Computing with Memory for Energy-Efficient Robust Systems

by

Somnath Paul

Submitted in partial fulfillment of the requirements

For the degree of Doctor of Philosophy

Thesis Adviser: Dr. Swarup Bhunia

Department of Electrical Engineering and Computer Science

Case Western Reserve University

August 2011

CASE WESTERN RESERVE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

candidate for the _____ degree *.

(signed) _____
(chair of the committee)

(date) _____

*We also certify that written approval has been obtained for any
proprietary material contained therein.

To my Parents . . .

Contents

List of Figures	vii
List of Tables	xi
Acknowledgements	xiv
Abstract	xv
1 Introduction	1
1.1 Research objectives	3
1.2 Contributions	4
1.3 Thesis organization	6
2 Background & motivation	9
2.1 Background	9
2.1.1 Reconfigurable computing - state of the art	10
2.1.2 Challenges in reconfigurable system design	20
2.1.3 Challenges in general computing systems	24
2.2 Motivation	29
2.2.1 A summary of requirements	29
2.2.2 Hardware acceleration	30
2.2.3 Bandwidth is not scaling	30

2.2.4	Opportunities provided by emerging technologies	31
2.2.5	Proposed solution	34
3	Memory based computing - overview	36
3.1	Computing in memory versus computing with memory	37
3.2	Overview of hardware and software architectures	40
3.2.1	Distinction with existing hardware frameworks	42
3.3	Application of MBC	45
3.3.1	Motifs	46
3.3.2	Contexts	47
3.3.3	Domains	48
4	Hardware framework	49
4.1	A memory based generic reconfigurable framework	50
4.1.1	μ -architecture for memory logic block	51
4.1.2	Sequence of operations inside MLB	59
4.1.3	Inter-MLB communication framework	60
4.1.4	Estimation of MLB cycle time	61
4.2	MAHA hardware architecture	62
4.2.1	Existing memory organization	62
4.2.2	Instrumenting memory for hardware acceleration	64
5	Software framework	74
5.1	Details of the software flow	78
5.1.1	CDFG format	78
5.1.2	Decomposition	81
5.1.3	Fusion	82
5.1.4	Resource-aware scheduling	83
5.1.5	Packing of partitions to multi-MLB framework	84
5.1.6	Placement	88
5.1.7	Report generation	93

5.1.8	Bitfile generation and functional validation	97
5.1.9	Design space exploration	97
6	MBC design space exploration	100
6.1	Design space exploration for MBC based stand-alone reconfigurable computing	101
6.1.1	Benchmarks and simulation setup	102
6.1.2	MLB architecture exploration	104
6.1.3	Simulation results	107
6.2	Design space exploration for MAHA framework	112
6.2.1	Benchmarks and simulation setup	113
6.2.2	MLB architecture exploration	114
6.2.3	Variation in energy-efficiency with varying MLB design parameters	117
6.3	Optimization in application mapping	119
6.3.1	Content-aware mapping	119
6.4	Preferential memory design for MBC frameworks	122
6.4.1	Case study - I : SRAM array	122
6.4.2	Case study - II : STTRAM array	127
6.4.3	Improvement in EDP for STTRAM based MBC	129
6.5	PLA based application mapping in MBC	131
6.5.1	Function representation using CAM based MBC framework	132
6.5.2	Simulation results	134
7	MBC using on-chip cache	137
7.1	Reliability improvement	137
7.1.1	Overall scheme	139
7.1.2	Cache and memory organization	143
7.1.3	Memory based addition procedure	144
7.2	Simulation results	146
7.2.1	Test setup	146

7.2.2	Tolerance to thermal stress	147
7.2.3	Tolerance to manufacturing defects	147
7.2.4	Overhead in MBC scheme	150
7.2.5	Yield improvement results	150
7.2.6	Use of result CAM	152
8	Improving the reliability of operations in MBC frameworks	154
8.1	Mitigating the effect of parametric failures in MBC frameworks .	154
8.1.1	Effect of Parameter Variations on QoS	155
8.1.2	A variation-aware preferential design approach	159
8.1.3	Preferential mapping	161
8.1.4	Post-Si generation of Reliability Map	161
8.1.5	Impact of Preferential Mapping on Output Quality	162
8.1.6	Preferential Mapping Algorithm Under Delay Constraint .	163
8.1.7	Preferential memory design	166
8.1.8	Preferential sizing of 6-T SRAM	166
8.1.9	Compensation for Increased Write Failure Probability . . .	168
8.1.10	Impact of cell sizing on output quality	169
8.2	Mitigating the effect of runtime-failures in MBC frameworks . . .	171
8.2.1	Impact of parameter variation on runtime failures	172
8.2.2	Reliability-driven ECC allocation for multiple bit error re- silience	175
8.2.3	Simulation setup and results	179
8.2.4	Tolerance to soft-error induced contiguous errors	180
8.2.5	A unified solution to tolerate random and contiguous errors	182
8.2.6	Energy saving with dynamic voltage scaling	184
9	Off-chip hardware acceleration using MAHA framework	186
9.1	Background and motivation	187
9.1.1	Von-Neumann bottleneck	187
9.1.2	Existing non-Von-Neumann architectures	190

CONTENTS

9.1.3	Why do we another computing framework?	193
9.2	Off-chip MAHA using NAND Flash technology	202
9.2.1	Modifications to Flash array organization	202
9.2.2	Modifications to CPU-Flash interface	205
9.2.3	Overhead estimation	207
9.3	Integration with CPU based software flow	211
9.4	Simulation results	213
9.4.1	Processor energy and performance estimates	214
9.4.2	MAHA energy and performance estimates	216
10	Conclusion and future directions	220
	Appendices	223
	Appendix A	223
A.1	Description of the hardware architecture	223
A.1.1	Custom datapath in the MAHA framework	223
A.2	Description of individual steps in software flow	225
A.2.1	Decomposition	225
A.2.2	Fusion	228
	Bibliography	233

List of Figures

2.1	Comparison of a reconfigurable framework with processor and ASIC	10
2.2	Possible couplings between the CPU core and a reconfigurable functional unit	11
2.3	Conventional island-style FPGA	14
2.4	DPGA, RaPiD and Chimaera reconfigurable frameworks	16
2.5	Matrix, Piperench and Morphosys reconfigurable frameworks . . .	19
2.6	Technological scalability of interconnects	22
2.7	Performance scaling trends for benchmarks mapped to Altera Stratix FPGA family	27
2.8	Comparison of available bandwidths at different levels of the memory hierarchy	30
3.1	Functional block diagram of memory based computation	41
3.2	Packing of logic nodes into multi-input multi-output partitions mapped to embedded memory blocks (87).	43
4.1	μ -architecture of a single memory logic block (MLB) in the MBC framework	52
4.2	Multi-MLB communication	58
4.3	Hierarchical organization of the memory array	62
4.4	Instrumenting a hierarchical memory array for reconfigurable computing	66

LIST OF FIGURES

4.5	Instrumentation of hierarchical memory interconnect for multi-MLB communication	68
4.6	Modes of operation for the MAHA framework	71
5.1	Software flow for the proposed memory based computing model	82
5.2	Example of CDFG scheduling	83
5.3	Grouping of load/store operations	84
5.4	Example of packing step	88
5.5	Simultaneous scheduling-placement-routing heuristic	91
6.1	Design parameters for MBC framework	102
6.2	Comparison of MLB tile area and switch area with baseline FPGA	102
6.3	Breakdown of MLB energy and delay in terms of individual components	109
6.4	Technological scalability of performance for MBC and FPGA frameworks	111
6.5	Methodology for exploring the hardware design space of the MAHA framework	114
6.6	Design overhead for custom datapath	115
6.7	Design overhead for operand selection logic inside a MLB	116
6.8	Routability for designs mapped to MAHA framework with different hierarchy and <i>Buswidth</i> configurations	117
6.9	Variation in latency and MLB count with MLB design parameters	118
6.10	A novel SRAM cell preferentially designed for improved read performance	123
6.11	Improvement in energy and stability with new SRAM cell	125
6.12	Asymmetric behavior of proposed SRAM cell	126
6.13	Preferentially design STTRAM cell	128
6.14	Comparison of energy-efficiency between STTRAM-MBC and SRAM-FPGA	130
6.15	NOR based T-CAM cell implementation and its encoding.	132
6.16	Proposed T-CAM based realization of individual partitions.	134

LIST OF FIGURES

6.17 Exponential reduction in memory requirement with CAM based MBC implementation	135
7.1 Abundance of narrow-width operations in general-purpose appli- cations	138
7.2 MBC using on-chip cache for reliability improvement	140
7.3 Physical and virtual address formation for MBC	141
7.4 Implementation of memory based carry-select addition	144
7.5 Modified issue logic with <i>Req</i> signal masking.	145
7.6 Improvement in defect-tolerance with MBC	148
7.7 Increase in cache conflict-miss due to MBC operation	149
7.8 Improvement in device failure probability with MBC	151
7.9 Improvement in MBC overhead with <i>Result-CAM</i>	152
8.1 Variation in read-stability across blocks of a large memory array .	157
8.2 Impact of variability on output image quality	159
8.3 Methodology for robust and low-power operation in MBC	160
8.4 Preferential mapping of DCT to memory blocks	163
8.5 Preferential mapping approach under timing constraint	164
8.6 Comparison of variability-aware placement under delay constraint and delay-optimal placement	165
8.7 Proposed SRAM cell sizing methodology to improve read-stability	166
8.8 Impact on writability due to proposed cell sizing	168
8.9 Improvement in output quality with preferential mapping and SRAM sizing	170
8.10 ECC allocation based on NMOS and PMOS variability in a mem- ory cell	173
8.11 Major steps in variable ECC allocation	174
8.12 μ -architecture for variable ECC allocation	177
8.13 Modifications to the states of the L2 ECC controller	178
8.14 Number of failures with $BFP = 10^{-4}$	180
8.15 Percentage increase in CPI and Energy with $BFP = 10^{-4}$	181

LIST OF FIGURES

8.16	Percentage increase in CPI and energy for soft-error tolerance . . .	182
8.17	Overhead for a combination of variable ECC and bit-interleaving scheme	183
8.18	Energy savings in variable ECC induced low-power mode	184
9.1	Variation in total energy for a system with an off-chip MAHA framework as the program behavior varies	196
9.2	Improvement in EDP for off-chip MAHA based execution with different program behavior	198
9.3	Technology trends for Flash and other non-volatile memories . . .	199
9.4	System-level diagram for off-chip MAHA framework based on non- volatile Flash memory	203
9.5	Narrow-read scheme for Flash memory array	205
9.6	Control engine external to Flash array for MAHA operation . . .	206
9.7	Area overhead for a Flash block due to instrumentation	209
9.8	Integration of MAHA to the conventional software flow	212
9.9	Breakdown of the total energy consumed in the MAHA framework	218
9.10	Breakdown of the total execution time in the MAHA framework .	218
A.1	Custom datapath for the MAHA framework	224
A.2	Example of vertex decomposition	226
A.3	Cycle formation with conventional hypergraph partitioning ap- proaches	228
A.4	Fusion of multiple bit-sliceable operations	231

List of Tables

2.1	Taxonomy of modern reconfigurable frameworks.	14
2.2	Taper in operands per unit energy with access to higher levels in the memory hierarchy.	27
5.1	Instruction set for MBC framework	79
6.1	MBC design space exploration - I	104
6.2	MBC design space exploration - II	104
6.3	MBC design space exploration - III	105
6.4	MBC design space exploration - IV	105
6.5	Comparison of MBC and FPGA frameworks for PTM 65nm . . .	110
6.6	Simulation setup for MAHA hardware exploration	113
6.7	Improvement in stored logic ‘0’ count achieved through skewing of LUTs	121
6.8	Comparison of memory requirement, area, delay and energy/vector for 12X12 partitions	135
7.1	Comparison of area overhead between MBC (due to glue logic) and resource duplication	150
8.1	Transistor widths for nominal and skewed memory cell	167
8.2	Cell failure probabilities for $V_{ddnom} = 0.9V$	168
9.1	Flash organization and performance (106)	208

LIST OF TABLES

9.2	Common MLB design configurations for our benchmark suite . . .	210
9.3	Processor configuration and energy estimates	214
9.4	Performance and energy results for processor based execution of benchmark suite	215
9.5	Benchmark mapping results	216
9.6	Improvement in energy-efficiency in off-chip MAHA framework . .	219

Acknowledgements

The three most important people in getting me this far are my father, Amarnath Paul, my mother, Sulata Pal and my advisor Prof. Swarup Bhunia. Words will not be enough to describe the sacrifice my parents made for me and my gratitude towards them. This dissertation would never have been possible without their support and blessings. I consider myself extremely fortunate in this regard.

I am deeply indebted to my advisor Prof. Swarup Bhunia who made this dissertation possible. More than guidance, he has been a constant source of motivation and enthusiasm over all these years. The best part is that I have always enjoyed my research under him and learnt an important lesson, “Ask not what an idea can do for you, ask what you can do with an idea”.

My special thanks to Prof. Christos Papachristou, Prof. Francis Merat, Prof. Prabhat Mishra and Dr. Jayabrata Ghosh Dastidar for serving on my dissertation committee. I would like to thank Prof. Saibal Mukhopadhyay, from Georgia Institute of Technology and Prof. Hamid Mahmoodi, from San Francisco State University for providing key insights into circuit design for the research projects we have collaborated on. I also thank Dr. Francis Wolff for his many constructive inputs. To all members of our *Nanoscape* lab, the past few years have been spectacular because of you. This dissertation would not have been possible without the discussions, debates and active collaboration that we shared over all these years.

Finally I would like to thank my aunt, Sabita Bandyopadhyay my uncle, Ramshyamal Bandyopadhyay, who made staying alone away from home possible.

Computing with Memory for Energy-Efficient Robust Systems

Abstract

by

Somnath Paul

Reconfigurable computing platforms that offer the flexibility to configure hardware resources according to application requirements, provide great opportunity to accelerate wide variety of applications. Over the past decade, Field Programmable Gate Arrays (FPGAs) have grown to be the most popular hardware reconfigurable computing platform. Modern FPGAs integrate an array of spatially distributed logic/memory blocks and programmable routing resources. Such a framework can provide several orders of magnitude more throughput compared to conventional microprocessor based designs. The power and performance of conventional FPGA platform is largely dominated by programmable interconnects, which have poor technological scalability. Moreover, the performance improvement for applications mapped to the FPGA platform is largely limited by the off-chip bandwidth. A reconfigurable framework which minimizes the contribution from the programmable interconnects and mitigates the bandwidth bottleneck by moving the computing engine close to the data is expected to significantly improve the energy efficiency of reconfigurable systems.

In this work, we propose a novel hardware reconfigurable framework, referred to as memory based computing (MBC) framework. The main

computing fabric for such a framework is a 2-D memory array which is used to store the functional behavior for the mapped application. Each computing element in the framework is temporal in nature and an array of these elements is used to map an application in a spatio-temporal fashion. Temporal execution inside each compute element reduces the requirement for programmable interconnects, thus improving the energy-efficiency over a fully spatial reconfigurable framework. In addition to storing the functional behavior, the memory arrays also store data, thus mitigating the off-chip bandwidth bottleneck. The framework is particularly appealing for system design with many emerging non-silicon nano-devices, which are amenable to dense, regular nonvolatile memory design. With the primary computing fabric being memory, the proposed MBC framework can be made robust to high device failure rates at nanoscale technologies. We have developed architecture and circuit level optimization techniques for the proposed framework along with efficient algorithms for automatic mapping of applications to this framework. Finally, we have investigated application of this framework as a reconfigurable computing resource in a processor for reliability improvement and hardware acceleration.

