake of simplicity), may vary in a large range. We first analyze the application to

locate typical computation constructs, and approximate the amount of computation and communication that each block requires. Blocks are identified by their leading keywords, such as *for*, *if*, *while,* etc. The selection of an optimal mode for each task is a complex procedure and is not the focus here. Reference [Watson, et. al., 1994] provides some insight into this issue based on PASM [Siegel, et. al., 1996], a partitionable SIMD/MIMD system employing COTS microprocessors and a multi-stage interconnection network. An SIMD task in our study is a data-parallel block (e.g., a nested loop), which can benefit from synchronous execution under SIMD, while an MIMD task is more of the control-flow style which may need one or more PEs. It is assumed that the computation cost of MIMD tasks is much less than that of SIMD tasks, which is common in data-parallel applications. An SIMD work may need just one PE based on its work load and my heuristics for candidate PE selection for tasks; this decision process will be introduced later in this paper. Specifically, two common types of tasks are handled with special attention: *IF-THEN-ELSE* conditional statements and loops.

7.4.2 IF-THEN-ELSE

SIMD is implicitly synchronous and conditional statements are generally inefficiently implemented on a pure SIMD machine. For example, consider the code and its SIMD and MIMD mappings in Figure 7.3. In this case, assume the conditions C_1, C_2, C_3 are determined based on the PE's local data which are not modified by $block_1$, $block_2$, and $block_3$. Let the best computing modes for $block_1$, $block_2$, and $block_3$ be MIMD, SIMD, and SIMD, respectively. It is also assume that these conditions are mutually exclusive.

(i.e., only one condition is true at any time in all PEs). Thus, PEs are divided into groups with various sizes and the PE locations are unknown at compile time. In the PE_C_i group (i = 1, 2 or 3), the condition C_i is true while the *OTHERS* group contains all the PEs where all the three conditions are false. Let the execute time of $block_i$ be $T(i)$. The total execution time of this code on a pure SIMD machine is $\sum_{i=1}^{3} T(i)$ while the time is $max\{T(1), T(2), T(3)\}$ if all PEs in the same machine run in the MIMD mode. Moreover, more PEs are idle in the SIMD mode and the corresponding resources are wasted. However, some overhead is introduced in the pure MIMD machine due to the unsuitable mode for $block_2$ and $block_3$. In our mixed-mode systems, this dilemma can be handled very efficiently by configuring the system into the MIMD/M-SIMD mixed-mode: PE_C_1 in MIMD, PE_C_2 in SIMD-1 and PE_C_3 in SIMD-2.

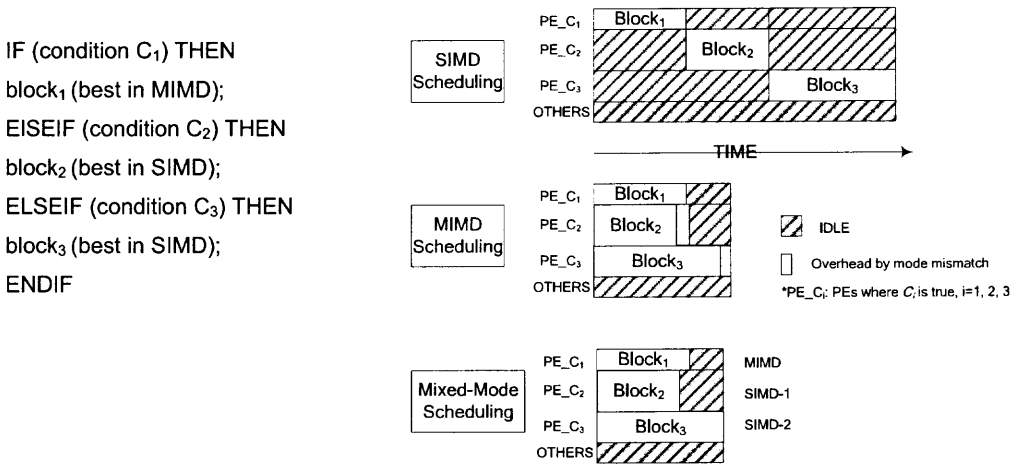


Figure 7.3 SIMD, MIMD, and mixed-mode mapping of conditional blocks.

7.4.3 Loops

Loops are extremely useful for managing and processing large amounts of data, and are very common in data parallel applications. They often represent the most time-

consuming parts in programs and correspond to the largest source of parallelism. Significant research efforts for many decades have concentrated on efficient parallelizing and scheduling loops for diverse parallel architectures [Rauchwerger, et. al., 1999; Gupta, 1992; Polychronopoulos, et. al., 1989; Polychronopoulos, et. al., 1987]. *FOR* loops are the most common loops in data-parallel algorithms. In practice, these loops have diverse characteristics and special preprocessing techniques are required to maximize the speedup. Specifically, the following type of loops is studied in this work.

```
FOR i1 = S1 TO N1 DO
        Statements;
        FOR i2 = S2 TO N2 DO
                Statements;
                ...
        ENDFOR
ENDFOR
```

where $i1$, $i2$, ..., $S1$, $S2$, ..., $N1$, $N2$, ..., are integers and $Si \leq Ni$, for $i = 1, 2, ...$

No conditional exits are present in this type of loops. Conditional *FOR* loops can essentially be transformed into *WHILE* loops and treated as the latter. Loops may be nested as frequently in practice as needed and may be treated as a single SIMD task with partitioning left to the scheduler. Matrix-matrix multiplication is an example of regular, simple nested loops. The three indices have the same iteration range and there is no data dependence inside or between loops. This work focuses on more complex scenarios where the data dependences inside loops and between different iterations of a single loop are allowed. Data dependences between different loops are treated on the task level. Flow dependences and cross-iteration dependences [Kwang, 1993], shown in Figure 7.4 (a) and (b), are the major obstacles for parallelization. In (a), the ten iterations of the loop are independent of each other, but there is a flow dependence [Kwang, 1993] between two statements. Each iteration of the loop can be executed in

parallel. In (b), the ten loops have to be processed in the appropriate order because the i^{th} iteration needs the result from the $(i\text{-}2)^{th}$ iteration. Also, there may be load imbalance across iterations. Figure 7.4 (c) shows such an example, which is the loop body of the forward substitution in direct solvers of linear systems of equations.

```
for (i = 0; i++; i<10)
{
B[i] = 2 * A[i];
D[i] = B[i] + 2;
}
```

(a) Flow dependence.

```
for (i=2; i++; i<12)
{
A[i]=2 * A[i-2];
}
```

(b) Cross-iteration dependence.

```
for (k=1; k<n; k++)
   for (i=k+1; i<=n; i++)
      x[i]= x[i]- A[k][i]*x[k]/A[k][k];
```

(c) Load imbalance across iterations.

Figure 7.4 Special examples of FOR loops.

7.5 Architecture Synthesis and Reconfiguration

Our first major effort in efficient resource management is the synthesis of a semi-customized HERA configuration. The rationale behind this is that FP computing cores are very resource expensive (especially for FPGAs) and not all FP operation types are needed all the time by all the tasks in an application. For example, FP division is less frequent than multiplication and addition in many data-parallel applications. Based on our implementation results, a single-precision IEEE-754 FP divider is at least more than two or six times larger in space than an FP adder or multiplier, respectively. These differences are even larger for a double-precision FP divider. Unlike computing platforms based on fixed hardware, such as microprocessors and MPSoC designs, our MPoPCs are based on FPGAs which can be repeatedly reprogrammed at both static and run times. Hence, it is possible and beneficial for an application to employ a dynamic HERA architecture where the FP functionality of individual PEs and their number can be modified as needed.

7.5.1 Parameterized Hardware Component Library

PHCL plays a major role in our methodology. The performance of the library function units (FUs), in terms of speed and resource requirements, is manipulated in our approach. All the components are designed in VHDL and placed and routed on the target FPGA device. The major parameterized components for our matrix-based applications include:

- Variable precision pipelined FP FUs (including IEEE-754 single- and double-precision implementations). Table 1 shows the major parameters of an FU. A slice in Xilinx Virtex II FPGAs consists of two flip-flops, two LUTs and associated MUX, carry, and control logic [Virtex II datasheet]. The cores are parameterized by the mantissa and exponent sizes. Different choices for the mantissa and exponent lead to different data ranges and resource requirements. For each operation type (+, -, *, /, and $\sqrt{}$) of the same precision, there are also several choices in terms of latency, resource requirement, frequency, and power consumption. P^{peak}, P^{active}, P^{idle}, and P^{stdby} represent the dynamic power consumption of the FU in the *worst-case*, *active*, *idle*, and *standby* states, respectively, and will be introduced in detail in the next section. Since FP cores are major consumers of logic resources and embedded DSP blocks in FPGAs, it is very important to choose each time the most appropriate precision for the function cores.
- HERA system and PE architecture templates used to create an instance of the system and PE, respectively. A PE or system template includes mainly the control logic, basic interconnects, generic PE or FU interface, and registers.
- Memory blocks of various sizes, including single and dual-port memories.

- Custom function blocks (CFBs), such as trigonometric function implementers.
- Various registers.
- Integer function cores parameterized by word size.

TABLE 7.1 Major Parameters of an FP FU in PHCL

Parameter	Data
Function	FP division
Mantissa (bits)	24
Exponent (bits)	8
Latency (cycles)	27
Frequency(MHz)	189
Logic resources (slices)	731
Embedded DSP blocks	None
Memory blocks (BRAM)	None
Target device	Virtex II XC2V6000-5
P^{peak} (mW)	1181.2
P^{active} (mW)	1041.8
P^{idle} (mW)	141.2
P^{stdby} (mW)	11.7

7.5.2 Application-Specific System Synthesis

The primary goal is to find early on a near-optimal configuration of the PEs for *each task in the critical path* (referred to as TiCP from this point on) so as to achieve given performance-energy objectives. A critical path in a TFG is a linear array that includes a pair of entry and exit tasks, and has the largest number of FP operations and communication volume among all paths. The critical path potentially has the largest negative impact on the overall execution time. The architecture generator takes a TFG and the PHCL as input to generate an initial architecture; it employs the operation types

and amounts of operations in the TFG. Only the required FP FUs are included in the PEs. It is possible that all the operation types are required throughout execution but only one or two operations are needed for a few tasks. Hence, FUs are added to appropriate PEs as needed. This way, we can potentially increase the number of PEs and reduce the execution time for the application. The PE functionality can be reconfigured at run time as needed by assigned tasks. We could get better solutions by configuring the PEs associated with each TiCP in such a way that only the FP operations exclusively dictated by a task are supported. However, this may require many full and partial device reconfigurations that may results in substantial reconfiguration overhead. Hence, we are mainly interested in PRTR of FPGAs when no computation can be scheduled on some PEs while the remaining PEs are still working on their assigned tasks. FRTR is employed only when the performance gains exceed the complete-system reconfiguration overhead. This is often true with large matrices as we will demonstrate in Section 7.7.2. The synthesis procedure is as follows:

1. Find the appropriate FP precision for the system based on the precision requirements of the application. This step largely determines the total number of PEs that can be implemented in the system.
2. Identify the required FP operation types (+, -, *, /, $\sqrt{}$) in the application's tasks.
3. Select a system template in PHCL assuming the basic PE interconnection and interface to the sequencer. Let the template requirements in logic cells and memory bits be $L_{sys}(p)$ and $M_{sys}(p)$, respectively, for p PEs. The PE datapath (functionality and width), total number of PEs and the PE layout are customizable.