1

Introduction

Designers of digital systems face a fundamental trade-off between flexibility and efficiency when selecting computing elements. The available alternatives span a wide spectrum with general-purpose microprocessors and application-specific integrated circuits (ASICs) at opposite ends. Microprocessors are used in personal computers, in workstations, in servers, as building blocks for contemporary supercomputers, and increasingly in embedded systems. They are flexible due to their versatile instruction sets that allow the implementation of any computation task. ASICs on the other hand are dedicated hardware circuits tuned to a specific task or few similar tasks. ASICs are mainly used in high-volume embedded system markets such as telecommunications, consumer electronics, or the automotive industry. For a given task, dedicated circuits execute faster, require less silicon area, and are more power efficient than general-purpose architectures. The drawback of such highly specialized architectures is their lack of flexibility i.e. if the application changes, a redesign of the ASIC is required. To address the lack of flexibility of ASICs, the new class of reconfigurable computing devices has emerged, which promises to combine the flexibility of processors with the efficiency of ASICs (1, 2, 3). The hardware of reconfigurable devices is not static but adapted to each individual application. Through hardware customization, reconfigurable devices potentially achieve a higher efficiency than microprocessors,

while the dynamics of the customization process allow a higher level of flexibility than ASICs.

The enabling technology for building reconfigurable systems was the field-programmable gate array (FPGA) (4). FPGAs were introduced to the market at the high-end of programmable logic devices (PLDs) in the mid 1980s. FPGAs consist of an array of logic blocks, programmable routing channels to interconnect the logic blocks, and surrounding I/O blocks. Static Random Access Memory (SRAM) based FPGAs use static RAM (SRAM) cells to control the functionality of the logic, I/O blocks, and programmable interconnect. They can be reprogrammed in-circuit arbitrarily often by downloading a bitstream of configuration data to the device. Typically, FPGAs are fine-grained architectures that operate on bit-wide data types and use look-up tables (LUTs) as computing elements. While early FPGA generations were quite limited in their capacity, today's devices feature millions of gates of programmable logic, dense enough to host complete computing systems. Traditionally the major challenge in FPGA design has been improving the area efficiency, i.e. fitting more resources per unit silicon area. A notable step made in this direction was to cluster distributed LUTs inside the FPGAs into LUT clusters. This led to considerable improvement in the area-delay product by reducing the requirement for programmable interconnects inside the FPGA. Another recent development has been the inclusion of what we call hard "specific-purpose" circuits, such as hard multipliers, adders, and memory blocks (5, 6). These custom blocks allow specific operations in a design to be more efficient in terms of area, power and performance. As device technology is continuously scaled, challenges in FPGA design are not limited to improvement in the area efficiency. Improving the energy efficiency and sustaining the reliability of operation with nano-devices which are prone to defects and failures have emerged as major challenges in designing reconfigurable systems such as FPGAs (7, 8).

1.1 Research objectives

The objective of this dissertation is to propose a novel hardware reconfigurable framework, which addresses the challenges of designing a generic reconfigurable framework at nanoscale technologies. The challenges are manifold and we try to describe them below.

Conventional FPGA frameworks are fully spatial computing frameworks and the area, power and performance for such frameworks are completely dominated by the programmable interconnects. As opposed to logic, interconnect power and performance is not scaling with each technology generation. It is therefore worthwhile to investigate an alternative reconfigurable architecture which minimizes the contribution from the programmable interconnects and therefore scales well with technology. Keeping in consideration the fact that at scaled technologies, nano-devices are more prone to manufacturing defects and runtime failures, such a framework must also be robust against hard-defects and variability induced failures. Finally, the framework must also be amenable to implementation using emerging non-CMOS nano-devices.

In addition to conventional FPGAs, currently, a strong trend towards *hybrid reconfigurable processors* can be observed (9). These hybrid architectures combine standard processor cores with a reconfigurable functional unit (popularly known as *hardware accelerator*) to leverage on both the flexibility of software based execution and the efficiency of hardware reconfigurable platforms. The key advantage is that the reconfigurable processing unit of the hybrid architecture can be used as a coprocessor and can accelerate the portions of the application which are expensive in software based execution. This is especially attractive for computationally demanding applications found in areas such as multimedia, cryptography, and communications. Hence, there is a growing demand for hardware accelerator which is low-overhead and energy-efficient and still retains the flexibility of mapping applications from diverse application domains.

With the explosion of data transmitted over the internet, there is an increasing requirement for compute systems which can service data-intensive applications. Performance of data-intensive applications when mapped to hardware frameworks

such as FPGAs or to *hybrid reconfigurable processors* have been shown to be limited by off-chip bandwidth in accessing the data from the off-chip memory. Technology trends predict that the off-chip memory bandwidth is not scaling at the same rate as logic density on the chip. It is therefore easy to deduce that future data-intensive applications will be severely limited by off-chip memory bandwidth, thus worsening the Von-Neumann bottleneck (10).

In this work, we take a clue from the existing and emerging technology trends and try to arrive at a design solution which addresses the above challenges.

1.2 Contributions

A close look at the challenges outlined in the previous section suggests that future compute systems need to be closely integrated with the memory array, preferably with a non-volatile memory where data can be permanently stored. Note that with such a requirement, a purely software reconfigurable system may not be scalable and would compromise the integration density of the memory array. A more plausible solution is to build an alternative or complementary hardware reconfigurable framework which would be spatially distributed across the data storage. Realizing a generic reconfigurable framework would however mean incorporating look-up tables in the proposed framework which can map any functions with arbitrary complexity. Moreover, it will require programmable interconnect similar to the FPGA framework for communicating between the look-up tables (LUTs). In this context, we realized that it is possible to store the look-up tables in the same memory which is being used to store data. A simple control logic can be used to access these look-up tables in a topological manner and thus evaluate a large function. We therefore propose a hardware reconfigurable framework where each computing element includes a dense 2-D memory array for both computing and storage and operates in a temporal manner, thus performing the computations locally. For a sufficiently large application, which needs to be spatially mapped across multiple compute blocks, a spatio-temporal computing model minimizes

the requirement for programmable interconnects and thus enjoys a better technological scalability compared to a fully spatial framework. Since defects and failures in memory can be easily tolerated, the proposed framework also satisfies the key requirement that the computing fabric is robust to high rate of failure for nano-devices. Lastly, we note that computing with memory is particularly appealing for emerging non-CMOS devices which are often amenable to the implementation of regular, dense, 2-D non-volatile memory array.

In this work, we make the following key contributions:

- i. We propose a novel generic reconfigurable framework which can map arbitrary application and execute it in a spatio-temporal manner. Computing at each reconfigurable compute block is performed by using memory array. LUTs which represent the function behavior are stored in a dense 2-D memory array and evaluated in a topological manner over multiple clock cycles. Each compute element of the framework thus operates in a temporal fashion, while multiple such blocks communicate with each other in a spatial manner. We describe in details the hardware architecture for the proposed memory based computing framework.
- ii. We present a complete software flow which is capable of automated mapping followed by functional validation of arbitrary applications to the proposed framework. Details of the software architecture are provided as part of this work.
- iii. We propose a novel hardware acceleration engine called *MAHA* which instruments an already existing memory organization in a processor with minimal additional logic so that the memory can be on-demand transformed into a hardware reconfigurable computing resource. We describe in detail the hardware modifications required for such a transformation and present system level results for energy-efficiency if an off-chip non-volatile memory is used to realize such a framework.

- iv. We describe how the proposed memory based computing model can be extended to improve reliability of operation for a general-purpose processor system with embedded memory.
- v. We also investigate a preferential mapping and a variability-aware error correction code assignment for on-chip memory to improve the reliability of operation for the memory based computing framework.

1.3 Thesis organization

The organization of the dissertation is as follows:

- Chapter 2 reviews the background for this work. First, it presents the taxonomy of different reconfigurable frameworks and then surveys the state of the art reconfigurable platforms. It then describes the challenges in reconfigurable system design at nanoscale technologies and summarizes the circuit, architecture and software solutions that have been proposed to date to counteract these challenges. This is followed by the challenges in the design of general computing systems which includes power constrained design, poor scalability of interconnect performance and sustaining the reliability of operation at nanoscale technologies. Finally, it puts forward the motivation for the memory based reconfigurable framework proposed in this work.
- Chapter 3 provides an overview of the hardware and software architectures for the proposed memory based computing (MBC) framework. It then distinguishes *Computing in Memory* from *Computing with Memory* concept proposed in this work. Difference between MBC and conventional FPGAs, embedded memory based FPGAs, time-multiplexed reconfigurable platforms and other spatio-temporal hardware reconfigurable frameworks are then presented. Finally we describe how MBC can be used for different motifs, contexts and domains of application.

- Chapter 4 describes in detail the hardware architecture for the proposed MBC framework. The μ -architecture for individual compute blocks is first elaborated followed by the communication and synchronization across multiple compute blocks. Then we describe a hardware reconfigurable framework which can be created on demand from an existing memory organization. The architecture is referred to as MAlleable Hardware Accelerator or MAHA. Conventional memory organization and how it may be instrumented to create a reconfigurable computing platform on demand is then described in details.
- Chapter 5 follows up the hardware description in the previous chapter with the software architecture we have developed for automatic mapping of applications to the MBC framework. Here we provide an overview of the main steps in the software flow for mapping applications to both MBC based stand-alone reconfigurable computing as well as to the MAHA framework.
- Chapter 6 analyzes the results for design space exploration of the MBC based stand-alone reconfigurable framework and the MAHA framework. It quantifies the effect of different MBC design parameters on the power and performance for representative benchmarks mapped to the proposed reconfigurable framework. Exploiting the fact that most of the memory operations in the MBC framework is read dominated, it reports the improvement in performance and power for the MBC framework realized using novel memory designs which are preferentially designed for read. Finally it describes how memory requirement for mapping applications to the MBC framework can be exponentially reduced by representing the input application in the form of programmable logic arrays (PLA) rather than look-up tables (LUT). A novel ternary-CAM based MBC framework is described in this context.
- Chapter 7 describes a novel application of the memory based computing where it is used to improve reliability of operation for general-purpose microprocessor. The idea is to migrate operations from defective or non-

operational (due to parameter variations) functional units inside the processor to the on-chip cache and perform on-demand computation inside the cache.

- Chapter 8 describes two approaches to improve the reliability of operation for the MBC framework. The first approach combines preferential mapping of input application to more reliable memory blocks with preferentially designed memory to tolerate read failures. The second approach is to use a variability-aware error correction code assignment to different sections of a large memory array to tolerate runtime failures.
- Chapter 9 describes the design considerations required to realize the proposed MAHA framework realized with commercially available CMOS-compatible NAND Flash memory. We note that such a framework effectively mitigates the Von-Neumann bottleneck and leads to significant improvement in energy-efficiency for the entire system.
- Chapter 10 concludes this research and proposes avenues for future investigations.

2

Background & motivation

Since their inception in 1984, the basic structure of the FPGAs has continued to consist of Configurable Logic Blocks (CLBs) and a programmable interconnect (PI) matrix (11). Although efficient application mapping algorithms have been proposed for improving the resource utilization and reduce the PI overhead in a FPGA architecture, the latter accounts for about 80% of power, 60% of delay and 75% of area in deep submicron processes (12, 13). In order to address the overhead due to an elaborate PI architecture, a significant number of investigations have been made on alternative implementations of hardware reconfigurable frameworks. In this chapter we present a brief survey of state of the art reconfigurable computing frameworks. We describe the challenges faced by these frameworks and computing systems in general. We then explore the opportunities to improve and propose directions based on the technology trends for current and future nano-devices.

2.1 Background

In this section we begin with a high-level review of current electronic system designs and describe how reconfigurable computing bridges the flexibility and

Figure 2.1: a) Reconfigurable computing bridges the gap between microprocessors and ASICs. b) Possible couplings between the CPU core and a reconfigurable functional unit.

performance gap between application specific integrated circuits (ASICs) and general purpose microprocessors. Based on the granularity of the processing elements, the reconfigurable systems are further divided into coarse and fine grained architectures. We introduce both these architectures and cite a variety of reconfigurable systems. Next, we identify the major challenges confronted by current reconfigurable architectures and discuss possible avenues to mitigate those challenges. Lastly, we extend our discussion to challenges faced by modern computer systems and how novel non-Von-Neumann architectures can be effective in bypassing them.

2.1.1 Reconfigurable computing - state of the art

Researchers and developers are constantly enlarging the application space and improving the performance of computing machinery. Some examples are weather prediction and nuclear explosion simulation in scientific computing, DNA sequence matching in bioinformatics, high-speed media applications over mobile networks, high speed switching and routing and so forth (14). Applications from

Figure 2.2: Possible couplings between the CPU core and a reconfigurable functional unit.

these diverse domains have been traditionally computed on a software reconfigurable platform such as a general-purpose processor (GPP). However, the energy-efficiency measured in terms of performance per watt achieved from a GPP has not found to scale with technology scaling (15). ASICs have always been found to offer the best performance per watt among all the computing platforms (16). The downside is however, such a platform is committed to one or more applications during fabrication and therefore possesses limited flexibility in executing arbitrary applications. Reconfigurable computing is a novel computing paradigm which combines the flexibility of software with the high performance of application specific hardware, bridges the gap between general-purpose processors and application-specific systems, and enables higher productivity and shorter time to market (11). Fig. 2.1) illustrates the same idea. As we note from Fig. 2.1, by definition, ASICs have a fixed functionality and only operate on input data. Processors are software reconfigurable and the application is defined in terms of instructions. Systems such as FPGAs are however hardware reconfigurable and the input application is defined in terms of configuration bits which must be loaded into the reconfigurable framework before it executes an application.

2.1.1.1 Taxonomy for reconfigurable systems

A reconfigurable computing system consists of an array of *reconfigurable* processing elements (PEs), the behavior for each of which as well as their connectivity

can be customized based on the application which is to be mapped to the reconfigurable system. The programmable interconnect which connect these processing elements together is responsible for steering the inputs and outputs as well as intermediate results to correct processing elements. A reconfigurable architecture with adequate interconnect and compute resources greatly improves the flexibility, resource utilization and therefore provides superior performance. In addition to the reconfigurable frameworks which are commercially available, a large number of reconfigurable architectures have been proposed till date. These can be distinguished from one another by specifying some or all of the following characteristics for each of these reconfigurable frameworks.

1. Granularity: The granularity of a reconfigurable architecture is defined as the width of the smallest processing element. Based on the granularity, reconfigurable systems can be classified into *fine-grained* and *coarse-grained* reconfigurable systems. In fine-grained systems, each PE primarily consists of logic gates, flip flops and lookup-tables. They operate at bit level, implementing a boolean function of a finite-state machine. On the other hand PEs in *coarse-grained* architectures contain complete functional units, like ALUs and/or multipliers that operate on multi-bit words. Coarse-grained reconfigurable frameworks have been shown to significantly reduce the PI overhead for the fine-grained reconfigurable framework and reduce the complexity for Place and Route (P&R) problems (17). However, the major drawback of coarse-grained reconfigurable architectures is that the optimal granularity of the reconfigurable resources varies from one design to another. Thus when the application data does not match with that of the logic in a coarse grained framework, performance decreases (18). Moreover, the specialized coarse-grain functional units and very structured communication resources makes application mapping less flexible for these architectures (18). Many modern reconfigurable systems including FPGAs therefore combine the benefits of both *fine-grained* and *coarse-grained* system and are known as *mixed granular* reconfigurable systems.

2. Depth of programmability: This pertains to the number of configuration programs or *contexts* that may be stored inside each PE. For *multi-context* PEs,

contexts for several programs reside concurrently in the system. This enables the execution of different tasks by simply changing the operating context without having to reload the configuration program.

3. Reconfigurability: In a reconfigurable system, reconfiguration is the process of reloading configuration programs (contexts). The process can be either *static* (execution is interrupted) or *dynamic* (in parallel to the execution). Dynamic reconfiguration is more relevant for multi-context PEs. This feature significantly reduces the overhead for reconfiguration.