4. Select CFBs in PHCL, assuming the resource requirement for logic resources, memory blocks, and DSP blocks of the m_{th} CFB is L_m^{CFB}, D_m^{CFB}, and M_m^{CFB}, respectively.
5. Select FP FUs for the PEs for each task S_i in the critical path according to the chosen performance-energy objective. In initial synthesis, all the PEs for the same task have the same configuration. The configuration of some PEs will be changed through PRTR during the scheduling phase as needed by the tasks outside of the critical path (non-critical tasks).

Let $FU_{j,k}$ denote the k^{th} implementation in PHCL of an FP FU capable of the operation type j. The resource requirements and the energy consumption per cycle E_i of PE_i for the TiPC S_i are:

$$L_i = \sum_{j=1}^{5} \sum_{k} L_{j,k}^{FU} * \gamma_{i,j} * \gamma_{j,k} + \sum_{m} L_m^{CFB} * \gamma_{i,m} \tag{7.1}$$

$$D_i = \sum_{j}^{5} \sum_{k} D_{j,k}^{FU} * \gamma_{i,j} * \gamma_{j,k} + \sum_{m} D_m^{CFB} * \gamma_{i,m} \tag{7.2}$$

$$M_i = \sum_{j}^{5} \sum_{k} M_{j,k}^{FU} * \gamma_{i,j} * \gamma_{j,k} + \sum_{m} M_m^{CFB} * \gamma_{i,m} \tag{7.3}$$

$$\begin{aligned} E_i &= \sum_{j=1}^{5} E_{j,k}^{FU} * \gamma_{i,j} * \gamma_{j,k} + \sum_{m} E_m^{CFB} * \gamma_{i,m} \\ &= \sum_{j=1}^{5} P_{j,k}^{peak} * \frac{1}{F_i} * \gamma_{i,j} * \gamma_{j,k} + \sum_{m} P_{CFBm}^{peak} * \frac{1}{F_i} * \gamma_{i,m} \end{aligned} \tag{7.4}$$

$$\gamma_{i,j} = \begin{cases} 0 & \text{if PE(i) does not support the FP operation type } j \\ 1 & \text{if PE(i) supports the FP oprtation type } j \end{cases}$$

$$\gamma_{j,k} = \begin{cases} 0 & \text{if PE(i) does not include an } FU_{j,k} \\ 1 & \text{if PE(i) includes an } FU_{j,k} \end{cases}$$

$$\gamma_{i,m} = \begin{cases} 0 & \text{if PE(i) does not include a } CFB_m \\ 1 & \text{if PE(i) includes a } CFB_m \end{cases}$$

where $L_{j,k}^{FU}, D_{j,k}^{FU}$, and $M_{j,k}^{FU}$ are the usage of logic resources, DSP blocks, and on-chip memory blocks of $FU_{j,k}$, respectively. $E_{j,k}^{FU}$ is this FU's energy consumption per cycle. L_m^{CFB}, D_m^{CFB}, and M_m^{CFB} are the usage of logic resources, DSP blocks, and on-chip memory blocks of the m^{th} *CFB*, respectively. F_i is the system frequency for task S_i. Note that up to one instance of an FU for each FP operation is included in each PE. Hence,

$$\sum_k \gamma_{j,k} \in \{0,1\} \tag{7.5}$$

Let p_i be the number of PEs to be implemented for task S_i. The total execution time of the application is dominated by the TiCPs and can be approximated by:

$$T_\Sigma = \sum_{i=1}^{c} \frac{C(O_i, p_i)}{F_i} + \frac{N_{conf} * C_{conf}}{B * F_c} \tag{7.6}$$

where c is the total number of the TiCPs. $C(O_i, p_i)$ is the minimum clock cycles needed for task S_i when the required p_i PEs are available; for simplicity, we represent $O(S_i)$ for task S_i by O_i. It is possible that the optimal PE number for the minimum cycles is not p_i. This will be explained and dealt with when we discuss run-time scheduling. $C(O_i, p_i)$ is obtained with symbolic simulation on HERA before synthesis. N_{conf} is the total number of reconfigurations during the entire execution of the application. If a required FP operation is not supported before scheduling a task, reconfiguration will apply. For each PRTR, we count the number of reconfiguration by the percentage of reconfiguration bits over the total configuration bits for the target device, which is represented by $C_{Conference}$ B is the configuration word width per cycle. For example, Xilinx Virtex FPGAs support both parallel (B = 8 or 32) and serial (B = 1) configuration modes. F_c is the configuration frequency which is usually lower than the system frequency F. The maximum F_c for Virtex II FPGAs is 50MHz (serial mode) or 66MHz (SelectMAP

mode). Assume capacities of $M_c(i)$ and $M_d(i)$ for the instruction and data memories, respectively, of each PE_i.

To estimate the energy consumption of the application, we sum up the average energy consumption of all the tasks and the total reconfiguration energy overhead:

$$E_{\Sigma} = \sum_{i=1}^{s} (\sum_{j} O_{i,j} * C_{j,k} * P_{j,k} * \frac{1}{F_i}) + T_{\Sigma} * (\tilde{P}_{sys} + \tilde{P}_{mem}) + \frac{N_{conf} * C_{conf}}{B * F_c} * P_{conf} \qquad (7.7)$$

where s is the total number of tasks, $O_{i,j}$ is the number of j^{th}-type operations in S_i, and $C_{j,k}$ is the latency (clock cycles) of $FU_{j,k}$. P_{conf} is the average configuration power for the entire chip. The average active power data of $FU_{j,k}$ is used. $\tilde{P}_{sys}$ and $\tilde{P}_{mem}$ are the average power of the system template and BlockRAM memory, respectively. For the sake of simplicity, we are primarily interested in first-order factors here and neglect some runtime effects, such as the impact of data dependencies. However, this is sufficient to serve our purpose here, and we will refine the performance and energy in task scheduling. The following three scenarios are considered to satisfy various performance-energy optimization objectives.

- **Case-1: Optimize the performance without energy constraints.**

  Our objective is to minimize T_{Σ} subject to Eq. (7.5) and the resource constraints imposed by the FPGA device. The objective functions are:

$$L_{sys} + \sum_{i=1}^{p_i} L_i \leq \Phi \qquad (7.8)$$

$$M_{sys} + \sum_{i=1}^{p_i} \{[M_c(i) + M_d(i)] + M_i\} \leq X \qquad (7.9)$$

$$\sum_{i=1}^{p_i} D_i \leq \Psi \qquad (7.10)$$

- **Case-2: Optimize the performance under energy constraints.**

Let E_B be the energy constraint (i.e., the allowable upper bound). The objective is then to minimize T_Σ subject to the following constraint in addition to Eqs.(7.5) and (7.8)-(7.10):

$$E_\Sigma \leq E_B \tag{7.11}$$

In general, the system frequency F does not affect E_Σ since $T_\Sigma \propto \frac{1}{F}$ and $P \propto F$.

Hence, in the aforementioned two cases, we use the maximum system frequency for each task S_i that can be achieved by the chosen FUs in the PEs that can reduce the execution time.

- **Case-3: Optimize the energy cost for a permissible performance loss.**

We assume that the base execution time T_B is given in Case-1. Let β be the permissible performance loss. Our objective is to minimize the total energy cost E_Σ subject to the following constraint in addition to Eqs.(7.5) and (7.8)-(7.10):

$$T_\Sigma \leq (1+\beta)T_B \tag{7.12}$$

Since device reconfiguration incurs significant time and energy overheads, we reduce the number of reconfigurations, and hence potentially increase the execution time, up to the limit set by Eq. (7.12). Another possible approach is to reduce the number of PEs for each TiCP according to the ratio of β. However, this may affect the scheduling of other tasks that may violate the performance constraint.

If the energy budget in Case-2 cannot be satisfied, we have to go back to Step 1 in the synthesis procedure and choose a lower precision for the FP FUs. From above equations, we can see that the exploration can be performed from several dimensions

and it is impossible to investigate all possible configurations for a reasonable period of time. Hence, we introduce several limiting factors, such as (1) limiting the number of reconfiguration, (2) using the fastest FUs, (3) using the same system frequency for all the tasks, as a starting point for synthesis in order to reduce the solution complexity and time. For less frequently used FP operations, we consider the following cases. An FP divider is used as an example. The final optimization can be solved with an ILP (Integer Linear Programming) solver.

A. Large tasks appear in the critical path.

For example, $S_1 \rightarrow S_3 \rightarrow S_5$ is the critical path in the TFG of Figure 7.5 (a); also an FP divider is required by S_3 which is a large task based on its number of FP operations. In this case, an FP divider is initiated for all PEs at the very beginning.

B. Small tasks appear in the critical path.

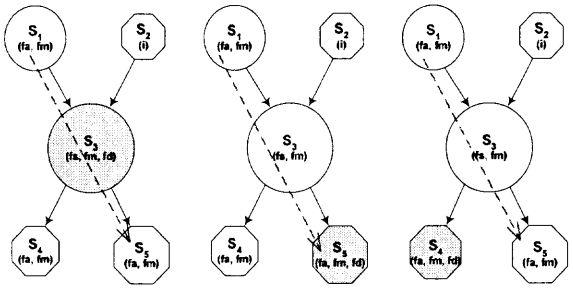
S_5 in Figure 7.5 (b) is such an example. Because S_3 is the task that potentially contributes the most to the execution time, the priority is to maximize the number of PEs for S_3 with the inclusion of an FP adder and a multiplier as well. No PE contains an FP divider until the execution of S_5. Some PEs will be reconfigured to add an FP divider when the time comes for S_5.

C. Tasks are not in the critical path.

This case is treated similarly to *Case B.* For example, an FP divider is added to some PEs at the time needed to accommodate task S_4 shown in Figure 7.5 (c).

Note that we are only interested in symbolic analysis of the application during synthesis instead of actual FP calculations, so the solution time for the synthesis should be reasonable. Also, since the synthesis happens at static time, it is tolerable and

worthwhile to go through such an optimization procedure for a given class of applications. The same procedure applies to different sets of data.



(a) (b) (c)

Figure 7.5 An example of function selection for PEs.

7.6 Dynamic Resource Scheduling for Performance-energy Optimization

This section presents our second step of resource management, namely, run-time adaptive scheduling with various performance-energy objectives.

7.6.1 Related Work

Based on various application scenarios, system architectures and performance objectives, extensive scheduling research targeting multiprocessors has been done for conventional fixed parallel architectures [Pinedo, 2002; Kwork, et. al., 1999; Grajcar, 2001; McCreary, et. al., 1994; Ziavras, 1993]. Scheduling is an NP-complete problem and a good heuristic for near-optimal performance should be the goal. To compare different scheduling techniques, we should consider the following aspects: objectives; target architectures and their internal organization; target applications; dynamic or static approach; central dispatch or distributed cooperation; and etc. Different scenarios

handle different parameters and constraints, hence, their efforts are different. The most appealing and efficient scheduling heuristics for multiprocessors are *list scheduling* [Adam, et. al., 1974; Gerasoulis, 1996] and a large body of its variants. List scheduling is a task-oriented strategy which statically assigns a priority to each task and schedules the tasks according to their reverse order of priority. Only one processor is considered for each task. The differences among the algorithms based on list scheduling are the assignment of priorities and several assumptions. A major problem is that their assignment of priority without any runtime knowledge may lead to an inefficient schedule [Grajcar, 2001; McCreary, et. al., 1994]. Dynamic critical path scheduling [Kwok, et. al., 1996] can reduce the schedule lengths of list scheduling by incorporating run-time information into the scheduling decision but can not save execution times and resources. Also, it assumes dedicated hardware for communication and computation that do not interfere with each other and these operations can happen simultaneously. However, in reality communication time is becoming more significant in state-of-the-art parallel systems as the computation power of processors is improved exponentially, especially when fine-grain tasks are the target. Most of these algorithms concentrate on one issue and make unrealistic simplifications on other issues, such as unlimited number of processors, no communication costs and no data dependence [McCreary, et. al., 1994; Grajcar, 2001]. Moreover, none of the above algorithms assumes reconfigurable architectures.