4. Interface: Reconfigurable computing systems can work as a stand alone computing system as well as in conjunction with a system's host processor. The reconfigurable system has a *remote* interface if it is not on the same die as the host processor and communicates over some I/O bus, e.g. PCI bus. The interface is *local* if the reconfigurable system and the host processor reside on the same die. The communication is similar to a co-processor model and is over a fast on-chip bus. A reconfigurable system is called *tightly coupled* if it is placed inside the processor. The instruction unit issues instruction to the reconfigurable unit as if it is a standard functional unit of the processor. Possible interfaces between the reconfigurable functional unit (RFU) and the host processor is shown in Fig. 2.2.

5. Application Domain: Different reconfigurable frameworks cater to different application domains. The suitability towards a particular application domain is mainly determined by the granularity of the framework and the computational model which may be single instruction multiple data (SIMD) or multiple instruction multiple data (MIMD) models.

Table 2.1 shows the classification of some of the popular reconfigurable frameworks based on the above taxonomy.

2.1.1.2 A brief survey of reconfigurable frameworks

Here we briefly describe some of the reconfigurable architectures proposed till date.

1. Field Programmable Gate Arrays (FPGAs): The most popular FPGA arrangement today is the *island-style* architecture (19). As seen from Fig. 2.3, it

2.1 Background

Table 2.1: Taxonomy of modern reconfigurable frameworks.

	Granularity	Programmability	Reconfiguration	Interface	Appl. Domain
FPGA	Mixed	Single	Static	Local/remote	General Purpose
DPGA	Fine	Multiple	Dynamic	Remote	Bit-level computations
RaPiD	Coarse	Single	Static	ND*	Systolic nature, easily pipelined
Chimaera	Fine	Single	Static	Tightly Coupled	Bit-level computations
Matrix	Coarse	Multiple	Dynamic	ND*	ND*
Piperench	Mixed	Multiple	Dynamic	Remote	DSP
Morphosys	Coarse	Multiple	Dynamic	Tightly coupled	Image Proc.

ND* - Not defined

Figure 2.3: Conventional island-style FPGA.

consists of computational resources which are divided into small islands of logic blocks which are surrounded by a sea of interconnect wires and programmable communication resources. Each logic block can generally implement any function of N-inputs 1-output through the use of Look-up tables (LUTs). LUTs are simply small memories which allow the inputs to the logic block to address a particular location in a read-only memory. By populating the LUTs appropriately, the user may map any N-input 1-output function to these LUTs. FPGAs also offer the opportunity to implement sequential logic by providing an optional flip-flop at the output of the LUTs. The LUT/flip-flop pair is sometimes referred to as the basic logic element (BLE).

The communication resources required for an island style FPGA can be divided into three components: i) *channels* are groups of individual wires that are locally organized into bundles by their physical location. Fig. 2.3(a) shows a channel width of 4. ii) *Connection boxes* are responsible for managing the movement of data into and out of the channel. They control the wires which become logic block input or primary output or become logic block output or primary input. iii) *Switch boxes* are responsible for connecting wires in different channels together. When synthesizing arbitrary netlists into FPGA, the mapping tool maps the input netlist to LUTs and flip flops and binds them to the logic block. These blocks are then placed and routed using the programmable interconnects.

In order to bridge the large area, energy and performance gap that exists between pure LUT based mapping and custom logic implementation for some functions, FPGA vendors have integrated more components on the FPGAs, such as general purpose processors, custom adders and multipliers for digital signal processing and block RAM modules. Moreover, in addition to single length wires as shown in Fig. 2.3(a), they have interconnect wires that span multiple logic blocks and improve the speed for long distance communication. FPGAs may also include *dedicated carry chains*. Lastly, in order to improve energy and performance, modern FPGAs cluster together multiple BLEs within a single logic block/cluster, which allows BLEs within a cluster to communicate with each other without using external routing wires.

Figure 2.4: a) Context memory at each logic block and routing switch in a DPGA architecture (20). b) RaPiD - a linear reconfigurable array ref:rapid. c) Chimaera reconfigurable architecture (21).

2. Dynamically Programmable Gate Arrays (DPGAs): DPGA is a programmable array which allows the strategic reuse of limited resources. In so doing, DPGAs promise greater capacity, and in some cases higher performance, than conventional programmable device architectures where all array resources are dedicated to a single function for an entire operational epoch. This is achieved by building a multicontext FPGA, which holds several configuration for each gate and switch as shown in Fig. 2.4(a). A broadcast context identifier tells each primitive which configuration to select at any given point in time. Through techniques such as temporal pipelining, utility functions, multiple function accommodation and state dependent logic, DPGAs have been proven to effectively improve resource utilization for a variety of applications (20).

3. Reconfigurable Pipelined Datapath (RaPiD): is a linear array of PEs, which is configured to form a linear computational pipeline, as shown in Fig. 2.4(b). Each PE comprises an integer multiplier, three integer ALUs, some

general-purpose data registers, and three small RAMs. A typical RaPiD contains up to 32 such PEs. These PEs are interconnected using a set of segmented buses that run the length of the linear array. The buses are segmented into different lengths and placed in different tracks. The framework distributes computations over the pipeline stages and is amenable to mapping of regular streaming applications (22).

4. Chimaera: It integrates a reconfigurable logic into the host processor (refer to Fig. 2.4(c)). Sequences of instructions are mapped to one or more lines of the reconfigurable array. In some cases, sequences of instructions are collapsed and then mapped. Reconfigurable operations (RFUOPs) are extracted using the Chimaera compiler. Chimaera is a fine grained architecture which supports multi-operand instruction. It does not however have any state elements and uses the *shadow register file* of the host processor for temporary storage. Because the reconfigurable operations are extracted at compile time, the Chimaera architecture cannot span multiple basic blocks. Also the mapped operations are still dependant in the reconfigurable array, so the operations are not really collapsed.

4. MATRIX: As shown in Fig. 2.5(a) is a coarse-grained reconfigurable array which supports configurable instruction distribution. MATRIX is an improvement over the DPGA concept. Rather than separate the resources for instruction storage and distribution from the resources for data storage and computation and dedicate silicon resources to them at fabrication time, the MATRIX architecture unifies these resources. Once unified, traditional instruction and control resources are decomposed along with computing resources and can be deployed in an application-specific manner. The adaptability is made possible by a multi-level configuration scheme, a unified configurable network supporting both datapaths and instruction distribution, and a coarse-grained building block which can serve as an instruction store, a memory element, or a computational element. At 0.5μ process point, MATRIX was demonstrated to achieve 10Gops, running at 100Mhz.

4. Piperench: It functions as a remotely-coupled reconfigurable co-processor for pipelined applications (refer to Fig. 2.5(b)). PipeRench contains a set of physical

pipeline stages called stripes. Each stripe has an interconnection network and a set of PEs. Each PE contains an arithmetic logic unit and a pass register file. Each ALU contains lookup tables (LUTs) and extra circuitry for carry chains, zero detection, and so on. Through the interconnection network, PEs can access operands from registered outputs of the previous stripe, as well as registered or unregistered outputs of the other PEs in the same stripe. Moreover, the PEs access global I/O buses. Piperench overcomes the cost issues of implementing a circuit using FPGA cells by implementing a large circuit on a small number of FPGA cells through a technique known as *hardware virtualization*. Piperench is suitable for stream-based media applications and regular fine-grained computations.

5. Morphosys: Morphosys as illustrated in Fig. 2.5(c) is a coarse-grained reconfigurable array which is tightly coupled with a general purpose processor for accelerating word-level compute-intensive applications. It is targeted at high-throughput and data-parallel applications. Morphosys incorporates a 2-D array of reconfigurable cells organized in a SIMD fashion and also a high bandwidth memory interface that consists of a streaming buffer to handle data transfers between external memory and the Morphosys array. The suitability of the Morphosys framework has been demonstrated for target application domains such as video compression, data encryption and target recognition.

In addition to the above frameworks, a large number of other stand-alone reconfigurable frameworks (25, 26, 27, 28, 29, 30, 31, 32, 33) can be found in literature. While some of these are for general purpose applications (30, 31, 32), others are targeted towards a specific application domain (28, 29, 33).

6. ASIPs and RISPs: Reconfigurable frameworks often work in conjunction with a host processor to map data-parallel and compute-intensive kernels which become performance and energy bottleneck in simple software based execution (17). Reconfigurable solutions in processor systems can be classified into two categories: i) static and ii) dynamic. The static solution otherwise known as application specific instruction set processors (ASIPs) are statically configured during their design and cannot be configured at runtime. The idea is to

2.1 Background

Figure 2.5: a) Matrix reconfigurable architecture with configurable instruction distribution and deployable resources (23). b) Piperench - an attached reconfigurable co-processor for pipelined applications (24). c) Morphosys - An integrated reconfigurable array for data-parallel and compute-intensive applications (9).

first identify the application domain for which a processor will primarily be used. Next common characteristics are extracted from applications pertaining to that application domain. Following this specialized hardware units are incorporated into the processor at design time and the processor instruction set is altered to support the operation for this specialized unit. During runtime, this hardware unit is used to accelerate domain specific instruction calls which would otherwise be costly to execute with the primitive instruction set. Tensilica’s Xtensa customizable processors (34) and Stretch’s S7000 software configurable processors (35) are examples of ASIPs.

In contrast to ASIPs which are statically designed, it is possible to incorporate a RFU inside the processor so that the system may dynamically adapt to computing demands from different application domains. Such architectures are

referred to as reconfigurable instruction set processors (RISPs). A RISP has more adaptability compared to an ASIP. There are however more hardware and software aspects that one needs to consider during the design of such a reconfigurable processor. These are: i) coupling of the processor and the reconfigurable logic, ii) configuration, iii) instruction coding and scheduling, iv) granularity, v) hardware cache and vi) reconfigurability. A detailed survey of reconfigurable processor architectures and their design considerations can be obtained in (17, 36).

2.1.2 Challenges in reconfigurable system design

As technology scales reconfigurable systems are confronted by a number of challenges at all levels of design abstraction, namely, circuit, architecture and software. The common challenges which applies to all three are i) minimizing the power for reconfigurable frameworks, ii) improving the performance of the reconfigurable framework and iii) tolerate defects. Here we try to summarize the solutions which have been proposed to address these challenges.

2.1.2.1 Circuit

1. Power: In nanoscale technologies, power has emerged as the primary constraint to the design of compute systems (37). Since reconfigurable systems are being used more and more in mainstream computing, there arises a strong demand for reducing the power requirement for FPGAs and similar other reconfigurable systems. At the same technology node, a FPGA has been found to consume $12\times$ the dynamic power compared to an ASIC implementing the same application (38). The major contribution to power comes from the programmable interconnect which has been found to contribute 60% (39) to the total dynamic power and 55% (40) to the total leakage power in the FPGA platform. Circuit level techniques for power reduction traditionally incorporate dual- V_{dd} and/or dual- V_t fabrics in the same FPGA device. Designs are mapped to a combination of computing elements with high and low V_{dd} and V_t depending on the performance constraint for dynamic and leakage power reduction (7). Leveraging on this dual

V_{dd} and V_t assignment, low power designs of the logic block (41) and routing switches (42) have been previously proposed. Region based power gating (43) as well as fine-grained power gating (44) have also been proposed for power reduction in FPGA frameworks.

2. Performance: Like power, interconnect is also the performance limiter in case of conventional FPGA frameworks. For modern FPGAs, interconnect delay has been shown to contribute 40-80% of the critical path delay (12). In our simulations with a clustered FPGA architecture built using PTM 45nm models (45), we have also seen that net delay contributes on an average 40.3% to the critical delay for MCNC benchmark circuits when mapped to the FPGA framework. The situation is particularly grim since the delay for through the Cu-interconnect is not decreasing, rather increasing with each technology generation. This is in complete contrast with the gate/logic delay which continues to scale with each technology node (refer to Fig. 2.6). The primary reason as explained in (46, 47) is that wire resistance is increasing with each technology node. It is therefore likely that performance for programmable interconnects will continue to remain a serious bottleneck in future FPGA designs (46). Little can however be done at the circuit level to mitigate the effect of the programmable interconnects. Circuit optimizations in routing switches through buffer and transistor sizing have only demonstrated minimal ($\sim 7\%$) improvement in delay (48).

3. Reliability: In nanoscale technology, reconfigurable frameworks such as FPGAs have been shown to have increased vulnerability towards logic and interconnect failures. Among the many causes for logic failure, the topic which is highly researched is soft-error in SRAM based FPGA. In scaled technologies, the impact of hit of energized particles is likely to be more severe since it may lead to multi-bit upsets, thereby causing detection and correction a serious challenge (49). At circuit level, researchers have proposed different implementations of SRAM (8) which are hardened to soft-error induced failures. Depending on the nature of the soft-error, the effect in the FPGA may be benign or devastating. For example, soft-error induced delay failures can only lead to transient failures. Configuration bits in routing switches however may be inadvertently switched due to soft-error

Figure 2.6: Transistor and wire delay scaling trends plotted on the basis of the ITRS 2007 report.

which may even destroy the chip. An effective solution towards soft-error immunity is therefore highly desired. Parametric variations in FPGA interconnects can also lead to delay failures (50) for designs mapped to the FPGA framework. However, no effective circuit techniques have been proposed to mitigate the effect of variability in programmable interconnects across a FPGA device.

2.1.2.2 Architecture

For reconfigurable frameworks, in general architectural modifications have been found to be more effective than circuit-level optimizations.

1. Power: LUT based evaluation of common arithmetic functions such as addition and multiplications were found to be extremely expensive in terms of power and performance. Modern FPGAs therefore include custom adder and multipliers and also memory blocks which significantly improves the area and energy-efficiency for these frameworks (5, 6). Other approaches include FPGA device and architecture co-optimization (51) which explores FPGA architectures with support for V_{dd} programmability.

2. Performance: A large number of architectural modifications have been

proposed to improve performance in FPGA frameworks. From completely distributed LUT based designs (52), FPGAs have shifted to LUT clusters for improved area and performance efficiency (53). LUTs of sizes 4-6 and cluster sizes of 4-10 have been shown to be most effective in terms of area-delay product at $0.18\mu\text{m}$ (53). As the technology is scaled further, we believe LUTs with larger input size would be more area efficient. In addition to LUT clusters, modern FPGAs have become heterogeneous by incorporating custom adders, multipliers and memory elements. With only LUT based function evaluation, the performance gap between FPGAs and ASICs were estimated to be $40\times$. This gap however reduces to $21\times$ by incorporating the hard adder and multiplier elements (38). Finally, to improve the performance of interconnects, researchers have looked into pipelining of FPGA interconnects (54). Vendors like *Achronix* (55) have demonstrated that asynchronous pipelining of FPGA interconnects can significantly improve the FPGA performance. New FPGA architectures have also been proposed that include segmentation and buffering to optimize speed and density (56).

3. Reliability: Architectural support for reliability primarily include automatic detection and correction capability for soft-error induced failures in the FPGA fabric. These mainly target the configuration bits and the datapath latches in the FPGA framework. Modern FPGAs come with such support (57). Recent works have also focussed on reducing the soft-error vulnerability of SRAM arrays present on the FPGA (58).

2.1.2.3 Software

Since applications are mapped off-line onto the FPGA framework, a large number of optimizations are possible in the software for improving the power, performance and reliability of the mapped design.

1. Power: A significant amount of works has been done on power-aware application mapping for the FPGA framework. In (59) a novel synthesis approach is proposed for FPGA which improves resource selection and reduces the total number of muxes to effectively reduce FPGA power consumption. In (60), the authors

propose a design methodology that implements various computational kernels in an energy efficient manner by characterizing various architectures (e.g. linear array, systolic array) that can implement the same kernel and choosing between the architectures based on performance and resource constraints. Although such an approach can be efficient in reducing the power consumption of applications mapped to platform FPGAs (consisting of hard processors, FPGA fabric, DSP blocks etc.), it does not directly address the power consumption for the reconfigurable framework. Other works target leveraging on the V_{dd} programmability of the FPGA fabric for drastic reduction in the interconnect power (61).

2. Performance: With modern FPGAs integrating LUT clusters in their design, application mapping tools have been modified to exploit the spatial locality of the LUTs during application mapping. Moreover, these mapping tools also take advantage of fracturable LUTs for packing of applications (62). Timing-driven packing, placement and routing algorithms (63) have also been proposed to improve the routing delay for designs mapped to the FPGA framework.

3. Reliability: A number of software level optimizations have been proposed to improve the reliability of operation for FPGA frameworks. One of the approaches skews the LUT contents for improving soft-error tolerance in FPGA (8). Moreover soft-error aware placement (64) has been shown to drastically reduce the number of bits sensitive to soft-error. Furthermore, single event upset aware routing algorithms have also been proposed for reliability improvement in FPGA frameworks (65).

2.1.3 Challenges in general computing systems

Computing systems which encompass general-purpose processors, reconfigurable frameworks and ASICs are also confronted with the same set of challenges as mentioned above, namely power, performance and reliability. Since ASICs provide the maximum performance per watt at the cost of limited flexibility, we limit our focus to general purpose processors and reconfigurable frameworks which are flexible enough to map any application.

2.1.3.1 End of Dennard scaling

In his seminal paper on metal-oxide semiconductor (MOS) device scaling in 1974, Robert Dennard showed that when voltages are scaled along with all dimensions, a device's electric fields remain constant, and most device characteristics are preserved (66). Following Dennard's scaling theory, chip makers achieved a triple benefit: First, devices became smaller in both the x and y dimensions, allowing for $1/\alpha^2$ more transistors in the same area when scaled down by α , where $\alpha < 1$. Second, capacitance scaled down by α , thus, the charge Q that must be removed to change a node's state scaled down by α^2 . The current also scaled down by α , so the gate delay D decreased by α . Finally, because energy is equal to CV^2 , energy decreased by α^3 . Most importantly, following Dennard scaling maintained constant power density. Thus, scaling alone was able to bring about significant growth in computing performance at constant power profiles.

Taking advantage of Dennard's scaling, designers continued to aggressively design their circuits by incorporating faster memories, adders, and latches, and created deeper pipelines that increased clock frequency. During this period, computer designers wisely converted a 'free' resource (extra power) into increased performance, which everyone wanted. In the early 2000s, however, high performance designs reached a point at which they were hard to air-cool within cost and acoustic limits. Moreover, the laptop and mobile device markets which are battery constrained and have even lower cooling limits were growing rapidly. Thus, most designs had become power constrained. At around the same time, technology scaling began to change as well. Up until the 130-nm node, supply voltage (V_{dd}) had scaled with channel length. But at the 90-nm node, V_{dd} scaling slowed dramatically, faced with exponentially increasing leakage power costs, if the threshold voltage V_{th} is also scaled with V_{dd} . As a result, V_{dd} scaling has stopped and it is still around 1 V for the 45-nm node. With constant voltages, energy now scales with α rather than α^3 , and as we continue to put $1/\alpha^2$ more transistors on a die, we are facing potentially dramatic increases in power densities unless we decrease the average number of gate switches per second. Although decreasing frequencies would accomplish this goal, it isn't a good solution, because

it sacrifices performance. In the power-constrained, post-Dennard era, creating energy-efficient designs is critical. Continually increasing performance in this new era requires lower energy per operation, because the product of operations per second (performance) and energy per operation is power, which is constrained.

In the past, the push for ever better performance has seen designs creep up to the steep part of this energy-performance tradeoff curve. However, as Pollack's rule states (15), the performance for a single CPU system only increases as $\sqrt{r}$, where r represents the number of transistors in the design. This has forced designers to reevaluate the situation, and this is precisely what initiated the move to multicore systems. Of course, this approach sacrifices single-threaded performance, and it also assumes that the application is parallel which isn't always true. However, we cannot rely on parallelism to save us in the long term for two reasons. First, as Amdahl noted in 1965 (67), with extensive parallelization, serial code and communication bottlenecks rapidly begin to dominate execution time. Thus, the marginal energy cost of increasing performance through parallelism increases with the number of processors, and will start increasing the overall energy per operation. The second issue is that parallelism itself doesn't intrinsically lower energy per operation; lower energy is possible only if sacrificing performance also yields a lower energy point in the energy-performance space. A widespread initiative across academia and industry has therefore been undertaken to find a solution to sustaining the performance improvement with technology scaling without violating the power constraint (68).

2.1.3.2 “Computer architecture is all about interconnect”

W.J. Dally made this famous comment in Symposium on High Performance Computer Architecture, 2002. He predicted it is going to be more difficult to control the interconnect overhead in 2010. The prediction is quite true as we see in the technical report on Exascale Computing as published from DARPA in 2008 (68). With technology scaling, memory cells and arrays and logic is getting better, while interconnect is getting worse and currently it dominates the delay and power in both logic and memory systems (47). Memory latency and bandwidth

Figure 2.7: Performance scaling trends for different benchmarks mapped to Altera Stratix FPGA family.

Table 2.2: Taper in operands per unit energy with access to higher levels in the memory hierarchy.

Path	Energy(pJ)				Operands per pJ
	Total	Transport	Access	% Transport	
Register File	5	0.0	5	0.0	1.436
L1 Hit	39	0.0	39	0.0	0.26
L1 miss, L2 Hit	385	167.3	217.7	43.4%	0.033
L1 miss, L3 Hit	1991.7	1774	217.7	89.6%	0.006
Cache Miss, On-module DRAM access	1380	1204.7	175.26	87.3%	0.003
On-module DRAM miss					
Off-module DRAM access	13819	13791.4	27.63	99.8%	0.000

which always appears to be a bottleneck in computing systems are not actually limited by memory performance but by the interconnect between the client and the memory. Conventional out-of-order processors hide the interconnect latency in the form of cache misses. But as correctly pointed in (68), “it is possible to hide the delay but not the energy”. Thus there exists a requirement to appropriately the schedule the tasks of an application in both space and time and manage communication across the storage hierarchy explicitly.

In the previous subsection, we already observed that compared to logic, pro-

programmable interconnects contribute significantly to the overall delay and power in a hardware reconfigurable system. Since the interconnect delay is not scaling with technology, the critical path delay for the FPGA platform is not improving appreciably with technology scaling. Our experiments with older and latest generation of Altera FPGAs, as shown in Fig. 2.7 also suggest the poor technological scalability of performance for the conventional FPGA platform. From Fig. 2.7 we note that while the average delay improvement due to technology scaling from 130nm to 90nm is 27%, the same from 90nm to 65nm is only 13%. For a general-purpose processor system, interconnect also plays a vital role at scaled technology nodes. A detailed study in (69) shows that in a large scale multiprocessor system, a significant percentage (75%) of energy is spent in data movement. As summarized in Table 2.2, only a small fraction of energy is actually spent in reading the data from different levels of the memory hierarchy. Most of it is spent in data transport across the hierarchy. This reduces the computational efficiency for real applications to less than 5%. Thus, a computing framework which is build on an existing memory organization, preferably off-chip non-volatile memory which permanently stores the data to be processed is the only solution to solving the performance and energy bottleneck of the interconnects.

2.1.3.3 Sustaining reliability of operation

Even with constant device failure rates, technology scaling makes sustaining system level reliability substantially more difficult. At the architecture level, the compute system must have the ability to identify system issues such as failures, and should to be able to operate through and optimize performance with available system resources. The three major causes of failures in scaled technologies have been identified as i) *hard failures*; ii) Single-Event Upsets (SEU) and iii) Variability. In order to counteract these failure mechanisms, encoding using error correction codes, sparing and replication have emerged as the popular design choices (68). We note that these resiliency techniques are extremely popular for protection of data stored in the memory (49, 70). The primary reason being the overhead due to these resiliency techniques are easily amortized over the size of

the memory being protected. On the other hand, note that the random logic or custom datapath are difficult to protect from manufacturing defects and runtime failures and can be addressed only through replication of the entire logic unit. A computing framework which uses regular memory array for computing is therefore more robust in the presence of manufacturing defects and runtime failures.

2.2 Motivation

2.2.1 A summary of requirements

The technology trends as put forward in the previous section point to following simple conclusions - i) *Performance can be achieved through parallelism*. Although Amdahl's law puts an upper bound on the maximum speed-up that can be achieved for an application through parallel-operation, J. Gustafson correctly noted that for massively parallel machines, it is possible to compute large data sets in the same amount of time (71). ii) *Efficiency is in locality*. Clearly the performance and energy overhead due to movement of data can easily surpass the energy spent in useful computations. Hence only localized computation with minimum data transport can keep the energy efficiency improving with technology scaling. This only can be achieved by co-designing the hardware and the software (69). It is therefore worthwhile to explore the design of novel computing frameworks which:

- improve the energy efficiency for current reconfigurable frameworks by reducing the contribution from the programmable interconnects.
- leverage on the benefits offered by emerging non-CMOS memory technologies.
- are robust to failures resulting from parametric variation.

Figure 2.8: A comparison of the bandwidths at different levels of the memory hierarchy in a modern computing system.

2.2.2 Hardware acceleration

The above requirements apply to any general purpose computing framework. With growing demand for low-power high-performance application execution, these *hardware accelerators* are playing a major role in speeding up diverse applications from different application domains (9, 24, 72, 73). FPGAs, graphical processing units (GPUs) are hardware acceleration devices which are commonly used today. A study of the algorithm *hotspots* which are commonly mapped to these hardware accelerators is presented in (14). These include dense and sparse matrix operations, signal transforms, mapreduce, graph traversal, dynamic programming, structured and unstructured grid etc. A look at the performance limitation of these *dwarfs* on software reconfigurable system reveals that most of these are limited by memory bandwidth and latency, rather than by computing resources. Common *hardware accelerators* such as FPGAs and GPUs partly solve this problem by integrating an on-chip memory hierarchy and having many operations in parallel to effectively hide the data transfer delay.

2.2.3 Bandwidth is not scaling

FPGAs and GPU fall in the broader category of non-Von-Neumann computing systems since the compute logic is in close association with the data, though

FPGA is a hardware reconfigurable framework and GPU is a software reconfigurable framework. Although they have been found to offer *10times* to *1000times* improvement in energy-efficiency compared to a single general-purpose processor, they are finally limited by off-chip memory bandwidth. And similar to interconnect, off-chip bandwidth is not scaling at the same rate on-chip logic density. Thus as pointed out in (16), while the logic density increases by a factor of $16\times$ from 2011 to 2024, the relative bandwidth only increase by $1.4\times$ over the same period of time. As a result for data-intensive applications, both GPU and FPGA will soon be limited by off-chip data bandwidth (16). The scenario is however completely opposite if a computing framework that resides inside the memory is considered. As shown in Fig. 2.8, the data highest bandwidth can be achieved inside the commercially available non-volatile memory technologies such as Flash. The large bandwidth is mainly due to the wide (2KB) page read performed in each of the individual Flash blocks (total number of blocks is typically 4096) at $\sim 15ns$. The bandwidth however sharply falls as data is transferred out from the Flash at an I/O interface which is only 16-bits wide. The same scenario holds for data bandwidth inside the conventional DRAM. Note that the Intelligent RAM (IRAM) (74) was implemented to leverage on this observation and mitigate the Von-Neumann bottleneck. From Fig. 2.8, we note that compared to internal DRAM bandwidth, the bandwidth between the main memory and the processor is order of magnitude less.

2.2.4 Opportunities provided by emerging technologies

In the quest of a potential alternative to CMOS at the end of its roadmap (47), multitude of research efforts have been directed towards investigating novel devices with interesting and unique switching characteristics. These emerging research devices are primarily classified into two types:

- *Memory devices:* Some of the promising memory devices include a) spin torque transfer random access memory (STTRAM) (75), b) phase change

random access memory (PCRAM) (76), c) resistive random-access memory (RRAM) (77) d) nanoelectromechanical memory (NEMM) (78) and e) molecular memory (79). One important characteristic that distinguishes the emerging memory devices from existing memory technologies such as static random access memory (SRAM) and dynamic random access memory (DRAM) are that all the above memories are non-volatile. This implies that configuration can be stored indefinitely in these memories without requiring any supply, consequently facilitating low-power applications.

- *Logic and Alternative Information Processing Devices:* These array of devices include a)single-electron transistors (SET) (80), b) carbon nanotube field effect transistor (CNTFET) (80), c) semiconductor nano-wires, d) quantum-dot cellular automata (QCA) (80) and e) molecular transistors (80).

Although most of these emerging nanodevices are still in their infancy, they hold tremendous potential in terms of integration density (in the order of 10^{10} devices/ cm^2) (80). low power operation and higher switching speed. An extensive review of these emerging devices can be obtained in (80) and (80). In spite of differences in the working of these individual devices, they bear the following common characteristics:

- Each of these technologies are still at their infancy and require interfacing with conventional CMOS technology for reliable operation.
- Inexpensive fabrication methodology which can support large scale manufacturing of nanoscale devices with reproducible behavior is still being investigated.
- Both the emerging memory and logic devices are characterized by high density and are more amenable towards the fabrication of regular structures.
- Due to limitations in fabrication techniques, these nanoscale devices are characterized by high defect rates and large variability.

In lieu of the above characteristics, it is necessary that a computing model beneficial for the emerging nanodevices should bear the following attributes:

- In order to leverage on the advantages such as density and low power of operation as offered by the nanodevices, it is desired that computation is actually performed using these nanodevices and CMOS is used for purposes such as level restoration and interconnection between computing blocks.
- Interface between CMOS and the nanoscale devices should be few and well defined. Due to inherent difficulty in alignment and accurate patterning of the emerging nanodevices, the computing model must be able to tolerate any variability in the CMOS-nano interface.
- The model should also account for the large defect rate for the emerging nanodevices and provide mechanisms of defect tolerance without any significant overhead.

In perspective of the attributes required for the nanocomputing models, memory based reconfigurable computing has emerged as one of the primary choice. The primary advantages offered by memory as a computing platform in nanoscale architectures are:

- As mentioned emerging nanodevices are highly amenable for dense and regular circuit design. Recently, researchers have reported successful fabrication of 160 Kbit of molecular memory patterned at 10^{11} bits per square centimeter (81). Such dense memory arrays are realized using either by self-assembly process or by nano-imprinting method (80).
- Memory preserves the density advantage offered by the emerging nanodevices. Since these nanodevices need to be interfaced with CMOS logic, a dense nanoscale memory structure can substantially reduce the CMOS overhead for a single memory bit. Moreover, memory structures realized using nanodevices have very well defined CMOS interfaces thus facilitating CMOS-nano hybridization.

- It is well known that redundancy or error-correction based defect tolerance schemes for random logic incur significant overhead. However, due to the regular nature of the memory, high defect rates can be tolerated by either re-mapping to non-defective locations, by the use of redundant rows and columns or by using error correction schemes (80). Overhead incurred per bit is much less in the latter case than for random logic.

2.2.5 Proposed solution

In this work, we propose a novel memory based hardware reconfigurable framework (MBC framework) to address the challenges of technological scaling as summarized in the previous sections. The model is independent of the underlying memory technology as long it satisfies memory read and write operations. The memory based computing model thus applies to both on-chip as well as off-chip memory. In particular, the framework has the following key advantages:

- The framework is spatio-temporal in nature. This increases the percentage of local computation at each compute node of the proposed framework, thus minimizing the data movement across the framework and improving the energy-efficiency. With effective contribution from programmable interconnects reduced, this framework enjoys a better technological scalability compared to fully spatial reconfigurable frameworks.
- The basic computing fabric for the framework is memory. Computations of arbitrary complexity can be suitably partitioned and mapped as Look-up Tables (LUTs) to this memory array which also stores the data which is to be processed. Computation and storage are therefore at the same location which mitigates the well-known Von-Neumann bottleneck.
- Since the main computing fabric is memory, the framework can leverage on the benefits of a number of emerging non-volatile memory technologies, namely large integration density, low power, high read speed and non-volatility.