Our target architecture is our semi-customized HERA mixed-mode MPoPC that supports partial reconfiguration; the input to our scheduling policies is the *Task Flow Graph* introduced in Section 7.4. Taking advantage of HERA's mixed-mode parallelism

and reconfigurability, our run-time scheduling focuses on dynamic decomposition and redistribution of active SIMD tasks to available PEs. In our approach, different numbers of PEs may be applied to an application's task in its lifetime as long as additional PEs are available and the scheduling objective allows allocating new PEs to a task. We propose scheduling schemes for various performance-energy objectives. An MPoPC advantage over traditional multiprocessors is that the communication overhead is dramatically reduced; this helps to collect information in dynamic scheduling. The closest work to our scheduling schemes is for MPSoCs. Most of the latter with similar objectives and target applications [Sun, et al., 2005; Srinivasan, et al., 2004] focus on exploiting dynamic voltage scaling (DVS) (e.g., [Meyer, et al., 2005]) or dynamic power management (e.g., [Zhu, et al., 2003]). They often assume streaming tasks with periodic or aperiodic rates and deadlines. Furthermore, most of them attempt either to minimize the energy/power or maximize the performance instead of studying tradeoffs that involve both metrics. [Kadayif, et al., 2005] proposes a static technique to determine the optimal number of processors for individual arrays in a bus-based shared-memory MPSoC. Besides major architectural differences as compared to HERA, it assumes one task at a time and independent arrays; also, a fixed processor size is used for each array throughout execution. Due to the shared-memory nature of their architecture, the best processor number for most benchmarks is less than five. In contrary, our runtime objective is to balance the available PEs among various data and control dependent tasks instead of applying the optimal number of processors to each individual task. There are more key features that distinguish our strategy from existing approaches. First, we target a real reconfigurable multiprocessor under real resource

constraints. Second, we everntually determine the appropriate number of PEs and the binding of tasks to PEs at run time. Third, we dynamically reconfigure HERA to accommodate the needs of tasks while reducing simultaneously the idle time of the resources.

Let us first look at some important subtasks in this procedure before presenting the overall algorithm.

7.6.2 Loop Partitioning

Our adaptive scheme explores runtime task decomposition and distribution. Most SIMD tasks in our target applications are loops. Hence, loop partitioning is the basis of our adaptive parallelization. We restrict our effort to assigning each time the complete or part of an iteration to a PE. Hence, flow dependence is allowed inside an assigned iteration. We distinguish among three cases.

FOR loops without cross-iteration dependence

Assume that the total number of PEs available to the loop is κ and the total number of iterations is L. The loop space is split into κ groups each of size $\lfloor L/\kappa \rfloor$ or $\lceil L/\kappa \rceil$. Each PE gets such a group and the corresponding data set. These loops conform to the SIMD mode and no communication is required. *FOR* loops without both flow dependence and cross-iteration dependence are treated the same way.

FOR loops with cross-iteration dependence

We assume that the distance between successive data dependent iterations is w. Figure 7.6 shows the data dependences in a loop with $w = 2$. Let the total iteration space L in the loop be a multiple of w and the loop can be divided into w partitions. The i^{th}

partition contains the iterations $l_0 + i + k*w$, where $k \in [0, L/w-1]$ and l_0 is the starting point of the loop index, for $i \in [0, w-1]$. Each partition is then further divided into κ groups of size $\lfloor L/(\kappa*w) \rfloor$ or $\lceil L/(\kappa*w) \rceil + 1$ with continuous iterations. Each PE gets such a group and the corresponding data set. By distributing data this way, data communication is restricted between two neighboring groups.

Heterogeneous Loops

Each iteration in heterogeneous loops has different FP operations. Let f_i be the number of FP operations in the i^{th} iteration and $f_i = a*i + b$. Such a loop can be transformed into a homogeneous loop by combining the i^{th} and $(n-i-1)^{th}$ iterations into a new loop with a constant number of operations [Cierniak, et. al, 1995]. Then the above partitioning techniques can be applied to these loops.

Other loops that can be transformed into *FOR* loops (e.g., *WHILE* loops) are treated similarly.

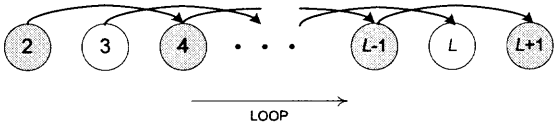


Figure 7.6 Cross-iteration dependence.

7.6.3 PE Search

As discussed earlier, the distance between PEs is one of the two critical parameters to the communication cost between tasks. Thus, the order in which we search for one or more candidate PEs is an important step affecting the overall mapping performance. Based on the HERA organization and interconnect network, we propose column-oriented PULSE search shown in Figure 7.7 (a). The motivations include the following:

- Our tasks are abundant in loops which favor the SIMD mode. The column buses in HERA can be used to broadcast instructions in SIMD and M-SIMD.
- In this searching pattern, the distance between two adjacent stops is always one.
- One port of the data memory of each PE is shared with its immediate neighbors to the west and south. By selecting candidate PEs in the PULSE pattern, there is a large chance that we can save on communication time. For example, consider the TFG shown in Figure 7.7 (b) and assume that S_1 and S_2 have already been executed by PE_3 and PE_4, respectively. We assume that S_3 has to be mapped to a PE other than PE_3 or PE_4. If S_3 is assigned to PE_8, then the communication distance is only one hop (PE_4 is asked to get the result of S_1 from PE_3, and PE_8 can access both results from S_1 and S_2 in the local memory of PE_4).

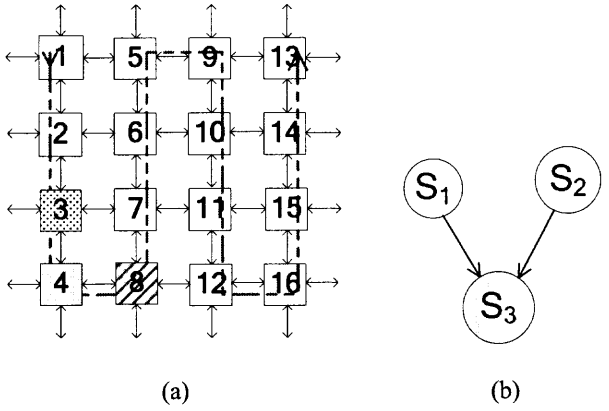


Figure 7.7 PE search path.

Assume that the numbers of PEs assigned to tasks S_i and S_j are p_i and p_j, respectively, and $x = min\{p_i, p_j\}$. The objective function to be minimized in this step is the communication time among tasks:

$$T_{com} = \sum_{i,j=1}^{n} \max\{[D_{ij}/(\lambda * x) + T_{ini}] * H(i,j) + T_{cflit}\} \tag{7.13}$$

where D_{ij} is the amount of data communicated between these two groups of PEs, λ is the transfer speed in bits/second between two immediate neighbors, T_{ini} is the overhead to initialize the transfer, $H(i, j)$ is the number of hops between two communicating PEs and T_{cflict} is the routing delay caused by data collisions. In order to reduce the collision and communication costs, data locality is taken into account when mapping tasks to PEs.

7.6.4 Dynamic Resource Scheduling Schemes

We model the target system as an undirected graph $GT = (P, L)$, where the vertex $P_i \in P$ represents PE(i) and the edge L_{ij} represents a bidirectional communication channel between PE(i) and PE(j), for $i, j \in [1, p]$. Each PE(i) is associated with a parameter v(PE(i)) that records its functionality and a parameter cm(PE(i)) that represents its current computing mode. The weight $w(L_{ij})$ on each L_{ij} denotes its minimum communication cost. The minimum communication cost is calculated based on the minimum hops between PE(i) and PE(j). Also, communication jobs always have higher priority than computation jobs (i.e., PEs are always forced to forward incoming data even when they are busy). A priority is assigned to each task in TFG; it changes dynamically as scheduling proceeds [Kwok, et al., 1996]. This is because the dynamic assignment of PEs, the dynamic partitioning and migration of tasks to multiple PEs, and the communication pattern and cost in our policy result in changes to the critical path. If two tasks are assigned to the same PE, then the communication is removed. A task is said to be *READY* when all its inputs are available. A *QUALIFIED* PE for a task is defined as a PE that supports all the operation types $\pi(S_i)$ in the task S_i.

According to Amdahl's Law [Hwang, 1993], the speedup is limited by the sequential code. Also, increasing the number of PEs after a certain point will deteriorate the performance due to disastrous communication overheads. For any application-system pair, there is an optimal number of PEs for minimum execution time. On the other hand, the energy is the product of the power and execution time. Due to the fact that PEs consume different power in different states, chances are that this optimal number of PEs does not necessarily correspond to minimum energy consumption. Optimality involving both energy and performance depends on the task characteristics as well as the architecture. Hence, we aim to optimize across two dimensions for each task in the critical path: energy and/or performance vs. number of PEs. The number of PEs for optimal energy and performance of S_i is represented by $N_{S_i}^e$ and $N_{S_i}^t$, respectively. The corresponding energy consumption and execution time are $e_{S_i}^m$ and $t_{S_i}^m$.

In our scheduling schemes, a task can be assigned various numbers of PEs during its execution subject to the system status. For example, a task may be assigned four PEs first and more PEs will join or leave later. Note that it is impossible to know the physical locations of the PEs when carrying out this study. Therefore, we define the following two dynamic metrics to evaluate the benefits of adding more PEs to a task. Let the number of PEs assigned to S_i be $p_i(k)$ and $P^k = \sum_{i=1}^{s} p_i(k)$, where P^k is the total number of PEs in the system in clock cycle k. We define the *average remaining completion time* (ACT) and *average remaining energy cost* (AEC) for each task S_i on $p_i(k)$ PEs:

$$ACT(S_i, p_i(k)) = T_{cp}(O_i(k), p_i(k)) + T_{com}(S_i, p_i(k)) \tag{7.14}$$

$$AEC(S_i, p_i(k)) = \sum_j (O_{i,j}(k) * \frac{1}{F_i} * \frac{\sum_{l=1}^{y_j(k)} (C_{j,l} * P_{j,l})}{y_{j(k)}} \quad (7.15)$$

where $O_i(k)$ and $O_{i,j}(k)$ are the remaining numbers of the total and j_{th} FP operations in S_i, at the clock cycle k, respectively. $C_{j,l}$ and $P_{j,l}$ represent the required clock cycles and power of the l^{th} FU of the j^{th} type. $T_{cp}(O_i(k), p_i(k))$ represents the execution time for S_i with $p_i(k)$ PEs while T_{com} (from Eq. 7.13) is the communication overhead caused by distributing S_i to $p_i(k)$ PEs as compared to scheduling it on just one PE. y_j is the total number of the FUs supporting the j_{th} operation in these $p_i(k)$ PEs. Given our detailed hardware and task information, $T_{cp}(O_i(k), p_i(k))$ can be estimated very accurately. T_{com} is more variant. However, since we use the NEWS interconnect for runtime task communication, we can always find a good route for the required data transfer. This is an advantage over shared buses where conflicts may cause significant performance degradation for a large number of PEs. Also, the local memories of PEs are addressable by the sequencer. We analyze the communication patterns of each task at compile time in distributing different subtasks (i.e., loop iterations) to multiple PEs in attempts to refine T_{com}. Since ACT is calculated at runtime, we approximate the overhead.