- Programming the proposed framework is facilitated through the use of standard instructions and the input application is automatically scheduled by the software flow in space and time to maximize the energy benefit of temporal computing without sacrificing the performance benefit of spatial computing. Data movement across the storage hierarchy is also explicitly controlled by the application mapping tool.
- Leveraging on the fact that memory arrays can be easily reconfigured around defects and failures by using redundant rows and columns (70), the proposed framework improves reliability of operation in nanoscale technologies.
- The proposed framework can act as a hardware accelerator for compute-intensive applications by mapping complex datapath as well as to the data-intensive applications by storing the data to be processed inside the same framework and leveraging on the data high-bandwidth available inside memory.

As opposed to a pure software based execution which is often non-optimal for many algorithmic tasks, the proposed hardware reconfigurable framework can improve performance and energy through techniques such as pipelining and parallel processing. As opposed to conventional FPGA frameworks which are fully spatial and are dominated by programmable interconnect, the proposed framework is spatio-temporal and reduces the contribution from interconnects. Finally unlike the GPU, it can map arbitrary applications using Look-up tables and perform bit-level computations which imparts higher flexibility compared to a GPU platform.

3

Memory based computing - overview

Memories are typically associated with data storage in computer system. They either store data temporarily (for example by volatile memories such as SRAM, DRAM etc.) or permanently (for example by non-volatile memories such as Flash, magnetic disks etc.). In this chapter we first draw the outline how the embedded memory may be used for computing as well in a time-multiplexed hardware reconfigurable system. In particular, this chapter makes the following key contributions:

- We propose a hardware architecture to utilize a dense 2-D memory array for the purpose of reconfigurable computing. The main idea is to map multiple multi-input multi-output LUTs to the embedded memory array at each compute block and evaluate them over multiple clock cycles. The proposed framework is therefore a spatio-temporal framework unlike the conventional hardware reconfigurable frameworks which are fully spatial. In the proposed hardware architecture, multiple computing elements communicate with each other over a time-multiplexed programmable interconnect.

- Along with the hardware architecture we outline an effective software framework which maps an input application to the proposed hardware reconfigurable framework. The software flow outlined partitions the input application and maps it to multiple compute blocks, and finally places and routes the mapped design.
- We describe the application domains of the proposed memory based computing model. We identify that the model can be applied to realize a stand-alone reconfigurable framework for mapping random logic as well as it can be used for hardware acceleration of algorithmic tasks.

3.1 Computing in memory versus computing with memory

Computing in memory or processing in memory was made popular by projects (74, 82, 83) which attempted to mitigate the Von-Neumann bottleneck. In all these approaches, a software reconfigurable framework was placed in close proximity with DRAM based off-chip main memory so that the performance and energy overhead in transferring the data and the instruction from the off-chip memory to the on-chip memory is minimized. The large data bandwidth available in off-chip memories such as DRAM is exploited by instantiating vector processors which can perform vector operations over the data which is loaded in parallel from the memory. Efforts have also been made in integrating DRAM memory with reconfigurable logic (74) so that hardware reconfigurable systems can also leverage on the concept of *Computing in memory*. Note that however, in all these systems, the memory is only used to hold data and in some cases instructions.

Computing with memory is a completely different concept from *Computing in memory* since it actually uses the memory arrays for computing. In order to explain the use of memory in computation, we will try to answer the following questions:

3.1 Computing in memory versus computing with memory

1. How can we perform computation in memory?

Ans. The main idea is to partition the input application into smaller control and data flow graphs, map the individual graphs using Look up Tables (LUTs) and map these LUTs to the memory array. The memory then holds not only the data, but also the function which will operate on the data. Note that a single memory array can hold one LUT, in which case the MBC framework is fully spatial or multiple LUTs can be mapped to a single memory array and evaluated over multiple clock cycles, thus creating a temporal computing model.

2. Which functions should we compute in memory?

Ans. Since the size of a LUT grows exponentially with the number of inputs and linearly with the number of outputs, it is desired that the function which is mapped as a LUT to the memory array has relatively small number of inputs and outputs. If the number of inputs and outputs to the original function is very large, then it is compulsory that the function may be bit-sliced into smaller operations. Arithmetic operations such as addition and multiplication and logic operations such as “and”, “xor” etc. can be easily bitsliced and mapped as LUTs to the memory array. The caveat over here is however, delay and energy for executing these arithmetic and logic operations in a custom logic is much smaller than compared to their evaluation using a LUT based approach with the LUT mapped to the memory array. However, the effectiveness of the memory based computing approach becomes more prominent for complex functions such as trigonometric and transcendental functions (84). It has been demonstrated that it is possible to decompose these functions in a spatial manner and evaluate them using multiple LUTs (85). The area and delay advantage of memory based computing has also been shown to hold true for functions which comprise of random logic. In case of mapping such functions to a LUT based hardware reconfigurable framework such as FPGA, it is possible to *fuse* multiple small logic functions and form an arbitrary complex function (86, 87) which is mapped to the memory array as a multi-input multi-output LUT. This minimizes the total number of operations in the function, thereby improving the area and performance for the mapped application. *In summary, applications which have complex datapath or comprises*

3.1 Computing in memory versus computing with memory

of simple operations which can be merged into a complex function with limited number of inputs and outputs are amenable to memory based mapping.

3. When do we perform memory based computation?

Ans. The idea of memory based computing applies to both software and hardware reconfigurable systems. For software reconfigurable systems, the idea of storing the LUTs in memory for function evaluation have been applied in diverse contexts such as: i) *Performance Improvement*: LUTs stored in on-chip memory have been widely used for performance improvement in software reconfigurable systems. Their primary use is in storing the pre-calculated results for: i) complex trigonometric functions; ii) transformed image pixel in image processing applications such as RGB to grayscale conversion and iii) complex logic function such as *population function* (88). Computing the results for such functions in software is expensive in terms of performance. Looking up the pre-computed value and possibly later interpolating two or more values to achieve higher accuracy is much more beneficial in such scenarios. ii) *Reliability Improvement*: In (89), we have demonstrated that it is possible to use memory based computing for improving the reliability for modern processor systems. The idea is to bitslice common arithmetic operations such as addition and multiplication and store the corresponding LUTs in on-chip cache. In case, the custom adder and multiplier inside the processor is faulty (either due to manufacturing defects or parametric variation), these operations can be migrated to the on-chip memory and calculated over multiple clock cycles. We however noted that the abundance of narrow width operands in general purpose applications reduces the performance overhead for memory based computation and improves the cycles per instruction (CPI) for a processor system with faulty execution units.

Memory based computing is more effective for the case of hardware reconfigurable frameworks which often partition the input application into smaller partitions and represent these individual partitions using LUTs. For both spatial (86, 87) and spatio-temporal hardware reconfigurable frameworks (90, 91, 92), memory based computing have been shown to improve the area, delay and energy for the mapped application compared to mapping the input application to

small 1-D distributed LUTs. The unique feature of a memory based hardware reconfigurable framework is that in such a framework, fabrics for computation and data storage are same, i.e. *memory*. It is therefore easier to realize a *Computing with memory* hardware framework on a *Computing in memory* platform. The rewards of such an integration is significant since the delay and energy associated with movement of data into and out from the memory is completely eliminated.

3.2 Overview of hardware and software architectures

In light of the above benefits, we have proposed a novel memory based computing (MBC) model. The hardware reconfigurable framework which embodies the proposed model is hereafter referred to as the memory based computing framework. The key highlights of the proposed computing model are as follows:

- i. It maps the input application as multiple multi-input multi-output LUTs and stores these LUTs inside a dense 2-D memory array. The application is then executed by accessing these LUTs topologically over multiple clock cycles. In doing so it mimics a temporal computing framework.
- ii. For sufficiently large applications which cannot be mapped to a single 2-D memory array, the model proposes a methodology to suitably partition the input application and map them spatially across compute elements, each of which is temporal in nature.

Fig. 3.1 illustrates the memory based computing model. The target application is partitioned into multi-input multi-output LUTs which are mapped to dense memory arrays (referred to as as function table) inside each MLB. Information regarding the address, scheduling steps and connectivity among the partitions is stored in a microcode format in a small ip op array (referred to as schedule table) during the application mapping phase. A register file holds the intermediate

Figure 3.1: Functional block diagram of memory based computing.

partition outputs. The schedule table, the function table and the register file forms the core of the MLB. Inside each MLB, the partitions are evaluated in a topological manner over multiple cycles. The requirement for communication between multiple MLBs arises when dependent partitions are mapped to two different MLBs. Inter-MLB communication is achieved through a FPGA-like PI framework.

3.2.1 Distinction with existing hardware frameworks

3.2.1.1 Comparison with conventional FPGAs

MBC bears marked distinction from conventional FPGA frameworks from the fact that the latter is a fully spatial computing framework which maps the input application as distributed small 1-D LUTs. As a consequence, FPGAs require an elaborate PI network for placement and routing of the LUTs. Although the requirement for interconnect has been reduced by replacing the individual LUTs with LUT clusters, the programmable interconnect has still been observed to contribute to 80% of the power, 60% of the delay and the 75% of the FPGA area in nanoscale technologies (12). The proposed MBC model differs from the conventional FPGAs on the following points:

- i. While FPGA is a fully spatial computing model, MBC is a spatio-temporal model which minimizes the requirement for programmable interconnect by multi-cycle localized execution inside each compute element of the framework.
- ii. In FPGAs, the input application is mapped to small 1-D LUTs, while the same is mapped as multi-input multi-output LUTs stored in dense 2-D memory arrays in the MBC framework.
- iii. Multi-MLB communication in the MBC framework utilizes the programmable interconnect in a time-multiplexed manner. FPGAs being fully spatial cannot leverage on the benefits of time-multiplexing the PI.

3.2.1.2 Comparison with EMB based FPGAs

As emphasized in (86, 87, 93), the embedded memory arrays inside conventional FPGA fabrics can be used for logic computations when they are not configured as on-chip memories. Fig. 3.2 shows the packing of multiple smaller LUTs to larger multi-input multioutput LUTs mapped to the memory array. Such a mapping of smaller LUTs to larger embedded memory arrays can improve i) the

Figure 3.2: Packing of logic nodes into multi-input multi-output partitions mapped to embedded memory blocks (87).

total area required to map a given netlist (86, 87); ii) a delay-oriented mapping algorithm (93) achieves significant improvement in performance of the mapped application. We refer to these frameworks as ‘EMB based Heterogenous FPGA’. Although these frameworks employ conventional two-dimensional (2-D) memory array for logic computation, it relies on a purely spatial computing model similar to that of conventional FPGA. The proposed MBC framework is similar but still different from EMB based heterogenous FPGAs on the following points:

- i. Similar to EMB based heterogenous FPGAs, MBC also partitions the input application as multi-input multi-output LUTs and maps them to embedded 2-D memory arrays.
- ii. Due to the use of larger multi-input multi-output LUTs, EMB based heterogenous FPGAs have less interconnect overhead as compared to conventional FPGAs. However, it is still a fully spatial framework and therefore cannot leverage on the time-multiplexing of the PI as can be achieved in the spatio-temporal MBC framework.
- iii. In EMB based heterogenous FPGAs, a single LUT is mapped to a 2-D memory array. To leverage on the integration densities of nanoscale memory technologies, current memory arrays are typically large (8-32KB) which allows multiple multi-input multi-output LUTs to the same 2-D memory array. A EMB based heterogenous FPGA cannot therefore benefit from

such a scenario. A temporal model at each MLB of the MBC framework allows multiple LUTs to be stored inside the memory array and evaluated in a topological manner over multiple clock cycles.

3.2.1.3 Comparison with time-multiplexed hardware reconfigurable frameworks

A number of time-multiplexed hardware reconfigurable frameworks have been proposed till date (20, 94, 95). The main idea in all these frameworks is to store the configuration for multiple applications in a small memory local to each compute element and routing switch so that it is possible to migrate from one application execution to another with minimum reconfiguration overhead. As demonstrated in (20), such a framework can significantly improve logic utilization on a hardware reconfigurable framework through temporal pipelining and even improve performance. A number of partitioning (96) and scheduling algorithms (94) have also been proposed for such time-multiplexed reconfigurable frameworks. MBC bears some similarities and certain differences with these time-multiplexed frameworks. They are as follows:

- i. Similar to the time-multiplexed architecture, configuration bits for a number of applications can be loaded into the each compute block MBC framework and they may be executed in a time-multiplexed fashion.
- ii. Unlike time-multiplexed frameworks, MBC however does not allow time-multiplexed inter-MLB routing switches. Hence only those applications which have similar interconnection graph can be mapped to the MBC framework and executed in a time-multiplexed fashion.
- iii. For a single application, previous time-multiplexed frameworks would still behave as a fully spatial framework and therefore incur large overhead in programmable interconnect.
- iv. In (20, 94, 95), the input application is still mapped to small 1-D LUTs at each compute node. On the other hand MBC advocates a temporal

execution model at each computing node by mapping multiple LUTs to a dense 2-D memory array.

3.2.1.4 Comparison with spatio-temporal hardware reconfigurable frameworks

A number of spatio-temporal hardware reconfigurable frameworks have been proposed till date (9, 24, 97). All these however are coarse grained reconfigurable frameworks which perform multi-cycle execution at each compute node by incorporating wide arithmetic logic unit (ALU) and storing the configuration bits in a small local memory at each compute node. Such frameworks also incorporate scheduled routing switches (98) similar to time-multiplexed reconfigurable fabrics. Note that communication in such frameworks is often on a word-basis (32-bits), so that the overhead for scheduling the interconnect is often amortized over the buswidth. MBC has the following differences with the spatio-temporal frameworks proposed till date:

- i. As opposed to (9, 24, 97), MBC is a fine-grained reconfigurable framework. In contrast to (9, 24, 97) which incorporate an ALU for multi-cycle execution, in the MBC framework, multi-input multi-output LUTs representing the input application are mapped to a dense 2-D memory array for the purpose of temporal computing.
- ii. Interconnect is statically scheduled in case of MBC. Hence, MBC poses unique placement and routing challenges, different from those addressed in (98).

3.3 Application of MBC

We believe that the proposed MBC model can be used for computing in diverse contexts for a variety of application domains. Following are some of the appli-

cation areas where the effectiveness of the MBC framework has already been validated. Other areas are topics of future research.

3.3.1 Motifs

The proposed memory based computing framework may be used for the purpose of: i) improving the performance; ii) energy and/or iii) reliability of the computing system.

- Interconnect performance and I/O bottleneck are the major limiters to performance improvement in modern computer systems (16, 20). They have poor technology scalability which will only degrade the performance in future computing systems. A memory based computing framework can be used to improve performance by: a) computing the data in close proximity with the memory array in which it is stored, thereby mitigating the I/O bottleneck; b) advocating a temporal computing model at each compute node, it can minimize the requirement for programmable interconnect.
- Similar to performance, energy overhead primarily comes from the movement of data from and to the memory. In MBC, computing is performed closest to the location of data storage and is therefore much effective in reducing the energy requirement for the computing system.
- In nanoscale technologies, both logic and memory are prone to manufacturing defects as well as parametric failures. Redundancy based defect tolerance is a commonly approach in handling such failures. Redundancy in logic however incurs significant overhead in terms of area and power, since the entire logic unit needs to be replicated (99). Failures in memory can however be easily tolerated by incorporating redundant rows and columns in the memory array and replacing the faulty ones with the functional ones. Hence memory based computing can be effective in improving the reliability of a computing system as well.

3.3.2 Contexts

The memory based computing model can be applied in diverse contexts. Two specific applications we have already investigated are as follows:

- As a stand alone reconfigurable framework used for LUT based mapping of random logic. We have already demonstrated that mapping the input application into multiple multi-input multi-output LUTs, storing the LUTs into a dense 2-D memory array and evaluating them over multiple clock cycles can be extremely effective in reducing the number of LUTs as well as the programmable interconnect requirement for such a framework.
- As a hardware accelerator it can be used to accelerate both compute and data intensive sections of algorithmic tasks from different domains of applications. Such a framework is hereafter referred to as MAHA - an abbreviation for MAlleable Hardware Accelerator. The idea is to instrument an existing memory organization and augment it with additional logic so that it can be transformed from a simple storage system to a memory based hardware reconfigurable framework on demand. The chief benefit with such an arrangement is that MAHA can potentially mitigate Von-Neumann bottleneck (10) since it can migrate the application to the data rather than the data to the application. In such framework, the additional logic not only includes control logic for memory based computing but also custom logic for mapping datapath operations such as addition, multiplication etc. It should be noted here that the instrumentation must be performed in a manner that incurs minimal overhead to the memory integration density and performance.
- The third context is to apply the proposed MBC model to commercially available general purpose hardware reconfigurable frameworks such as FPGA. The idea is to map the input application in a spatio-temporal manner on the FPGA itself, with each MLB built in close proximity with the memory in which the data is being stored. This idea is being currently evaluated.

3.3.3 Domains

Following are the domains of applications which benefit from memory based computing:

- Applications which are rich in complex functions such as trigonometric and transcendental functions as well any other compute intensive functions are amenable to memory based computing. These include applications from diverse domains such as scientific computing, cryptography, image processing, informatics (100) and general purpose signal processing (2).
- Since the proposed MBC framework uses LUTs to map the input application it is amenable for mapping functions with random logic. Applications where simple logic functions can be fused to form complex logical operations which may be mapped as LUTs to the memory array are also attractive for memory based computation.
- Similar to FPGAs, MBC is a generic reconfigurable framework which can map any function. Compared to their logic implementations, operations such as integer and floating point addition and multiplication incur considerable performance and energy overhead when mapped as LUTs. Note that however, for many applications such as signal processing application, the operand width is limited to 16 bits or less (89, 101). In such scenarios, MBC can be used to substitute multiple datapath operations with a single lookup operation, thus improving performance and energy.

4

Hardware framework

This chapter describes the hardware architecture for MBC based stand alone reconfigurable computing as well as for the MAHA framework. The key contributions in hardware architecture are as follows:

- For stand alone hardware reconfigurable computing, it replaces the conventional fine-grained fully spatial computing model with a mixed-granular spatio-temporal computing model. Sparse 1-dimensional LUTs as present in conventional FPGA platforms is replaced with a dense 2-D memory array which can support multi-input multi-output LUTs of larger size. Since a large memory array can map multiple LUTs of varying input and output sizes, a temporal computing model has been developed which advocates multi-cycle time-multiplexed execution inside each compute block (MLB) of the proposed computing framework. The main fabric of this computing unit is the 2-D memory array to which the LUTs are mapped. Multiple such computing units communicate with each other to form a spatio-temporal computing model.
- The major contribution to area, power and delay for the present hardware reconfigurable frameworks come from programmable interconnects. The proposed hardware architecture minimizes the requirement for an elaborate

routing framework by multi-cycle execution within each compute block. Essentially, the distributed configuration bits for routing is replaced by a dense and energy-efficient register file implementation which holds the microcode for multi-cycle operation inside each MLB. A small routing framework local to each MLB is responsible for selection of data for LUT based evaluation. This routing framework being local to each MLB scales better with technology compared to the global interconnects in conventional FPGAs.

- The hardware architecture for the MAHA framework builds on an existing memory organization to transform it from a simple storage device to a hardware reconfigurable framework on-demand. The architecture preserves the integration density for modern memories by building the additional logic required for reconfigurable computing around the 2-D memory modules into which a large memory is conventionally divided. As a result, it does not replicate the decoders and hence does not affect the normal functionality of the memory. The large bandwidth available in the form a hierarchical interconnect inside the memory is utilized for communication between multiple compute blocks during operation as a hardware reconfigurable unit. The performance and energy benefits achieved through the proposed *computing in memory* approach, justified the small amount of additional hardware required for such a transformation.

4.1 A memory based generic reconfigurable framework

A generic reconfigurable framework is one which can map logic functions with arbitrary complexity. It is well-known that with no resource constraint, a look-up table (LUT) based implementation will eventually be able to map any logic function. The MBC framework for stand-alone reconfigurable computing as described here therefore considers only LUT based logic evaluation. In later sec-

4.1 A memory based generic reconfigurable framework

tions, we will investigate how to incorporate dedicated datapath elements in the MBC framework for evaluation of common arithmetic operations such as addition, multiplication and logic operations such as shift. Note that the addition of these datapath elements do not change the basic functionality of each MLB or the inter-MLB communication as described in this section. It merely adds more compute resources to each MLB.

4.1.1 μ -architecture for memory logic block

Each compute block of the MBC framework is referred to as the Memory Logic Block or MLB. Each MLB is responsible for LUT based evaluation of multiple logic partitions mapped to the respective MLBs over multiple clock cycles. The hardware organization of each MLB along with the design parameters are illustrated in Fig. 4.1. Before describing each of the MLB components in detail, it is essential to define the term *Granularity*. For any signal internal to a MLB, *Granularity* refers to the number of bits for the signal which may be selected separately. For example, if $a[11:0]$ denotes as 12-bit signal and *Granularity*=1, then $a[0]$, $a[1]$, ... $a[11]$ can be selected separately. However, if *Granularity*=4, then $a[0]..a[3]$ can only selected together. Note that to ease the design process, all design parameters discussed below are always expressed in multiples of *Granularity*. Each of the MLB components are now described below in detail.

4.1.1.1 Function table

The principal component of each MLB is the *function table*. It is represented by a dense 2-D memory array to the which multi-input multi-output LUTs are mapped and evaluated over multiple clock cycles. The design parameters which define the *function table* are listed below:

- *Log_num_slices*: The implementation of the memory array inside each MLB lends itself to the opportunity of more than one memory access in each clock cycle. This means it is possible to evaluate two or more LUTs in the same

4.1 A memory based generic reconfigurable framework

Figure 4.1: μ -architecture of a single memory logic block (MLB). Figure show the essential hardware components, namely: i) temporary and dummy registers; ii) muxtree; iii) function table and iv) schedule table.

clock cycle within a given MLB. This is possible if the memory itself is implemented as multiple slices (or arrays), which may be accessed independent to each other. Modern memories provide opportunities for similar implemen-

4.1 A memory based generic reconfigurable framework

tation without compromising the integration density of the memory array (102). The MBC model abstracts the underlying memory implementation here and presents a parameterized interface to the mapping tool for application mapping. The maximum number of parallel memory accesses within a given clock cycle inside each MLB is thus calculated as $2^{\text{Log_num_slices}}$. Note that however, for $2^{\text{Log_num_slices}}$ parallel memory accesses, the same number of address decoding is required.

- *Row_subarray*: The number of rows in the 2-D memory array used to realize the function table. The term *subarray* is used in keeping with the MAHA architecture where the memory module in the lowermost hierarchy of a hierarchical memory organization is referred to as the *subarray*.
- *Col_subarray*: The total number of columns in the 2-D memory array used to realize the *function table*. This value is irrespective of the number of slices in which the *function table* is implemented. The number of columns per slice is therefore calculated as $\text{Col_subarray} / (2^{\text{Log_num_slices}})$.

Based on the above design parameters, it is possible to derive the following mapping parameters which facilitate mapping LUTs to the *function table*.

- *Utblky*: If the *function table* supports a total address width of N , then it is possible to access any memory location with an address of bitwidth $M \leq N$. That implies, that *function table* can essentially support any M -feasible LUT where $M \leq N$. In such a scenario, to achieve maximum efficiency in packing the LUTs into the *function table*, the starting address for the LUT accessed needs to be stored beforehand and added with the LUT address to access the actual memory location (refer to Fig. 4.1). To allow the LUTs to start from any location in the *function table* implies the framework needs to store a large starting address for each LUT and add to it the corresponding LUT offset. This degrades both area as well as performance for the MLB. For mapping LUTs, the *function table* has therefore been divided into number of virtual segments both along rows as

4.1 A memory based generic reconfigurable framework

well as columns. *Utblk_y* refers to the number of rows inside each segment. A LUT can only start from an *Utblk_y* boundary. The total number of segments along the rows is therefore $Row_subarray/Utblk_y$.

- *Utblk_x*: Similar to segmentation along the rows, the columns are also divided into virtual segments. *Utblk_x* refers to the number of columns in each such segment. It also defines the maximum number of outputs possible from a LUT. *Utblk_y* and *Utblk_x* together define the virtual unit block with which the *function table* can be thought to be composed of.
- *Max_LUT_input*: The number of rows and columns of the *function table* memory array, the number of slices and the *Utblk_x* parameter together define the maximum LUT input size possible for a given MLB design. The formula is given below: $Max_LUT_input = \log_2(Row_subarray) + \log_2(Col_subarray/(Utblk_x \times 2^{Log_num_slices}))$.

4.1.1.2 Register file

At the end of each clock cycle, outputs from the *function table* are latched into the temporary register. At the beginning of the next clock cycle, contents of the register file are read to form the address for LUT access. The location in the register file to which a particular slice of the *function table* writes is currently fixed. The flexibility however is provided in the selection of the operands from the register file to form the address for the LUTs. Two design parameters define the structure of the register file: i) *Reg_width* defines the width of each register file. In case of MBC based generic reconfigurable frameworks, the register width should be the same as *Utblk_x*. However, for the case where both *function table* and custom datapath is present, the width of the register should be set to the maximum of the output bitwidth of the *function table* and the custom datapath. ii) *Reg_count* defines the number of temporary registers used for storing the temporary variables inside the MLB. Although referred to as the *register file*, these temporary registers are better modeled using the flip-flops since the data should be available at the output of temporary registers as soon as it is written.

4.1 A memory based generic reconfigurable framework

If a conventional register file implementation is used for the *register file* (103), then a half cycle delay spacing must be maintained between the write and read operations.

4.1.1.3 Dummy registers

In addition to the temporary registers, the MLB architecture currently includes a set of dummy registers to store *feed-forward signals*. By feed-forward signals we mean the signals which are generated in cycle C_i but being used in cycle C_j , where $j > i + 1$. Since the requirement to read and write into these registers are 1 cycle apart, they may be realized using conventional register file implementation. Their structure is defined by the following parameters: i) *Dummy_reg_count*: which indicates the number of dummy registers in each MLB and ii) *Reg_width* which is same as the temporary registers. An alternative to dummy register would be to increase the size of the temporary registers and perform register renaming in order to preserve the feed-forward signals. Note that in such a scenario, the destination register for the function table and datapath output must be selected instead of writing the values to a fixed register file location. Investigations regarding the VLSI implementation of such temporary registers, either as distributed and hierarchical (104) register file or as multi-ported register (103) file will be part of our future work.

4.1.1.4 Schedule table

The sequence of operations inside each MLB is controlled by the microcode stored inside the *schedule table*. The structure of the *schedule table* is defined by the following parameters: i) *Maxlvl_per_MLB* which denotes the number of rows in the schedule table and is currently set to the maximum number of cycles possible in the cyclic scheduling policy followed in scheduling the vertices. *Maxlvl_per_MLB*= K indicates that vertices with levels from 0 to $K-1$ are evaluated in consecutive clock cycles before vertices with 0 start to be evaluated again. As shown in Fig. 4.1, a small counter and decoder is present inside each MLB to

4.1 A memory based generic reconfigurable framework

select the row of the schedule table to read out. The size of the counter is given by $\log_2(Maxlvl_per_MLB)$. ii) *Num_sched_bits* denotes the number of bits in each row of the schedule table. These control bits are responsible for the following inside each MLB: a) enable signal for the temporary register write which are set depending on the number of parallel LUT accesses and datapath operations that are scheduled in a given clock cycle; b) enable for the function table slices which are read in a given clock cycle; c) starting location (row and column) for the LUTs mapped to the function table; d) signals to select the source (MLB input, temporary register or the dummy register) for each LUT access and the location of the operand which form the LUT address; e) enables for the dummy registers to be written in a given clock cycle; f) select signals for selecting the source operands for writing into the dummy register; g) if multiple datapath operations are supported in the same clock cycle (i.e. addition and multiplication), then enables for each of these operations; h) select signals to select the source as well as the location of the operands for custom datapath operation; i) selecting the MLB outputs from the outputs of the function table and the datapath operations. Note that currently most of the control information is present in one-hot encoded manner, which increases the size of the schedule table. It is however possible to reduce the size for the schedule table by storing the encoded control bits and decode them after reading it from the schedule table. The trade-off between reduction in schedule table size at the cost of introducing additional decode logic will be investigated in future. In the proposed MBC framework, the schedule table is implemented as a conventional register file for the following reasons: a) the size of the register file is much larger compared to the temporary registers. Hence a more compact register file implementation significantly improves the area and energy requirement compared to a flip-flop based implementation of the same. b) the schedule table is only written during the process of reconfiguration of the MBC platform and is read every clock cycle. Thus there are enough clock cycles between consecutive read and write operations which allows the schedule table to implemented as a conventional SRAM based register file design (103).

4.1.1.5 MLB input and output

Design parameters $Num_DI_subarray$ and $Num_DO_subarray$ denote the number of inputs and outputs to each MLB. Signals which are inputs to each MLB at every clock cycle of the cyclic schedule are known during the process of application mapping. Control bits stored inside the schedule table are responsible for selecting the appropriate inputs among $Num_DI_subarray$ physical inputs to form LUT address, to form datapath operands or to forward them to the dummy registers as feed-forwarded signals. Selection logic should also be present at the output of the MLB to steer the outputs from each slice of the function table or the custom datapath to each of the $Num_DO_subarray$ physical outputs. Such a requirement primarily arises due to the fact that inter-MLB communication uses interconnect in a time-multiplexed manner, i.e. the same routing channels are used to route different signals over different clock cycles. Hence the same physical output will require transmitting the outputs from different function table slices over different clock cycles.

4.1.1.6 Muxtree

Similar to the clustered FPGA architecture described in (53), we consider a ‘fully connected’ approach for intra-MLB communication across multiple operations mapped to a given MLB. Selection of operands from the temporary register, the dummy register and the MLB inputs to form the LUT address or the datapath inputs inside each MLB is thus achieved through a local multiplexer network, hereafter referred to as the *muxtree*. Select signals for the *muxtree* is derived from the schedule table inputs. Design parameters which govern the area, delay and energy for the muxtree are: i) *Granularity*; ii) $Num_DI_subarray$; iii) Reg_count ; iv) Reg_width ; v) $Dummy_reg_count$; vi) Log_num_slices ; vii) Max_LUT_input . For e.g., if $Granularity=G$, $Num_DI_subarray=DI$, $Reg_count=R$, $Reg_width=W$, $Dummy_reg_count=D$, $Log_num_slices=S$ and $Max_LUT_input=M$, then the muxtree will consists of $(S \times M)/G$, $\{3 \times Max((R \times W)/G, (D \times W)/G, DI/G) to 1\}$ multiplexers. The number of inputs is multiplied by 3 since there are 3 possible

4.1 A memory based generic reconfigurable framework

Figure 4.2: a) Synchronization of operations across multiple MLBs; b) Critical path for multi-MLB communication; c) Sequence of events inside a MLB and across MLBs for execution of applications mapped to multiple MLBs.

sources for each LUT operand, namely, temporary register, dummy register and the MLB input. The *muxtree* serves as a local routing network inside each MLB, where the performance and energy is dominated by the logic delay and energy used to realize the *muxtree*. This is much unlike the programmable routing resources present in a FPGA, where both performance and energy is dominated by the interconnect parasitics. A transistor level custom implementation (as opposed to synthesis using standard gates) of the *muxtree* is preferred to its minimize delay, area and performance impact.