7.6.4.1 Optimize the Performance without Energy Constraints

This scenario happens when the performance of the system is crucial and the consumed energy is not a concern. The Case-1 solution in the system synthesis phase of Section 7.5 is then applied. We focus on the TiPCs in this scenario and propose the following scheduling algorithm.

Algorithm-1

1. Wait until $P(k) \neq \varnothing$ and $S \neq \varnothing$ DO // *$P(k)$ is the set of available PEs in cycle t_k. with p_k PEs.*
2. { Calculate the priority of each task and sort S in the decreasing order of priority;
3. Reconfigure available PEs for active TiPCs whose reconfiguration requirements from synthesis have not been satisfied in the order of their age in the system;
4. IF all PEs are used, GOTO 1
5. Pick the top task $S_i \in S$;
6. IF $S_i \in$ CP
7. { IF S_i is READY // *CP: the set of TiPCs.*
8. {Reconfigure the available resources according to the synthesis.
9. Assign $\min\{p_k, N^t_{S_i}\}$ PEs to S_i;
10. Remove S_i from S to S'; } // *S': running (active) tasks in the system.*
11. ELSE
12. { $\forall S_j, j \in [1, m]$, that hold the inputs of S_i,
Add the qualified PEs $\in P(k)$ to these m tasks and repartition the tasks so as to
$ACT'(S_1, p_1) = ACT'(S_2, p_2) = \ldots = ACT'(S_j, p_j)$, // *$ACT'(S_i, p_j)$: Current ACT for S_j* based on its remaining number of operations, $j \in [1, m]$.
13. Set $cm(PE(k)) = \varepsilon(S_j)$, if PE($k$) is assigned to process S_j ;}}
14. ELSEIF $\varepsilon(S_i)$ = MIMD and READY THEN
15. { $\forall PE(j) \in P(k), j = 1, \ldots, p_k$,
16. Search for the PEs that hold the inputs of S_i;
17. IF ($\hat{p}_i \neq 0$) THEN // $\hat{p}_i$: the number of qualified PEs in $P(k)$ for S_i;
18. {Find q PEs in these $\hat{p}_i$ PEs with the minimum $ACT(S_i, q)$ and assign these q PEs to S_i;
19. $\forall PE(j), j = 1, \ldots, q$, Set $cm(PE(j)) = \varepsilon(S_i)$;}
20. ELSE
21. {Reconfigure one PE into a qualified PE for S_i;
22. Assign this PE to S_i;
23. Set $cm(PE) = \varepsilon(S_i)$;
24. Remove S_i from S to S'; }}
25. ELSEIF $\varepsilon(S_i)$ = MIMD and NOT READY THEN
26. GOTO 12;
27. ELSEIF $\varepsilon(S_i)$ = SIMD and READY THEN
28. { $\forall PE(j) \in P(k), j = 1, \ldots, p_k$,
29. Search for the PEs that hold the inputs of S_i;
30. IF ($\hat{p}_i \neq 0$) THEN // $\hat{p}_i$: the number of qualified PEs in $P(k)$ for S_i;
31. {Find q PEs in these $\hat{p}_i$ PEs with the minimum $ACT(S_i, q)$ and assign these q PEs to S_i;
32. $\forall PE(j), j = 1, \ldots, q$, Set $cm(PE(j)) = \varepsilon(S_i)$;}
33. ELSE
34. {Reconfigure $\min\{p_k, N^t_{S_i}\}$ PEs $\in P(k)$ into QUALIFIED PEs for S_i;
35. Assign these PEs to S_i;
36. Set $cm(PE(j)) = \varepsilon(S_i)$;}
37. Remove S_i from S to S'; }}
38. ELSEIF $\varepsilon(S_i)$ = SIMD and not READY THEN
39. GOTO 12;
}

During the procedure, we record and calculate the following values for each task in addition to other aforementioned statistics:

- $<p_i(k), t_k>$, where $p_i(k)$ is the number of PEs assigned to S_i in cycle t_k.

  We define the average number of PEs for S_i as:

$$p_{i,a} = \left\lceil \sum_k (p_i(k) * \frac{O_i(k)}{O_i}) \right\rceil \tag{7.16}$$

- The clock cycles spent by S_i in each FU of a PE.
- The total energy cost E_{Si}^{Σ} of S_i:

$$E_{si}^{\Sigma} = \sum_{i=1}^{p} \sum_{j=1} \sum_{k} (C_{si,i,j,k}^{active} * E_{j,k}^{active} + C_{si,i,j,k}^{idle} * E_{j,k}^{idle} + C_{si,i,j,k}^{stdby} * E_{j,k}^{stdby}) \tag{7.17}$$

  where $C_{si,i,j,k}^{active}$ is the total number of clock cycles that S_i spends on $FU_{j,k}$ in PE(i) in the active state, and so on.

7.6.4.2 Optimize the Performance with an Energy Constraint

In some cases systems are limited by power input, such as planet exploration rovers and battery-powered embedded systems. It is then desirable to maximize the performance while meeting energy constraints. For this scenario, we use the Case-2 solution in the system generation phase of Section 7.5. We first analyze the energy consumption in Senerio-1 (Section 7.6.4.1) using the energy model, and then estimate the difference between the actual consumption and its upper bound. Our system synthesis phase assures that this difference is not significant. Hence, we focus on the ToPCs during scheduling in this scenario in order not to significantly degrade the performance. The

following procedure is applied and the resulting decision table is incorporated into Algorithm-1.

Algorithm-2 */* Generate a decision table for each task, which will be checked by Algorithm-1 for Case-2*/*

1. Find out the energy gap: $E_g = E_B - \sum_{S_i}^{s} E_{S_i}^{\Sigma}$;
2. IF $E_g < 0$, Stop;
3. Calculate the total energy cost of the TiPCs, E_{TiPC}^{Σ} ;
4. Calculate the energy budget for all the ToPCs, $E_{ToPC}^{B} = E_B - E_{TiPC}^{\Sigma}$;
5. Assign an energy budget, $e_{S_i}^{B} = E_{ToPC}^{B} * \frac{O_i}{\sum_i O_i}$, to each ToPC according to its computation weight;
6. Find $< e_{Si}^{B}, N_{S_i,1}^{B} >$, $< e_{Si}^{B}, N_{S_i,2}^{B} >$ and $< e_{Si}^{m}, N_{S_i}^{e} >$ in the Energy-PE table;
7. FOR all ToPCs, DO
8. { IF $e_{Si}^{m} < e_{S_i}^{B}$
9. Assign up to $N_{S_i,2}^{B}$ PEs to S_i and put more PEs into standby when scheduling S_i; using AEC when adding a new PE to this task, if the $E_{S_i}^{\Sigma}$ exceeds e_b, do not add, else add;
10. ELSEIF $e_{Si}^{m} > e_{S_i}^{B}$
11. Assign up to $N_{S_i}^{e}$ PEs to task S_i and add $\Delta e_i = E_{S_i}^{\Sigma} - e_{S_i}^{B}$ to ΔE; } //ΔE: *the total energy mismatch.*
12. IF $\Delta E > 0$ // We have to compromise the TiPCs;
13. { Find the smallest TiPC S_i;
14. Look for a p_x such that Δe *(the energy difference between these two numbers of PEs)* >= ΔE; // $p_{S_i}^{0}$: *the number of PEs when* S_i *enters the system;* Δe*: the energy difference between* p_x *and* $p_{S_i}^{0}$.
15. Assign p_x PEs to S_i and no more PEs will be assigned to S_i; } // *Note that the unused PEs can be assigned to other tasks that are work under its local budget.*

We assume that the energy budget is reasonable; if the last iteration fails, we have to either lower the computing precision or increase the total energy budget.

7.6.4.3 Optimize the Energy Cost under a Permissible Performance Loss

When energy consumption is of paramount importance, performance can be sacrificed to an allowable extent in order to reduce the required energy. The total execution time is

largely determined by the TiPCs. Hence, we apply Case-1 in the system generation phase and focus on the TiPCs to reduce the energy consumption.

Let β be the allowed loss ratio. We decrease the performance of each task S_i in the critical path by the ratio β. We find a number p_i of PEs, with $p_i < p_{i,a}$ for an execution time determined by:

$$T_{pi} = (1+\beta) * T_{pi,a} \tag{7.18}$$

where $T_{pi,a}$ is the execution time when $p_{i,a}$ PEs are assigned to S_i. Whenever x PEs are to be added to a critical task S_i in the critical path for Algorithm-1, we apply the following algorithm.

Algorithm-3 */* Generate a decision table for each task, which will be checked by Algorithm-1 for Case-3 */*

/* Whenever x PEs are to be added to a TiPC S_i for Algorithm-1;*/
1. $m = 0$;
2. For $k = 1, x$
3. IF ACT(p_i) > T_{pi};
4. {IF ACT($p_i + k$) > ACT(p_i)
5. { $p_i = p_i + 1$;
6. $m = m + 1$;}
7. *ELSE* $p_i = p_i$;
8. }
9. For $j = m, x$
10. IF AEC($p_i + j$) < AEC(p_i) & ACT($p_i + j$) < ACT(p_i) : $p_i = p_i + 1$;
11. ELSE $p_i = p_i$;
12. Assign the p_i PEs to current task and putting the other PEs into standby;

7.7 Experimental Results

7.7.1 Singular Value Decomposition

Singular Value Decomposition (SVD) has been chosen as a computation-intensive algorithm to evaluate the performance of the design methodology. Given a matrix $A \in R^{M \times N}$, SVD factorizes A in the form $A = U\sigma V^T$, where $U = [u_1, \ldots, u_M]$ is an M x M orthogonal matrix, $V = [v_1, \ldots, v_N]$ is an N x N orthogonal matrix, and $\sigma = diag(\sigma_1, \ldots, \sigma_r)$

with $r = min(M, N)$ and $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r$ is a diagonal matrix. The σ_i's are known as the *singular values* of A, and the vectors u_i and v_i are the i^{th} *left singular vector* and *right singular vector*, respectively. The column-vectors of U and V are the normed eigenvectors of AA^T and A^TA, respectively. The singular values σ_i of matrix A can be found as the square roots of the eigenvalues of A^TA or AA^T. Hence, the computation of SVD is transformed into an *eigenvalue problem* to find all the eigenvalues λ and the corresponding eigenvectors v of matrix A by solving the equation $Av = \lambda v$. For symmetric matrices, the classic solution to this equation is the Jacobi method introduced in the 1900s; it reduces the matrix to a diagonal form via an iterative procedure. It is very slow and, hence, generally not recommended for large matrices having more than a few hundreds of rows or columns. The Golub-Kahan SVD algorithm [Golub, et. al., 1965] is the most efficient and commonly employed implementation. It employs the Householder method and the Givens reflection to reduce an N x N symmetric matrix to a bidiagonal form instead of a diagonal form by a fixed (N-2) number of iterative transformations, and then applies a QR decomposition. The advantage of SVD is that it yields a solution for any matrix. A typical implementation of SVD requires $O(MN^2)$ floating-point operations on a sequential processor [Golub, et. al., 1996].

The algorithm being studied here is based on the Golub-Kahan technique. This algorithm is abundant in nested loops and requires for square matrices more than 20 times the number of floating-point operations in LU factorization. In this experiment, a modified sequential description of the SVD algorithm in [Netlib] was analyzed and then divided into the tasks summarized in Table 7.2. As we can see, the SVD problem is very computation-intensive and is full of data dependences.