4.1.2 Sequence of operations inside MLB

Fig. 4.2(c) shows the sequence of events inside a MLB. These may be classified into two groups based on whether they occur during the positive or negative half of the clock cycle:

4.1.2.1 Events during negative half-cycle

The reason for describing the negative-half events first is that the schedule table read for an operation which is scheduled in cycle C_i is performed in the negative half of cycle C_{i-1} . Values read from the schedule table serve as control signals for the selection of operands during the positive half of cycle C_i . Note that some control signals such as those for MLB output selection are read from the schedule table in the same clock cycle. When schedule table is being read for operations scheduled for cycle C_{i+1} , *function table* is accessed in parallel for operations scheduled in cycle C_i . The addresses for LUT accesses generated during the positive-half of the clock period are latched using negative-edge triggered flops. During the negative-half cycle the *function table* is accessed using these addresses and LUT content is read out. In the MBC model, memory access operation during the negative half cycle does not consider the underlying memory technology. Without loss in generality we assume that the negative half-cycle delay is long enough to access the 2-D memory array under consideration. For MAHA framework, datapath operations also occur in parallel to *function table* access during negative half cycle of MLB operation. Note that for large *function table* size, the schedule table read and datapath operation are likely to be faster than accessing the 2-D memory array. Hence, memory access time is likely to determine the delay for negative half cycle of the MLB operation. At the end of the negative half cycle, outputs from the function table and custom datapath are latched into the temporary registers and MLB outputs are selected from the signals produced in the same clock cycle.

4.1.2.2 Events during positive half-cycle

Inputs to a MLB in cycle C_i are expected to arrive in the positive half of the clock cycle C_i . Control bits read from the schedule table in the negative half of cycle C_{i-1} are responsible for selecting the inputs to a MLB in cycle C_i as well as selecting the LUT address from operands stored in the temporary register, dummy register and the selected MLB inputs. The LUT addresses formed serve as offset which is added to the LUT base address stored in the schedule table to form the complete LUT address. Other operations include selection of the dummy operands to be written into the dummy register file. Unlike the negative half cycle, for which the duration is primarily determined by the memory array access time, the positive half cycle delay is primarily determined by the inter-MLB routing delay and intra-MLB delay through the *muxtree*.

4.1.3 Inter-MLB communication framework

When multiple MLBs communicate among themselves, the proposed MBC model is similar to other spatial computing frameworks. The major difference is that the same routing channels are used in a time-multiplexed manner, i.e, they carry different signals over multiple cycles. Thus, the maximum number of channels that need to be routed to MLB M_i is $\sum_j (Max(Ch_{i,j,k} \forall k))$, where $Ch_{i,j,k}$ is the number of signals that is transmitted from M_j to M_i in *cycle* k . Fig. 4.2(a) shows the multi-MLB communication framework and synchronization between the operation of multiple MLBs. MLBs A and B communicate with MLB C in *cycle* 2 and 3. The signals that need to be transmitted in *cycle* 2 are *Sig* 1 and *Sig* 3 and those in *cycle* 3 are *Sig* 2 and *Sig* 4, respectively. In spite of the number of signals being 4, only 2 channels (*Ch* 1 and *Ch* 2) are required to connect MLB A and B with MLB C . *Ch* 1 will hold the values of the signals *Sig* 1 and *Sig* 2 and *Ch* 2 will hold the values of the signals *Sig* 3 and *Sig* 4 in *cycle* 2 and *cycle* 3, respectively. Thus a time-multiplexed operation inside each MLB leads to time sharing of interconnects for inter-MLB communication. Fig. 4.2(a) and (c) shows the synchronization between the operation of multiple

MLBs. All the Level 1 ($L1$) partitions in MLB A, B and C are executed in the same clock cycle. For the execution of $L1$ partitions, primary inputs form the effective address for accessing the LUTs. Outputs from the function table are available on the channels $Ch\ 1$ and $Ch\ 2$ before the rising edge of the next clock cycle. In cycle $C2$, inputs to the LUT address space for MLB C may either come from $L1$ partitions of A and B and/or from $L1$ partitions locally evaluated in MLB C . Register read and portion of the multiplexor delay in MLB C is partially overlapped with the routing delay required by the operands to arrive from MLB A or B . It may happen that an input from cycle C_i is not used in cycle C_{i+1} , instead used in cycle C_k , $k > i + 1$. Such a feedforward input would continue to be latched in the dummy registers till cycle C_k . This ensures topological execution of the partitions.

4.1.4 Estimation of MLB cycle time

The cycle time for an application mapped to a multi-MLB framework is determined by the inter-MLB cycle time ($T_{Inter-MLB}$). Inter-MLB cycle time is different from intra-MLB cycle time $T_{Intra-MLB}$ which is true when MLBs do not need to communicate amongst themselves. While $T_{Intra-MLB}$ purely depends on the positive ($Delay_{poscyc}$) and negative ($Delay_{negcyc}$) half cycle delays within a MLB, the $T_{Inter-MLB}$ is dependant on the routing delay and can be different for different pairs of MLBs. The final cycle time after mapping a given design is however obtained by the maximum of all inter-MLB cycle times. $T_{Inter-MLB}$ is determined by the delay between two successive memory accesses in two communicating MLBs. Since each memory access starts at the negative edge of the clock cycle, the output from the function table in $MLB\ 1$ is required to pass through the routing network and form the partition inputs for memory access in $MLB\ 2$ before the negative edge of the next clock cycle. This inter-MLB routing delay ($Delay_{Inter-MLB}$) effectively increases the positive-half of the clock period, although a portion of it is masked by the positive half-cycle delay local to the destination MLB. The modified expression for the positive half-cycle is thus

Figure 4.3: Hierarchical organization of the memory array. The hierarchy consists of *banks*, *subbanks*, *mats* and *subarrays*. The interconnect is also hierarchical where the bandwidth decreases progressively down the hierarchy.

given as $\text{Max}(\text{Delay}_{\text{Inter-MLB}}, \text{Delay}_{\text{poscyc}})$ and the total inter-MLB cycle time is evaluated as $T_{\text{inter-MLB}} = \text{Max}(\text{Delay}_{\text{Inter-MLB}}, \text{Delay}_{\text{poscyc}}) + \text{Delay}_{\text{negcyc}}$. The concept is conveyed in Fig. 4.2(b).

4.2 MAHA hardware architecture

MAHA stands for Malleable Hardware Accelerator. The idea is to equip existing memory architectures with additional logic which would allow a storage-only system to be transformed into a hardware reconfigurable platform on demand. In this section, we describe the hardware architecture of the MAHA framework and steps for instrumenting the existing memory architecture to realize the same.

4.2.1 Existing memory organization

We refer to the CACTI 5.1 documentation (102) for a description of the existing memory organizations. As illustrated in Fig. 4.3, modern memory arrays are organized in a hierarchical fashion. The number of levels of in the hierarchy depend

on many factors, one of them being the total size of the memory array. For convenience, we adhere to the organization with 4 levels in the hierarchy. These levels are referred to as *Banks*, *Subbanks*, *Mats* and *Subarrays*. The number of elements in each level of the hierarchy again depend on a number of parameters such as total memory size, width of data input and output bus to the memory array, overhead for the decode logic and also on the constraints (namely area, energy or performance) for which the memory is being designed. As in (102), the number of elements in each of these levels are design parameters indicated by variables *Banks*, *Ndbl_2*, *Ndwl_2* and *Num_subarrays_per_mat* respectively. *Subarrays* are the elements at the lowest level of the hierarchy and actually represent the 2-D memory array with the associated decoding logic. The bitwidth for the input and output data bus to the entire memory array is denoted by *Buswidth*. Note that the interconnect inside the memory is also distributed in a hierarchical manner with the bandwidth progressively decreasing from the *Banks* to the *Subarrays*. The interconnect follows a hierarchical H-tree pattern (102) where the bandwidth at each level of the hierarchy is given as follows:

- *Num_DI/DO_banks*: Number of data inputs and outputs to each bank. This is same as the *Buswidth* since in memory only one bank can be accessed at one point in time.
- *Num_DI/DO_subbanks*: Data inputs and outputs to a bank gets divided into buses with bitwidth *Num_DI/DO_Subbanks* depending on *Ndbl_2*. The formula is given as $Num_DI/DO_Subbanks = Num_DI/DO_banks / Ndbl_2$.
- *Num_DI/DO_mats*: Data inputs and outputs to a subbank gets divided into buses with bitwidth *Num_DI/DO_mats* depending on *Ndwl_2*. The formula is given as $Num_DI/DO_mats = Num_DI/DO_subbanks / Ndwl_2$.
- *Num_DI/DO_subarrays*: Data inputs and outputs to a mat gets divided into buses with bitwidth *Num_DI/DO_subarrays* depending on *Num_subarrays_per_mat*. The formula is given as $Num_DI/DO_subarrays = Num_DI/DO_mats / Num_subarrays_per_mat$.

The total size of the memory array and the number of elements at level of the hierarchy indicate the number of rows and columns of memory cells present at the lowest level of the hierarchy, i.e. *subarrays*. The design parameters *Row_subarray* and *Col_subarray* define the size for each subarray and the corresponding decode logic.

Apart from the above design parameters, a number of other design parameters define the memory organization. Some of these are: i) Number of address bits; ii) Associativity if the memory array represents the cache data array; iii) degree of multiplexing at the sense amps etc. During instrumentation, these memory parameters are not modified. As a result, the framework can be seamlessly switched between a reconfigurable computing framework and a normal memory array without any additional consideration for normal memory operation.

4.2.2 Instrumenting memory for hardware acceleration

There are three key questions to be answered in instrumenting an existing memory organization to perform reconfigurable computing on demand. These are as follows:

- What should be the computing element of such a reconfigurable framework and how should it be realized?
- How can the compute blocks of the reconfigurable framework communicate among themselves?
- How can such framework be seamlessly switched between a normal memory array and a computing framework?

In the following text, we provide answers to the above questions.

4.2.2.1 MBC for MAHA and modifications to the MLB μ -architecture

The major challenge in instrumenting the memory array to perform reconfigurable computing is to add additional logic into the memory array without seriously

compromising the integration density of the array. This leaves the designer with two choices:

- Opt for a fully spatial computing architecture similar to the conventional FPGA. This implies the memory arrays should be implemented as small 1-D LUTs, connected to each other with through a programmable interconnect framework. In addition to LUTs, datapath elements such as custom adders and multipliers can also be part of this design. We note that a similar organization has been proposed previously in (105). In (105), the authors propose to model the memory arrays as multi-input multi-output LUTs, connected in a purely spatial manner with multiplexors and dedicated carry chains for implementing up to 2-bit adders and constant coefficient multipliers. The proposed design is used to realize a 2D-DCT transform. We note that such a design has several shortcomings when modern memory organization is considered. *Subarray* sizes in modern on-chip memories (such as SRAM array) are typically 8KB or more (102). For off-chip memories such as DRAM or flash, the minimum array size is much more (256KB (106)). It would be extremely inefficient to store the LUTs for 2-bit adders in such large memory arrays and compute them in a spatial manner. Moreover, in a typical memory organization, these arrays do not communicate directly to each other, but only do so with the primary input and output of the memory array through a hierarchical interconnect. Hence it cannot be assumed that the outputs from these subarrays can be cascaded through a dedicated carry chain.
- The other option is to implement a MBC framework on top of the existing memory organization. Although it has several implementation challenges, the reasons in favor of such an option are as follows: i) MBC is a spatio-temporal framework where each MLB executes multiple operations in a topological manner over multiple clock cycles. These operations are mapped as large multi-input multi-output LUTs to a dense memory array inside each MLB. Hence for the case of MAHA framework, it is possible to construct a MLB with one or more *subarrays* serving as the function table. These

Figure 4.4: Instrumenting each subarray in the memory organization to construct a MLB. The *subarray* itself acts as the *function table* to which LUTs are mapped.

subarrays would store the data as well as multiple LUTs for the application being mapped to the framework. ii) Localized temporal computing inside each MLB significantly reduces the demand for PI as required in a fully spatial framework. Hence it can leverage on the hierarchical interconnect already present inside the memory for inter-MLB communication. Clearly the resource utilization is better in case of MAHA.

Fig. 4.4 shows the changes made to the existing memory array to realize a multi-MLB framework. As evident from the figure, each of the subarrays have been augmented the additional logic. This additional logic corresponds the control and selection logic for the MLB architecture discussed earlier and includes: i) *schedule table*, ii) *temporary and dummy registers*, and iii) *muxtree*. We propose to use each subarray as the function table inside each MLB. Note that the choice to build a MLB around is each subarray is a design choice based on the maximum area overhead allowed in instrumenting the memory. It is possible to consider a group of N-subarrays as N-slices of the *function table* and build the additional MLB logic around these subarrays. The MAHA framework therefore scales well

with the hierarchical organization of current memory arrays. Following modifications are required at the interface of each subarray to use it for reconfigurable computing:

- In a typical memory organization, the outputs from each subarray in a bank are concatenated to form the bank output and hence the *buswidth*. For typical *buswidth*=256 or 512, and number of subarrays per bank = 8, the bitwidth for each subarray output is 32 or 64. However, the LUTs to which the input application is mapped typically has 4 to 8 bit outputs, as defined by the *Utblkx* MLB design parameter. Hence it is necessary to perform an additional level of column multiplexing to select the LUT outputs from the data that is being read out from the subarray.
- During normal memory operation, data is read out from the subarray using the address supplied on the address bits from outside the memory array. However, during operation as a MBC framework, the address to access the LUTs mapped to the memory array are generated from the operands inside the MLB. Hence based on the whether the framework is acting in normal memory mode or as a reconfigurable framework, proper address must be selected to access the subarray.

In contrary to the MBC based generic reconfigurable framework, which maps random logic in terms of LUTs, MAHA is expected to perform as a hardware accelerator engine for accelerating applications from diverse domains. It is well-known that common arithmetic operations such as addition and multiplication and logical operations such as shift/rotate and select operations appear heavily in applications from diverse domains (98). Many coarse-grained hardware reconfigurable frameworks have therefore been proposed (9, 17, 97) where each compute block incorporates the above custom ALUs which perform the above arithmetic and logical operations. The benefit of using custom datapath to perform these arithmetic and logical operations are twofold: i) performance and ii) energy. The benefits of using a custom datapath for the above operations has already been validated in the case of FPGA (38). In order to reap the same advantages, we

Figure 4.5: Instrumenting the hierarchical interconnect network in conventional memory organization to support multi-MLB communication.

propose to incorporate custom logic supporting i) addition, ii) multiplication, iii) shift, iv) rotate and v) select operations inside each MLB. Details on the organization of the custom datapath and its functionality are provided in the Appendix 10.

MAHA framework using CMOS-compatible flash memory technology (106) derives an additional benefit from the conventional flash organization. The flash memory is typically organized into multiple blocks (each block being $\sim 256\text{KB}$), each of which is organized into multiple pages ($\sim 2\text{KB}$). At each block of the flash memory, a page register with size same as one page is present to latch the page output. We note that this page register (denoted as *PR*) can not only serve as the temporary register for the MLB constructed from one block of the flash memory, but also as a temporary storage which can be accessed using special *loadPR* and *storePR* operations. Such a storage will behave as a large dummy register file, storing operands which must be preserved till the end of execution.