Table 7.2 SVD Task Information

Step	Tasks	Function Description	Comp. Mode	Comp. Cost	Comm. Vol.	FP Op. Types
Householder reduction (i = *1*, ..., N)	1	Calculate s	SIMD	$4M$	1	+, /, *
	2	Calculate g, h, and A_{ii}	MIMD	4	1	√, /, -, *
	3	Update A	SIMD	$4MN$	MN	+, *, /
	4	Scale A	SIMD	$2M$	M	+, *
	5	Calculate w	MIMD	1	*1*	*
	6	Calculate s	SIMD	$3(N\text{-}i\text{-}1)$	$N\text{-}i\text{-}1$	+, *, /
	7	Calculate g, h, and A_{ii}	MIMD	4	1	√, /, -,*
	8	Calculate rv1	MIMD	N	N	/
	9	Update A	SIMD	$(M\text{-}i\text{-}1)(N\text{-}i\text{-}1)^2$	MN	*, +
	10	Scale A	SIMD	$N\text{-}i\text{-}1$	N-i-1	*, +
	11	Find *max* {*anorm*, $\|w\|$ $+\|rv1\|$}	MIMD	$2^{\log N}$	1	+
Accumulation of right-hand transformations (i = 1, ..., N)	12	Calculate v	SIMD	$2(N\text{-}i\text{-}1)$	$N\text{-}i\text{-}1$	/
	13	Calculate s and v	SIMD	$4(N\text{-}i\text{-}1)^2$	$(N\text{-}i\text{-}1)^2$	+, *
Accumulation left-hand transformation (i = 1, ...,*min*(M,N))	14	Update A with s,f conditionally	M-SIMD	$2(m\text{-}i\text{-}1)$[n-i+1+$2(M\text{-}i)$] or $(M\text{-}i)$	$2MN$	+, *, /
Diagonalization of bidiagonal form (i = 1, ..., N)	15	Check *flag*	MIMD	$N\text{-}i$	$N\text{-}i$	None
	16	Update A	SIMD	$6i(M\text{-}1)$	MN	+, -, *, /
	17	Make singular values nonnegative	SIMD	N	N	*
	18	Calculate f	MIMD	20	4	+, -, *, /
	19	QR transformation	SIMD	$12(i\text{-}1)N$	$2MN$	+, -, *, /

This experiment employs a square root IP from [QinetiQ] and in-house developed single-precision FP units and the Annapolis WILDSTAR II-PCI board with two XC2V6000-5 FPGAs was used. The system was clocked at 125MHz. We first evaluated the effect of runtime reconfiguration (RTR). The chosen numbers of PEs for

the four steps in Table 7.2 were 36, 42, 42 and 42, respectively. Five randomly generated dense matrices were used; the execution times are shown in Figure 7.8. The results prove that partial dynamic RTR system reconfiguration can improve the performance significantly; the speedup increases with the matrix size. Figure 7.9 shows the performance comparison of our adaptive scheduling with an alternative dynamic scheduling where all the available PEs are assigned to each task when it is ready (fixed through the task lifetime). Our adaptive scheduling, enforced by RTR, performs much better than the naive scheduling strategy. It was observed during the experiments that adaptive scheduling greatly reduces the effect of data dependences and the idle times of PEs. It effectively shortens the critical path, which largely determines the execution time of the entire application. An increase in the number of PEs improves dramatically the execution of the largest SIMD tasks (three-nested loops), and the overheads of loop partitioning and scheduling become less significant for larger matrix sizes.

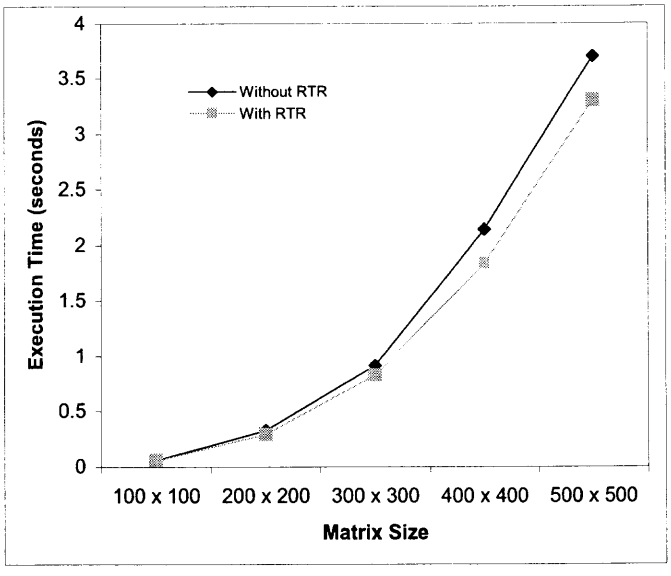


Figure 7.8 Execution times with and without partial runtime reconfiguration (RTR).

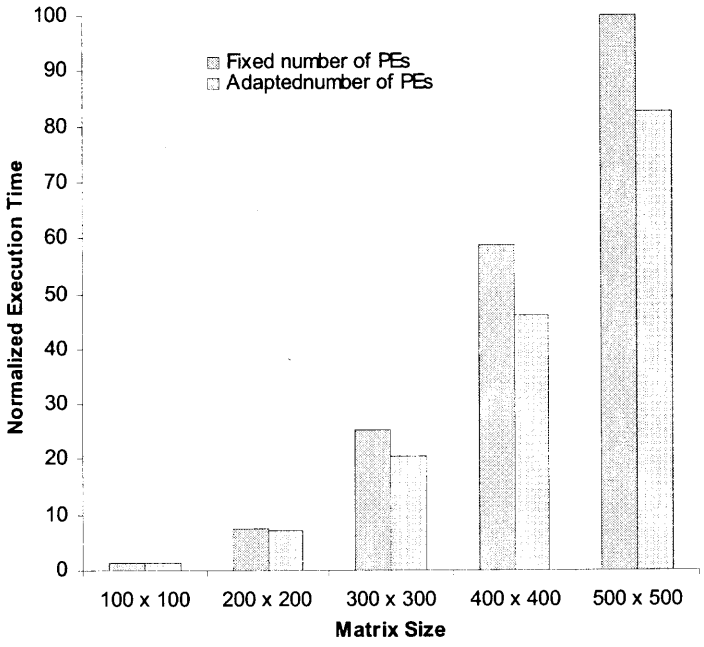


Figure 7.9 Normalized execution times for our strategy and naive dynamic scheduling.

7.7.2 Parallel Power Flow Analysis

The parallel solution of computation-intensive power flow analysis (Section 4.4) is employed in this set of experiments to evaluate system synthesis and performance-energy optimization techniques. Table 7.3 shows the task information.

Single-precision FP FUs were chosen based on the two benchmark matrices in Table 7.4. The fixed 125MHz system frequency was used in the experiments. From the task table, we can see that tasks $S_5^i(k_i)$, $S_7^i(k_i)$, $S_{10}^i(k_i)$ and $S_{11}^i(k_i)$ consume most of the execution time, especially for large matrices. The available number of PEs to these tasks has a large impact on the entire performance. We compared the performance without energy considerations for three cases: no device reconfiguration (NR), FRTR, and the optimal solution during the synthesis and scheduling phases. We assume that

the device reconfiguration time is 50 msec [Virtex II datasheet]. Table 7.5 shows the numbers of PEs available at the beginning of scheduling the tasks in the two benchmarks for various synthesis approaches. The corresponding execution times and energy consumptions are shown in Table 7.6. Note that only the available numbers of PEs are shown for each task; the actual numbers of PEs for tasks vary at run-time. We can see that the optimal solution of the synthesis procedure also depends on the size of the matrices. The 7917-bus system has a larger number of reconfigurations than the 1648-bus system for the LU and multiplication tasks because of more matrix blocks. The number of reconfigurations in HERA is reduced significantly as compared to ASPCs for large matrices because it provides instructions for the PEs. In general, FRTR provides the largest number of PEs for all the tasks. With NR, the smallest number of PEs is used for all the tasks. However, we must consider the cost of different schemes. Although we have a small number of PEs in NR, there is no reconfiguration overhead. The overhead is significant with FRTR, especially when dealing with many reconfigurations during the execution of small applications. For this reason, the performance and energy consumption of FRTR for the 1648-bus system is worse than for the other cases. FRTR cannot overlap tasks and substantial time is required to save and restore data. Also, some resources are wasted waiting for reconfiguration. In the optimal system configurations, both FRTR and PRTR are employed only when the benefits exceed the penalty; of course, partial reconfiguration should overlap computations as much as possible; tasks can be overlapped as well, so the overhead is minimized. For the 7917-bus system, the reconfiguration overhead becomes much less significant as compared to the computation time. Hence, FRTR shows a better

performance. In all the cases, the optimal configuration achieves better performance than the other approaches. With an increase in the matrix size, the amount of computation in large tasks also increases and the benefit of resource reconfiguration becomes important as a result of increased PE numbers; this is shown for the 7917-bus case.

For combined performance-energy optimization, we first evaluate the accuracy of our energy models. The 1648-bus system is used in this experiment. The results calculated from our energy model for the computation-dominant tasks are compared with the XPower results reported in Table 7.7. An average activity rate is extracted from ModelSim simulation results using Algorithm-1. The average error is about 7.6%, which is an acceptable rate for system-level estimation models. Our power data for the FUs come from implementations and the reasons for the differences are: (1) Only one power value is assumed for FUs in the active state while FUs consumes different power with different input activity rates; (2) The average activity rate varies with different sets of data for various FUs; we used a fixed rate for all data instead. (3) The measurements of the energy consumption for the bus system tend to be less accurate than for FUs due to coarse-grain power modeling. System-level models generally have a higher error rate than RTL-, logic- or gate-level models due to their simplified and less accurate parameters. However, our objective is to develop fast, yet useful models for exploring performance-energy optimizations without involving tedious and time-consuming low-level simulations.

Finally we evaluate the performance of our performance-energy optimization techniques. Table 7.8 shows the optimization results for the 7917-bus system. In

Scenario-II, we simply put idle FUs into the standby state and we reduce the total energy consumption by 10.06% without major performance penalty. This is mainly due to tasks S_6 and S_9, which cause many PEs to be idle. A performance penalty of 7.64% is observed when we reduce the energy consumption by 20%, as shown in Scenario-III. In Scenario-IV and Scenario-V, we relax the performance by 14.6% and 19.4%, and the reduction in energy consumption is 26.2% and 40.8%, respectively. The energy consumption can be further reduced by using lower-precision FP FUs.

Table 7.3 Task Information of the DBBD Power Flow Algorithm

Tasks	Description	Mode	FP Op.
$S_1(1), \ldots, S_1(n)$	Evaluate Eqs (1) and (2) and calculate $\triangle P$ and $\triangle Q$	SIMD	+, -, * cos, sin
$S_2(1), \ldots, S_2(n)$	Check individual convergence	SIMD	None
S_3	Check global convergence	MIMD	None
$S_4(1), \ldots, S_4(n)$	Construct 3-block groups $\{J_{ii}, J_{in}$ and $J_{ni}\}$	SIMD	+, -, * cos, sin
$S_5^i(1), \ldots, S_5^i(k_i)$, $i = 1, \ldots, m$	LU factorization of 3-block groups of the approximate same sizes	SIMD	+, -, *, /
$S_6(1), \ldots, S_6(n)$	LU factorization of 3-block groups of diverse sizes	MIMD	+, -, *, /
$S_7^i(1), \ldots, S_7^i(k_i)$, $i = 1, \ldots, m$	Multiplication of factored border blocks in S_5	SIMD	+, *
$S_8(1), \ldots, S_8(n)$	Multiplication of factored border blocks in S_6	MIMD	+, *
S_9	Factor the last block J_{nn}	SIMD	+, -, *, /
$S_{10}(1), \ldots, S_{10}(n)$	Forward reduction	SIMD	+, -, *
$S_{11}(1), \ldots, S_{11}(n)$	Backward substitutions	MIMD	+, -, *

n: the total number of 3-block groups in the matrix J
m: the total number of task pools that have the approximate same computing cost; each group i contains k_i 3-block groups.
q: the total number of 3-block groups with diverse computing cost

$$n = q + \sum_{i=k_1}^{k_m} m$$

Table 7.4 Optimal Partitioning of the Y_{bus} Matrices for the Benchmark Systems

Benchmark System	1648-bus	7917-bus
Dimensionality of admittance matrix (Y_{bus})	1648	7917
Dimensionality of Jacobian matrix (J)	2982	14508
Number of independent diagonal blocks	18	67
Minimum dimensionality of independent diagonal blocks	33	15
Maximum dimensionality of independent diagonal blocks	120	150
Dimensionality of the last block	134	541
Size distribution of independent diagonal blocks in Y^*	120, 109, 99, 3(90), 5(85)*, 79, 5(75), 33	7(150),17(130), 10(120), 13(100), 19(90), 1(15)

*5(85) stands for 5 blocks of approximate size 85 x 85.

Table 7.5 The Parallelism Profile (i.e., Numbers of PEs) during the Execution

Task	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
NR	24	24	24	24	24	24	24	24	24	24	24
FRTR	48	64	64	48	24	24	48	48	24	48	48
Optimal (1648)	48	48	48	24	24	24	48	48	32	32	32
Optimal (7917)	48	48	48	48	24	24	48	48	39	48	48

NR: no reconfiguration is allowed;
FRTR: Full Run-Time Reconfiguration is used in order to get a maximum number of PEs;
Optimal (1648): Optimal solution for the 1648-bus system from the synthesis procedure
Optimal (7917): Optimal solution for the 7917-bus system from the synthesis procedure

Table 7.6 Execution Times and Energy Consumptions for the Benchmark Matrices

Case		NR	FRTR	Optimal
1648-bus	Time (sec)	12.01	12.29	10.14
	Energy (J)	271.3	288.4	231.5
7917-bus	Time (sec)	391.5	369.3	315.1
	Energy (J)	10214	9630	8252

Table 7.7 Comparison between the Modeled and XPower-Reported Energy Consumption (J)

Task		Modeled	XPower-Reported
S5	1648	84	77
	7917	2526	2289
S7	1648	78	72
	7917	2391	2192
S10	1648	34	32
	7917	1030	974
S10	1648	32	30
	7917	1023	965

Table 7.8 Performance-Energy Optimization for the 7917-Bus System

Scenario	Objective	Constraints	Energy Consumption (J)	Execution Time (sec)
I	Minimize T	None	8249	314
II	Minimize T	E <7424	7419	316
III	Minimize T	E <6600	6598	338
IV	Minimize E	T <361	6087	360
V	Minimize E	T <377	4883	375

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

This research started with the observation that PC clusters and SMP multiprocessors are the dominant parallel platforms for the majority of high-performance applications while the supercomputer industry has shrank in volume since the mid-1990s. However, the shared-memory nature of SMP systems limits their scalability and the high communication latencies of cluster systems make them more effective for loosely-coupled tasks that lack frequent communications. On the other hand, with the advent of multimillion-gate FPGAs, it has become feasible to build high-performance parallel systems on reconfigurable chips. These platforms are characterized by very low cost compared to supercomputer platforms that often employ ASIC chips, and the parallel system design could match well the target applications.

Conventional FPGAs have been employed in the past primarily as application-specific coprocessors to accelerate computation-intensive algorithms. This approach is based on a development flow much like that for ASIC designs with the additional advantage of hardware reconfiguration at runtime. However, significant time and energy overheads are often required to reprogram the hardware due to not being able to fit large applications that oversize the available resources. This dissertation advocates a new philosophy in designing and implementing parallel reconfigurable systems: *Multiprocessors-on-a-Programmable Chip (*MPoPCs). The regularity and localization of MPoPC designs reduce dramatically the number of wires spanning long distances, which is a critical performance issue in million-gate chips. By employing user-

programmable PEs, instead of task-specific function units, full reconfiguration of the hardware at runtime for large applications is eliminated. Also, the general-purpose user instructions for MPoPCs alleviate the burden of complex hardware design and, hence, bring FPGA-based parallel processing closer to mainstream computing. At the same time, existing research results in parallel processing could be exploited for the new platforms.

To provide credible and reliable references for further study and also discover real issues with MPoPCs, two MPoPCs were with pipelined hardware FPUs designed and implemented based on the above philosophy. Previous FPGA-based custom computing machines did not implement hardware FPUs due to the insufficient resources in conventional FPGAs. Our MPoPCs target matrix-based data parallel applications that require floating-point representations. The first MPoPC implementation, CG-MPoPC, is based on a configurable IP processor core from Altera (i.e., Nios) and has been implemented on the Altera SOPC FPGA development board. It is designed to run in the MIMD mode, contains more instructions than HERA, the second MPoPC implementation, and is capable of performing well for more diverse applications. HERA on the other hand, concentrates on mixed-mode parallelism and has been implemented on Xilinx FPGAs. It is equipped with more fine–grain communication channels and is optimized for variant-grain matrix-based applications. The PE/node was designed with an emphasis on matrix operations. The system can be reconfigured dynamically at runtime to support a variety of independent or cooperating computing modes, such as SIMD, MIMD and M-SIMD, to best match in the time spectrum all the subtask needs of a given application. Both MPoPC systems are user-programmable with

general-purpose instructions. Their PEs are equipped with large data and instruction on-chip memories. These platforms feature much lower cost and risk compared to ASIC-based multiprocessors. Parallel solutions for representative computation-intensive benchmark algorithms, namely matrix-matrix multiplication (MMM), LU factorization and power flow analysis, were developed and realized on the two MPoPCs. Application mapping and dynamic load balancing schemes were proposed and analyzed as well. The performance of both systems was tested with large sparse, real-world matrices from power engineering; they have size of up to 10279 x 10279. The results are better than those of two commercial PCs. We expect even more dramatic performance gains in the near future by employing ever improving FPGAs.

The success of such systems will highly depend on high-level design tools to efficiently map applications onto the reconfigurable logic and remove the hardware details from the end users. The seminal quality of MPoPCs is that we can semi-customize a personal system in the field for any given application to match better its computation and communication characteristics. An architecture-conscious resource management framework was proposed and implemented to efficiently map data-parallel applications onto HERA; the applications are depicted in a high-level functional form. A two-phase approach is used. A mixed-mode weighted Task Flow Graph (w-TFG) is constructed for the application, where tasks are classified according to their appropriate computing mode, i.e., SIMD or MIMD. At compile time, an MPoPC is synthesized under various performance and energy constraints for the given TFG; an Integer Linear Programming (ILP) technique that uses a parameterized hardware component library is applied. While it is always desirable to maximize performance, energy consumption has

emerged as a first-order design constraint, especially for embedded systems. A hybrid system-level energy model for HERA is proposed to guide run-time architectural and software decisions. Various dynamic scheduling schemes under different performance-energy objectives are proposed. Partial run-time reconfiguration can be employed to further increase resource utilization. A parallel power flow analysis technique by Newton's method is proposed and employed to verify the methodology and evaluate the performance.

8.2 Future Research

High-performance low-cost reconfigurable computing is an emerging new area and many problems are still unexplored. Continuation of this work could focus on the following fundamental issues:

1. ***Innovative FPGA circuits to reduce the device reconfiguration overhead in terms of time and energy, and facilitate the synthesis of various system-level architectures.***

   The reconfiguration overhead has been a critical obstacle to SRAM-based FPGAs since ever their birth and will continue to be a must-solve problem if we want FPGA-based reconfigurable systems to enter mainstream computing. Although relevant research has shown promising results, it has failed to appeal to FPGA vendors.

2. ***High-performance reconfigurable architectures and design methodologies***

   Reconfigurable logic provides tremendous flexibility. Research in this area should not isolate the levels, that is, the circuit, gate, logic and system levels. A good

integration effort involving various levels will result in significant improvements for reconfigurable computing.

3. ***Energy modeling for energy-efficient/aware design and scheduling techniques targeting coarse-grain architectures***

   Power consumption has become a very critical issue in the design of many computing systems involving current CMOS processes. A fundamental breakthrough in device manufacturing processes is required to save the computing industry in the long term. This is also true of FPGA-based platforms.

4. ***Efficient coupling and communication schemes between reconfigurable fabric and general-purpose platforms.***

   Before we see a breakthrough in reconfigurable computing, the reconfigurable fabric will have to work efficiently with general-purpose processors. The interface between the two is another critical problem that may reduce substantially any performance gains. Data prefetching, operation overlapping and intelligent memory designs could potentially provide viable solutions.

Also, MPoPCs could aid network applications, such as intrusion detection and packet forwarding, which have diverse performance and energy metrics. Bioinformatics is another field that could benefit from the massive parallelism potentially provided by FPGAs.

BIBLIOGRAPHY

Adam, T. L., Chandy, K. M., and Dickson, J. R. (1974). *A comparison of list schedules for parallel processing systems.* Communications of the ACM, Vol.17, No.12, pp. 685-690.

Ahmedsaid, A., Amira, A., and Bouridane, A. (2003). *Improved SVD systolic array and implementation on FPGA.* IEEE International Conference on Field-Programmable Technology (FPT), pp. 35-42.

Altera. http://www.altera.com.

Amestoy, P. R., Davis, T. A., and Duff, I. S. (1996). *An approximate minimum degree ordering algorithm.* SIAM Journal on Matrix Analysis and Applications, Vol. 17, No. 4, pp. 886-905.

Anderson, J. H., and Najm, F. N. (2004). *Power estimation techniques for FPGAs.* IEEE Transactions on VLSI Systems, Vol. 12, No. 10, pp. 1015-1027.

AnnapMicro (Annapolis Micro Systems, Inc.). http://www.annapmicro.com/.

Bailey, D. H. (1998). Challenges of future high-end computing, in High Performance Computer Systems and Applications. Jonathan Schaeffer (ed.), Kluwer Academic Press, Boston.

Baker, J. M. Jr., Bennett, S., Bucciero, M., Gold, B., and Mahajan, R. (2002). *SCMP: a single-chip message-passing parallel computer.* International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02), Las Vegas, NV, pp.1485-1491.

Barroso, L., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Piranha, B. V. (2000). *A scalable architecture based on single-chip multiprocessing.* IEEE International Symposium on Computer Architecture (ISCA'00).

Becker, J. and Vorbach, M. (2003). *Architecture, memory and interface technology integration of an industrial/academic configurable System-on-Chip (CSoC).* IEEE Symposium on VLSI, pp. 107-112.

Bell, G., and Gray, J. (2002). *What's next in high-performance computing?* Communications of the ACM, Vol. 45, No. 2, pp. 91-95.

Benini, L., and Micheli, G. De. (1999). *System-level power optimization: techniques and tools.* IEEE International Symposium on Low Power Electronics and Design, pp. 288-293.

Benini, L., Bogliolo, A., and Micheli, G. De (2000). *A survey of design techniques for system-level dynamic power management.* IEEE Transactions on VLSI Systems, Vol. 8, No. 3, pp. 299-316.

Bergamaschi, R., Bolsens, I., Gupta, R., Harr, R., Jerraya, A., Keutzer, K., Olukotun, K., and Vissers, K. (2001). *Are single-chip multiprocessors in reach?* IEEE Design and Test of Computers, Vol. 18, No.1, pp. 82-89.

Bischof, C. (1987). *The two-sided Jacobi method on a hypercube.* SIAM Conference on Hypercube Multiprocessors, pp. 612-618.

Bomhof, W., and van der Vorst, H. A. (2000). *A parallel linear system solver for circuit simulation problems.* Numerical Linear Algebra Applications, Vol. 7, pp. 649-665.

Bondalapati, K., and Prasanna, V.K. (2002). *Reconfigurable computing systems.* Proceedings of the IEEE, Vol. 90, No. 7, pp.1201-1217.

Brent, R. P., and LulG, F. T. (1985(1)). *The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays.* SIAM Journal on Scientific and Statistical Computing, Vol. 6, No. 1, pp. 69-84.

Brent, R. P., Luk, F. T., and Van Loan, C. F. (1985(2)). *Computation of the singular value decomposition using mesh-connected processors.* Journal of VLSI Computer Systems, Vol. 1, No. 3, pp. 243-270.

Brooks, D., Tiwari, V., and Martonosi, M. (2000). *Wattch: A framework for architectural-level power analysis and optimization.* IEEE International Symposium on Computer Architecture (ISCA00), pp. 83-94.

Cannon, L. E. (1969). A cellular computer to implement the kalman filter algorithm. PhD Thesis, Montana State University.

Cardoso, J. M. P. (2003). *On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures.* IEEE Transactions on Computers, Vol. 52, No. 10, pp. 1362-1375.

Cardoso, J. M. P., and Neto, H. C. (2003). *Compilation for FPGA-based reconfigurable hardware.* IEEE Design and Test of Computers, Vol. 20, No.2, pp. 65-75.

Cesario, W. O., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., Jerraya, A. A., Gauthier, L., and Diaz-Nava, M. (2002). *Multiprocessor SoC platforms: a component-based design approach.* IEEE Design and Test of Computers, Vol. 19, No. 6, pp. 52-63.

Chen, S. D., Chen, J. F. (2005). *A novel approach based on global positioning system for parallel load flow analysis.* International Journal of Electrical Power and Energy Systems, Vol. 27, No.1, pp. 53-59.

Cierniak M., Li, W., and Zaki, M., J. (1995). *Loop scheduling for heterogeneity.* 4th IEEE International Symposium on High-Performance Distributed Computing (HPDC), Pentagon City, Virginia, pp. 78-85.

Compton, K., and Hauck, S. (2002). *Reconfigurable computing: a survey of systems and software.* ACM Computating Surveys, Vol. 34, No.2, pp. 171-210.

DeHon, A. (1997). *Multicontext field-programmable gate arrays.* http://www.cs.caltech.edu/courses/cs184/winter2003/reading/dpga_preprint.ps.

Demmel, J. W., Gilbert, J. R., Li, X. S. (1999). *An asynchronous parallel supernodal algorithm for sparse gaussian elimination.* SIAM Journal on Matrix Analysis and Applications, Vol. 20, No.4, pp. 915-952.

Duff, I. S. (1998). *Direct methods.* Technical Report RAL-98-056, Rutherford Appleton Laboratory, Oxfordshire, UK.

Duff, I. S., Erisman, A. M., and Reid, J. K. (1990). Direct methods for sparse matrices. Oxford Univ. Press, Oxford, England.

Ebeling, C., Cronquist, D. C., and Franklin, P. (1996). *RaPiD-reconfigurable pipelined datapath.* International Workshop on Field-Programmable Logic and Applications, pp. 126-135.

Falcao, D. M. (1996). *High performance computing in power system applications.* International Meeting on Vector Parallel Processing (VECPAR'96), Porto, Portugal.

Fox, G., Salmon, J., Otto, S., Lyzenga, G., and Johnson, M. (1988). Solving Problems on Concurrent Processors, Vol. 1: General Techniques and Regular Problems. Prentice-Hall, New Jersey.

Fu, C., Jiao, X., and Yang, T. (1998). *Efficient sparse LU factorization with partial pivoting on distributed memory architectures*. IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 2, pp. 109-125.

George, A. and Liu, J. (1989). *The evolution of the minimum degree ordering algorithm.* SIAM Review, Vol. 31, pp. 1-19.

Gerasoulis A., and Yang, T. (1992). *A comparison of clustering heuristics for scheduling DAGs on multiprocessors.* Journal of Parallel and Distributed Computing, Vol. 16, No. 4, pp. 276-291.

Ghiasi, S., Nahapetian, A., and Sarrafzadeh, M. (2004). *An optimal algorithm for minimizing runtime reconfiguration delay.* ACM Transactions on Embedded Computing Systems, Vol. 3, No. 2, pp. 237-256.

Gokhale, M., Stone, J. M., Arnold, J., and Kalinowski, M. (2000). *Stream-oriented FPGA computing in the streams-C high level language.* IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 49-58.

Golub, G. H., and Kahan, W. (1965) *Calculating the singular values and pseudoinverse of a matrix.* SIAM Journal on Numerical Analysis, Vol.2, No.3, pp. 205-224.

Golub, G. H., and van Loan, C. F. (1996). Matrix Computations (3rd edition). Johns Hopkins.

Govindu, G., Choi, S., Prasanna, V. K., Daga, V., Gangadharpalli, S., and Sridhar, V. (2004). *A high-performance and energy-efficient architecture for floating-point based LU decomposition on FPGAs.* Reconfigurable Architectures Workshop (RAW).

Grajcar, M. (2001). *Strengths and weaknesses of genetic list scheduling for heterogeneous systems.* International Conference on Application of Concurrency to System Design, pp. 123-132.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). Introduction to Parallel Computing (2nd Edition). Addison Wesley.

Gupta, A. (2002). *Recent advances in direct methods for solving unsymmetric sparse systems of linear equations.* ACM Transactions on Mathematical Software, Vol. 28, No.3, pp. 301-324.

Gupta, R. (1992). *Synchronization and communication costs of loop partitioning on shared-memory multiprocessor systems.* IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 4, pp. 505-512.

Hammond, L., Hubbert, B., Siu, M., Prabhu, M., Chen M., and Olukotun, K. (2000). *The Stanford Hydra CMP. IEEE MICRO*, March-April 2000.

Henkel, J., Wolf, W., and Chakradhar, S. (2004). *On-chip networks: a scalable, communication-centric embedded system design paradigm.* International Conference on VLSI Design, pp. 845-851.

Hillis, W. D., and Steele, G. L. (1986). *Data parallel algorithms.* Communications of the ACM, Vol. 29, No. 12, pp. 1170-1183.

Hoare, R., Tung, S., and Werger, K. (2004). *An 88-way multiprocessor within an FPGA with customizable instructions.* IEEE International Parallel and Distributed Processing Symposium (IPDPS).

Hofstee, H. P. (2005). *Power efficient processor architecture and the cell processor.* International Symposium on High-Performance Computer Architecture (HPCA-11), pp. 258 –262.

Hung, A., W. Bishop, D., and Kennings, A. (2005). *Symmetric multi-processing on programmable chips made easy*. ACM/IEEE Design Automation and Test in Europe (DATE-2005).

Hwang, K. (1993). Advanced Computer Architecture: Parallelism, Scalability, Programmability. McGraw Hill.

IEEE Committee Report. *Parallel processing in power systems computation*. IEEE Transactions on Power Systems, Vol. 7, No. 2, pp. 629-638.

Jerraya, A., and Wolf, W. (eds.) (2004). Multiprocessor Systems-on-Chips. Morgan Kaufman.

Jin, Y., Satish, N., Ravindran, K., and Keutzer, K. (2005). *An automated exploration framework for FPGA-based soft multiprocessor systems*. IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES).

Kadayif, I., Kandemir, M., Chen, G., Ozturk, O., Karakoy, M., and Sezer, U. (2005). *Optimizing array-intensive applications for on-chip multiprocessors*. IEEE Transactions on Parallel and Distributed Systems, Vol. 16, No. 5, May 2005, pp. 396-411.

Koester, D. P., Ranka, S., and Fox G. C. (1994). *Parallel block-diagonal-bordered sparse linear solvers for electrical power system applications*. IEEE Scalable Parallel Libraries Conference.

Krashinsky, R., Batten, C., Hampton, M., Gerding, S., Pharris, B., Casper, J., and Asanovic, K. (2004). *The vector-thread architecture*. IEEE International Symposium on Computer Architecture (ISCA-31), Munich, Germany.

Kuck, D. J. (1996). *What is good parallel performance? And how do we get it?* IEEE Computational Science and Engineering, Vol. 3, No. 1, pp. 81-85.

Kwok, Y., and Ahmad, I. (1996). *Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors*. IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 5, pp. 506-521.

Kwok, Y., and Ahmad, I. (1999). *Static scheduling algorithms for allocating directed task graphs to multiprocessors*. ACM Computing Surveys, Vol. 31, No. 4, pp. 406-471.

Lan, Z., and Deshikachar, P. (2003). *Performance analysis of large-scale cosmology application on three cluster systems*. IEEE International Conference on Cluster Computing, pp. 56-63.

Li, F., Lin, Y., He, L., Chen, D., and Cong, J. (2005). *Power modeling and characteristics of Field Programmable Gate Arrays*. IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 24, No. 11, pp. 1712-1724.

Li, Z., and Hauck, S. (2001). *Configuration compression for Virtex FPGAs.* IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), pp. 147-159.

Liang, J., Tessier, R., and Mencer, O. (2003). *Floating point unit generation and evaluation for FPGAs.* IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03), pp.185-194.

Ling, X. P., Amano, H. (1993). *WASMII: a data driven computer on a virtual hardware.* IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93), pp. 33-42.

Loghi, M., Poncino, M., and Benini, L. (2004). *Cycle-accurate power analysis for multiprocessor systems-on-a-chip.* ACM Great Lakes Symposium on VLSI, pp. 401-406.

Marculescu, D. and Iyer, A. (2001). *Application-driven processor design exploration for power-performance trade-off analysis.* IEEE/ACM International Conference on Computer Aided Design, pp. 306-313.

Matrix Market. *http://math.nist.gov/MatrixMarket/.*

McCreary, C. L., Khan, A. A., Thompson, J. J., and McArdle, M. E. (1994). *A comparison of heuristics for scheduling DAGs on multiprocessors.* International Parallel Processing Symposium, pp. 446-451.

Meilander, W. C., Baker, J. W., and Jin, M. (2003). *Importance of SIMD computation reconsidered.* IEEE International Parallel and Distributed Processing Symposium (IPDPS2003), pp. 266-273.

Meyer, B. H., Pieper, J. J., Paul, J. M., Nelson, J. E., Pieper, S. M., and Rowe, A. G. (2005). *Power-performance simulation and design strategies for single-chip heterogeneous multiprocessors.* IEEE Transactions on Computers, Vol. 54, No.6, pp. 684-697.

Mirsky, E., and DeHon, A. (1996). *MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources.* IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'96), pp.157-166.

Miyamori, T., and Olukotun, K. (1998). *REMARC: reconfigurable multimedia array coprocessor.* ACM/SIGDA International Symposium on Field Programmable Gate Arrays.

Najjar, W., Bohm, W., Draper, B., Hammes, J., Rinker, R., Beveridge, R., Chawathe, M., and Ross, C. (2003). *From algorithms to hardware–a high-level language abstraction for reconfigurable computing.* IEEE Computer, Vol. 36, No. 8, pp. 63-69.

Netlib. http://www.netlib.org/.

Nollet, V., Marescaux, T., Avasare, P., and Mignolet, J.-Y. (2005). *Centralized run-time resource management in a Network-on-Chip containing reconfigurable hardware Tiles.* IEEE ACM/IEEE Design Automation and Test in Europe (DATE2005), pp. 234-239.

OGSA-WG, https://forge.gridforum.org/projects/ogsa-wg.

Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., and Chang, K. (1996). *The case for a single-chip multiprocessor.* International Symposium Architectural Support for Programming Languages and Operating Systems, pp. 2-11.

Ou, J. and Prasanna, V. K. (2004). *Rapid energy estimation of computations on FPGA based soft processors.* IEEE International SoC Conference (SOCC), Sept. 2004.

Pan, J. H., Mitra, T., and Wong, W. F. (2004). *Configuration bitstream compression for dynamically reconfigurable FPGAs.* IEEE International Conference on Computer Aided Design (ICCAD), Nov. 2004, pp. 766-773.

Parhami, B. (1995). *SIMD machines: do they have a significant future?* Report on a Panel Discussion, Symposium on the Frontiers of Massively Parallel Computation, McLean, LA.

Pinedo, M. (2002). Scheduling: Theory, Algorithms, and Systems (2nd Edition). Prentice Hall.

Polychronopoulos, C. D., and Kuck, D. J. (1987). *Guided self-scheduling: a practical scheduling scheme for parallel supercomputers.* IEEE Transactions on Computers, Vol. 36, No. 12, pp. 1425-1439.

Polychronopoulos, C. D., Kuck, D. J., and Padua, D. A. (1989). *Utilizing multidimensional loop parallelism on large scale parallel processor systems.* IEEE Transactions on Computers, Vol. 38, No.9, pp.1285-1296.

Power4, www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html.

Prasanna, V. K. (2005). *Energy-efficient computations on FPGAs.* Journal of Supercomputing. Vol. 32, No.2, pp. 139-162.

QinetiQ. http://www.quixilica.com.

Raghunathan, A., Jha, N. K., and Dey, S.(1998). High-Level Power Analysis and Optimization. Kluwer Academic Publishers.

Rauchwerger, L., and Padua, D. A. (1999). *The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization.* IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 2, pp. 160-180.

Ravindran, K., Satish, N., Jin, Y., and Keutzer, K. (2005). *An FPGA-based soft multiprocessor system for IPv4 packet forwarding.* International Conference on Field Programmable Logic and Applications, pp. 487-492.

Reed, D.A. (2003). *Grids, the TeraGrid and beyond.* IEEE Computer, Vol. 36, No. 1, pp. 62-68.

Ronen, R., Mendelson, A., Lai, K., Lu, S-L., Pollack, F., and Shen, J. (2001). *Coming challenges in microarchitecture and architecture.* Proceedings of the IEEE, Vol. 89, No.3, pp.325-340.

Salminen, E., Kulmala, A., and Hamalainen, T. D. (2005). *HIBI-based multiprocessor SoC on FPGA.* IEEE International Symposium on Circuits and Systems, pp. 3351-3354.

Sangiovanni-Vincentelli, A., Chen, L. K., and Chua, L. O. (1977). *An efficient heuristic cluster algorithm for tearing large-scale networks.* IEEE Transactions on Circuits and Systems, Vol. 24, No.12, pp. 709-717.

Scalera, M. S., Murray, J. J., and Lease, S. (1998). *A mathematical benefit analysis of context switching reconfigurable computing.* Reconfigurable Architecture Workshop (Lecture Notes in Computer Science, No. 1388), pp. 73-78.

Shang, L., and Jha, N. K. (2001). *High-level power modeling of CPLDs and FPGAs.* IEEE International Conference on Computer Design (ICCD), pp. 46-51.

Shang, L., Kaviani, A. S., and K. Bathala (2002). *Dynamic power consumption in Virtex™-II FPGA family.* ACM/SIGDA International Symposium on Field-programmable Gate Arrays, pp. 157-164.

Siegel , H. J., Braun, T. D., Dietz, H. G., Kulaczewski, M. B., Maheswaran, M., Pero, P., Siegel, J. M., So, J. J. E., Tan M., Theys, M. D., and Wang, L. (1996*). The PASM project: a study of reconfigurable parallel computing.* International Symposium on Parallel Architectures, Algorithms, and Networks, pp. 529-536.

Siegel, H. J., Maheswaran, M., Watson, D.W., Antonio, J. K., and Atallah, M. J. (1996). Mixed-Mode System Heterogeneous Computing in Heterogeneous Computing, M. M. Eshaghian (Ed.), Artech House, Norwood, MA, pp. 19-65.

Simon, H.D. (2003). *The divergence problem.* International Supercomputer Conference (ISC2003), Heidelberg, Germany.

Singh, H., Lee, M.-H., Lu, G., Kurdahi, F. J., Bagherzadeh, N., and Filho, E. M. C. (2000). *MorphoSys: an integrated reconfigurable system for data-parallel and computation-Intensive Applications.* IEEE Transactions on Computers, Vol. 49, No. 5, pp. 465-481.

Sinha, A., and Chandrakasan, A. P. (2001). *JouleTrack - A web based tool for software energy profiling.* IEEE Design Automation Conference.

Srinivasan, K., and Chatha, K. S. (2004). *An ILP formulation for system level throughput and power optimization in multiprocessor SoC architectures.* IEEE International Conference on VLSI Design, pp. 255-260.

Stolberg, H.-J. Berekovic, M., L. Friebe, Moch, S., Kulaczewski, M. B., Flugel, S., Klusmann, H., Dehnhardt, A., and Pirsch, P. (2005). *HiBRID-SoC: a multi-core SoC architecture for multimedia signal processing.* Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, Vol. 41, No. 1, pp. 9-20.

Sun, F., Ravi, S., Raghunathan, A., and Jha, N. K. (2005). *Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors.* IEEE International Conference on VLSI Design, pp. 551-556.

Taylor, M. Kim, B., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, Hoffmann, B., Johnson, H., P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Frank, V. S. M., Amarasinghe, S. and Agarwal, A. (2002). *The RAW microprocessor: a computational fabric for software circuits and general purpose programs.* IEEE Micro.

Tessier, R., and Burleson, W. (2001) *Reconfigurable computing and digital signal processing: a survey.* Journal of VLSI Signal Processing, pp. 7-27.

Theys, M. D., Braun, T. D., and Siegel, H. J. (1998). *Widespread acceptance of general-purpose, large-scale parallel machines: fact, future, or fantasy?* IEEE Concurruncy, Vol. 6, No.1, pp. 79-83.

Tinney, W. F., and Hart, C. E. (1967). *Power flow solution by Newton's method.* IEEE Transactions on Power Apparatus and Systems, Vol. PAS-86, No. 3, pp. 1146-1152.

Tiwari, V., S. Malik, and Wolfe, A. (1994). *Power analysis of embedded software: A first step towards software power minimization.* IEEE Transactions on VLSI Systems, Vol. 2, No.4, pp.437-445.

TMS320C6711/11B/11C/11D floating-point digital signal processors, http://focus.ti.com/docs/prod/folders/print/tms320c6711.html.

TOP500, www.top500.org.

Tosic, P.T. (2004). *A perspective on the future of massively parallel computing: fine-grain vs. coarse-grain parallel models comparison and contrast.* 1st Conference on Computing Frontiers, Ischia, Italy, pp. 488-502.

Underwood, K. (2004). *FPGAs vs. CPUs: Trends in peak floating-point performance.* ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, pp. 171-180.

Vaughan-Nichols, S. J. (2004). *New trends revive supercomputing industry.* IEEE Computer, Vol. 37, No. 2, pp. 10-13.

Venkataramani, G., Najjar, W., Kurdahi, F., Bagherzadeh, N., Bohm, W., and Hammes, J. (2003). *Automatic compilation to a coarse-grained reconfigurable system-on-chip.* ACM Transactions on Embedded Computing Systems, Vol. 2, No.4, pp. 560-589.

Wang, X., and Ziavras, S. G. (2005-1). *Exploiting mixed-mode parallelism for matrix operations on the HERA architecture through reconfiguration.* IEE Proceedings, Computers and Digital Techniques, accepted.

Wang, X., and Ziavras, S. G. (2005-2). *A framework for dynamic resource management and scheduling on reconfigurable mixed-mode multiprocessor.* IEEE International Conference on Field-Programmable Technology (FPT'05), Singapore.

Wang, X., and Ziavras, S. G. (2005-3). *Adaptive scheduling of array-intensive applications on mixed-mode reconfigurable multiprocessors.* IEEE Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California.

Wang, X., and Ziavras, S. G. (2004-1). *Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines.* Concurrency and Computation: Practice and Experience, Vol. 16, No. 4, pp. 319-343.

Wang, X., and Ziavras, S. G. (2004-2). *Mixed-mode scheduling for parallel LU factorization of sparse matrices on the reconfigurable HERA computer.* International Conference on Advances in Computer Science and Technology (ACST 2004), St. Thomas, U.S. Virgin Islands, Nov. 22-24, 2004.

Wang, X., and Ziavras, S. G. (2004-3). *HERA: A reconfigurable and mixed-mode parallel computing engine on platform FPGAs.* International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), MIT, Cambridge, MA.

Wang, X., and Ziavras, S. G. (2004-4). *A configurable multiprocessor and dynamic load balancing for parallel LU factorization.* IEEE Workshop on Parallel and Distributed Scientific and Engineering (in conjunction with the 18th International Parallel and Distributed Processing Symposium-IPDPS2004), Santa Fe, New Mexico.

Wang, X., and Ziavras, S. G. (2003-1). *Performance optimization of an FPGA-based configurable multiprocessor for matrix operations.* IEEE International Conference on Field-Programmable Technology (FPT'03), Tokyo, Japan.

Wang, X., and Ziavras, S. G. (2003-2). *Parallel direct solution of linear equations on FPGA-based machines.* IEEE International Workshop on Parallel and Distributed Real-Time Systems (in conjunction with the IEEE International Parallel and Distributed Processing Symposium-IPDPS2003), Nice, France.

Wang, X., Ziavras, S. G and Savir, J. (2003-3). *Efficient LU factorization on FPGA-based machines.* IASTED International Multi-Conference on Power and Energy Systems, Palm Springs, California.

Watson, D., Siegel, H. J., Antonio, J. K., Nichols, M. A., and Atallah, M. J. (1994). *A block-based mode selection model for SIMD/SPMD parallel environments.* Journal of Parallel and Distributed Computing, Vol. 21, No. 3, pp. 271-287.

Wirthlin, M. J., and Hutchings, B. L. (1996). *Sequencing run-time reconfigured hardware with software.* ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp.122-128.

Wolf, W. (2004) *The future of multiprocessor systems-on-chips.* IEEE Design Automation Conference, pp. 681-685.

Xilinx Power Tools (2003): *The power estimator*, XAPP152 (Ver. 2.1).

Xilinx Virtex II datasheet. http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

Xilinx. http://www.xilinx.com.

Ye, W., Vijaykrishnan, N., Kandemir, M., and Irwin, M. J. (2000). *The design and use of SimplePower: A cycle-accurate energy estimation tool.* IEEE Design Automation Conference, pp. 340-345.

Ye, Z. A., Moshovos, A., Hauck, S., and Banerjee, P. (2000) *CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit.* IEEE International Symposium on Computer Architecture (ISCA), pp. 225-235.

Zhu, D., Melhem, R., and Childers, B. (2003). *Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems.* IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 7, pp. 686-700.

Zhuo, L., and Prasanna, V. K. (2004). *Scalable and modular algorithms for floating-point matrix multiplication on FPGAs.* IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 92-101.

Ziavras, S. G. (1993). *Efficient mapping algorithms for a class of hierarchical systems.* IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 11, pp. 1230-1245.