4.2.2.2 Multi-MLB communication in the MAHA framework

An important challenge in the implementation of the MAHA framework is achieving the multi-MLB communication. The challenge is primarily two-fold: i) multi-MLB communication must follow the hierarchical interconnect architecture (similar to a H-tree organization) that is already present as part of the memory organization. This is a severe constraint, particularly for applications which are communication intensive; ii) in case of a stand-alone memory model, the input and outputs for each module in the memory hierarchy comes from the primary input and outputs of the entire memory array. However, in case of MAHA framework, there must be architectural support so that each MLB may receive its inputs from the outputs of other MLBs. We propose to address the above challenges in the following manner:

- The first challenge is primarily addressed by the MBC model and the placement and routing algorithm used to place and route the MLBs in the hierarchical memory organization. The temporal computing model inside each MLB significantly reduces the multi-MLB communication requirement as compared to a fully spatial model such as FPGA. Because of the same reason, typical interconnect bandwidth present in the memory array has been found to be sufficient in routing applications as complex as *Motion estimation (ME)* and *Advanced encryption standard (AES)*. The placement and routing procedure followed in our software flow ensures that MLBs which have less communication are spaced furthest apart (in different *banks* or *subbanks*), while MLBs which require more communication are mapped in close proximity (*mats* and *subarrays* within the same *subbank*). Together, we have found that it is possible to satisfy the communication requirement for complex applications using the hierarchical interconnect framework of the baseline memory array.
- The second challenge is addressed by adding programmable switches at the input of each subarray, each mat, each subbank and each bank, such that each subarray, mat, subbank and subbank can either receive inputs from

the outputs of the modules at the same level of hierarchy or from the inputs to that hierarchy level. The idea is illustrated in Fig. 4.5. If there are N -modules (say *subarray*) in a given hierarchy of the memory organization (say *mat*), then the input to *subarray* can come from the outputs of each $N-1$ subarrays and the input to the *mat*. Now note that the input and output of each *mat* is N times the input and output from each *subarray*. Hence, the switch which selects the input to a *subarray* must be able to select an input from $N-1 + N$ possible sources, or effectively from $2N$ possible sources. The programmable switch used in our implementation is a simple $2N$ to1 multiplexer. The reason for use of multiplexer is to minimize the routing complexity since it offers full connectivity. Typical FPGA frameworks use complex routing switches (48) which allow multi-way connection at the cost of increase in the route complexity. In future, we will investigate the impact of different routing switch designs on routability of the mapped design. Note that for normal memory operation, the switches at a given level are set in such a manner that it receives the inputs from only the inputs at that particular level.

4.2.2.3 Modes of operation

Following are the modes of operation identified to alternate the MAHA framework between the normal memory mode of operation and operation as a hardware reconfigurable framework.

- *Normal memory operation:* Read and write are seamlessly supported to the memory array during this mode of operation. The data written into a particular address must be retrieved during read from the same address.
- *Reconfiguration of the function table:* When the MAHA framework transitions from the normal memory operation to the compute mode, function tables in the MLBs to which the input application is mapped are written first with the LUT contents. This is same as the normal memory write operation with the LUT contents being provided as the *datain* to the memory

Figure 4.6: Modes of operation for the MAHA framework. During compute mode inputs must be provided to the framework over the cycles of the cyclic schedule. Outputs are also available from the framework in different cycles of the same schedule.

array. The address for writing the LUTs is also determined during the process of application mapping and provided as address input to the memory array.

- *Reconfiguring the schedule table:* It is essential to reconfigure the schedule table for all MLBs that may be realized on the MAHA framework. For the MLBs which are used to map the input application, the schedule table

controls the sequence of operations inside the MLB by storing the appropriate control signals as a microcode. For MLBs which are not utilized, schedule table must store control signals which prevent unwanted datapath and memory operations inside the unused MLB. The data required to reconfigure the schedule table is sent into the memory as part of *datain*. Only in this mode, the *schedule table* is written instead of the *function table*.

- *Reconfiguration of the hierarchical interconnect:* Next the programmable switches inserted in the hierarchical routing network are programmed to allow multi-MLB communication. Values for the configuration bits of these programmable switches are determined once routing is completed. The configuration of the hierarchical interconnect is performed in a hierarchical manner starting with the multiplexors at the input of the banks, going down to subarrays. For programming the routing switches we allow the original memory interface to have an additional serial programming port. The width of this serial programming interface is defined by the design parameter *Serial_cfg* and is set to 16 bits. Note that this is much smaller than the data input and output buswidth, which is typically 256 bit each. The serial port bitwidth can be reduced at the cost of additional reconfiguration latency. For serial programming, in our implementation we assume that the configuration bits for the switches at each level of the hierarchy are connected in a serial fashion and loaded using the serial data when PI reconfiguration is enabled for a particular level in the hierarchy. After the programmable switches are reconfigured, multi-MLB connection is established in the MAHA framework. Following the configuration of the routing switches to provide input to the MLBs, switches which select the individual bank outputs to form *dataout* of the memory array are configured in a similar serial fashion.
- *Execution of the mapped application:* The execution of the mapped execution commences following the configuration of the PI. During this mode, input vectors are provided at the *datain* of the memory array. This input

data will be directed towards one or more MLB depending on the mapped application. After execution with the given input vector, primary outputs from one or more MLBs are available at the *dataout* port of the memory. Note that since MBC advocates multi-cycle execution and uses the interconnect in a time-multiplexed manner, for the same input vector set, different signals will require to be inputs to the memory array in different cycles of the cyclic schedule depending on the cycle in which they are used. This is illustrated in Fig. 4.6. Similarly different signals will be outputs from the memory array in different clock cycles.

- *Return to memory mode:* Following execution of multiple input vectors on MAHA, the framework may return to normal memory mode by resetting the programmable switches and simply switching the mode control signal.

In order to support the above modes of operation in the MAHA framework, following data and control signals are essential:

- Read and write enables:* These enable the read and write operation to the normal memory and also write during *function table* configuration.
- cfg_in and cfg_out:* *cfg_in* denotes the serial port to configure the routing switches at the inputs of each bank, subbank, mat and subarray. *cfg_out* indicates the serial port to configure the switches which select the bank output to form *dataout* of the memory.
- mode:* It is the most important control signal which decides one of the 4 modes of operation as described above. The 4 modes are encoded into 2 bits of mode control signal.

5

Software framework

Any hardware reconfigurable framework must be supported by an efficient software architecture which allows the user to map any application with arbitrary complexity to the reconfigurable platform. We have developed a software flow compatible with the working model of the proposed MBC platform for the purpose of application mapping. The software framework developed can be extended to serve different flows of the proposed MBC model, namely for mapping an application to the stand-alone reconfigurable platform, as well as for mapping to the MAHA framework to achieve hardware acceleration.

Memory based computing embodies a hardware reconfigurable architecture with multiple time-multiplexed computing elements where the underlying computing fabric is memory. As a result, it has unique application mapping challenges which have not been addressed by previous CAD algorithms which target spatial hardware reconfigurable frameworks such as FPGA. Key contributions of the software framework developed, common to all the flows are as follows:

- *Description of input application using standard instruction set:* Conventionally instruction sets have been used in describing the functionality of an application to be executed on a software reconfigurable framework such as a general purpose processor (107). In a few cases, they are also used to

specify the functionality of applications executed on an application specific instruction set processor (ASIP) or a reconfigurable instruction set processor (RISP). Only a few attempts have been made to map an input application in terms of a standard instruction set architecture to a hardware reconfigurable framework. Morphosys (9) and Mosaic are such examples (97). However, as stated in (108), the main challenge in computing with FPGAs is the lack of a standard instruction set which serves to abstract the underlying hardware and establishes an agreement between hardware and software. Our software flow attempts to define the operations possible on the proposed MBC framework through a standardized instruction set architecture.

The main idea is to map the input application to the hardware reconfigurable and store the relevant instructions in the form of a micro-code inside the framework during the process of reconfiguration. Morphosys and Mosaic platforms investigated earlier are purely coarse grained reconfigurable frameworks. In this work, we extend the applicability of an instruction set based application mapping to a hardware reconfigurable framework for both fine as well coarse grained datapaths. This is achieved by defining different instruction types for the coarse and fine grained operations in the input CDFG file. Fine grained operations are typically specified by instruction type *complex* (to distinguish it from coarse grained operations) with the subtype specifying the nature of complex operation. During the process of application mapping, the complex operations are scheduled as look-up operations inside a MLB.

- *Application mapping to a time-multiplexed computing fabric with both fine and coarse-grained computing resources:* Previous application mapping methodologies target either a time-multiplexed fine-grained (20, 90) or a coarse-grained computing fabric (109, 110), but not both. This work includes two key contributions, namely, i) decomposition of fine and coarse grained operations and ii) fusing multiple operations of the input application to form multi-input multi-output LUTs. Previous works on time-multiplexed

coarse grained reconfigurable frameworks assume 32-bit datapath elements (98) which allow a single cycle execution for most operations mapped to the reconfigurable frameworks. However, applications which primarily involve bit manipulation (such as advanced encryption standard (AES), variable length coding (VLC), automatic target recognition (ATR) etc.) cannot be efficiently mapped to 32-b custom datapath elements. Further, not all applications require 32-b wide datapath which leads to poor utilization of resources (111). We therefore adopt a smaller datapath width (typically 16-bits) and leverage on a temporal computing model to decompose the same over multiple clock cycles. Our mapping tool incorporates the above decomposition routines for a wide variety of logical and arithmetic operations. Application mapping software for conventional FPGA frameworks partition the input application into multi-input single-output LUTs. Hypergraph partitioning techniques (112) widely used in these algorithms, typically target minimizing the cut-edges between partitions and do not ensure topological order. For MBC frameworks, such a partitioning approach can lead to formation of cycles, which may result in multiple evaluation of the same partition in the MBC framework. However, MBC framework imposes the constraint that the original netlist be partitioned in a topological manner and the partitions are evaluated topologically. In light of the above observation, we have developed a heuristic for partitioning a target application into multi-input multi-output partitions.

Following contributions are specific to the application of memory based computing model in a hardware accelerator framework:

- *Application mapping across multiple MLBs using a time-multiplexed PI model:* Previous works (113) which investigate time-multiplexed PI model for coarse-grained reconfigurable architectures assume that each switch-box in the PI can be scheduled using a cyclic schedule. The primary reason is, for a word (32-b) based coarse-grained datapath, the area and energy overhead for scheduled switch boxes is amortized over multiple bits of a word. However, for a mixed granular reconfigurable framework (with both fine

and coarse-grained computing elements) the area overhead due to scheduled switch boxes would significantly compromise the integration density of the memory fabric. Our application mapping tool therefore assumes a statically scheduled switch box. The connection box is however assumed to be dynamically scheduled, since it is local to each MLB and incurs less area overhead compared to its switch-box counterpart.

- *Placement and routing for a hierarchical interconnect model:* Instrumentation of existing memory architecture brings the unique requirement of placing the individual MLBs in a hierarchical fashion. Knowledge of interconnect bandwidth at each level of hierarchy is used to route the inter-MLB signals. Care is taken so that statically scheduled switch-boxes which connect multiple modules at the same level of hierarchy can be re-used over multiple cycles in a cyclic scheduling policy.
- *Functional validation of the proposed framework:* The application mapping tool is interfaced with a verilog model of the proposed MAHA framework. The tool successfully generates the reconfiguration bits for: i) individual function tables, ii) schedule tables as well as iii) statically scheduled switch boxes. This helps to validate the fact that the MAHA framework can be interchangeably used as a stand alone memory array or converted into a reconfigurable accelerator on demand.

Following contributions are specific to the application of memory based computing model as a stand alone reconfigurable framework:

- *Interfacing with VPR:* As a stand alone reconfigurable framework, multiple MLBs in a MBC framework is expected to behave as a clustered FPGA architecture with programmable interconnects similar to FPGA. VPR (63) is a popular academic software for placement and routing in a FPGA framework. We have therefore interfaced our software flow with the VPR toolset for placement and routing of a multi-MLB netlist.

The rest of this chapter describes the major steps of the software framework developed and possible optimizations in the software framework for improving area, power and performance for the MBC framework.

5.1 Details of the software flow

The application mapping tool is developed in C language and currently stands at 20K lines of code. The tool has been found to successfully map diverse applications of varying complexity with different input constraints. Functionality of the mapped applications have been verified for the MAHA framework by loading the bitstream generated by the tool into the verilog model and performing functional simulation. The tool has also been interfaced with *graphviz* graph visualization software (114) for visually analyzing and debugging the outputs from each intermediate step of the mapping tool. The first step towards mapping the application is to define it in terms of the MBC instruction set architecture.

5.1.1 CDFG format

A text file containing the control data flow graph (CDFG) for the input application serves as the input for the application mapping tool. Table 5.1 lists the operation types currently supported by the tool. Each instruction/operation is currently represented as a vertex in the CDFG and is specified by the following fields: i) name, ii) type, iii) subtype, iv) inputs, v) outputs and vi) bitwidth. ‘*name*’ field should specify the name for each vertex and should be unique. ‘*type*’ specifies the type of operations, as mentioned in Table 5.1. Each *type* of operation may have several *subtypes*. For example, addition and subtraction both fall under the greater type of operations which can be bit-sliced with a carry function (denoted as *bitswC*). ‘*rand*’ subtypes indicates a particular *type* does not have any *subtype*. For memory operations such as *load* and *store*, the subtype represents the bitwidth of the operand that is read out or stored in the memory. Maximum number of inputs as expected for each type of operation is fixed, except for *select*

Table 5.1: Instruction set for MBC framework

Instr Type	Instr Subtype	Inputs	Outputs	Bitwidth	Comment
bitswC	2inadd	a b cin	d e	$x_1 x_2 1$	2-input add bit-sliceable w/ carry
bits	xor	a b	c	$x_1 x_2$	2-input xor bit-sliceable w/o carry
mult	mult	a b	prod	$x_1 x_2$	2-input mult, subtype with rand is ignored
delay	rand	a	a_d	x_1	Eqvt to one cycle delay
shift	left/right	a shftamt	a_s	$x_1 x_2$	Shift operand left/right w/ shftamt
rot	left/right	a rotamt	a_r	x	Rotate operand left/right w/ rotamt
sel	rand	a .. sel	out	$x_0 \dots x_n$	Select one from n operands
complex	rand	a ..	lutout	$x_1 x_2 x_3$	Arbitrary LUT operation
load	x_n	addr	val	x_1	Load val at addr from memory
store	x_n	addr val		$x_1 x_2$	Store val at addr into memory
loadPR	x_n	addr	val	$x_1 x_2$	Load val at addr into page reg.
storePR	x_n	addr val		$x_1 x_2$	Store val at addr into page reg.

type which can represent N to1 selection, where N is variable. The maximum number of outputs for each type of operation is fixed. Bitwidths for each operation indicate the bitwidths of the input operands. Output bitwidth is either implied by the bitwidth of the computing resource or specified in the CDFG when they are inputs to other operations. There is no restriction on the bitwidth, i.e.

the size of each input operand. Additional fields like *selpos* and *enid* are specific to datapath operations such as *select* and memory operations such as *load* and *store* respectively. Following is a brief description of the operation types that are currently supported by the software architecture:

1. *bitswC* represents operations which are bit-sliceable with carry can be bit-sliced into sub-operations with smaller bitwidth and these sub-operations can be evaluated in consecutive clock cycles. For example, a 32-bit add operation can be bit-sliced into 4 8-bit add operations, with the carry-input for each 8-bit add coming from the carry-out of the previous 8-bit addition operation. Apart from addition and subtraction, comparison of two operands is another example operation which can be bit-sliced with a carry output. It has three inputs, two operands and one carry in.
2. *bits* represents operations which are bit-sliceable without carry. Simple logic operations or a combination of them constitute these type of operations. After bit-slicing, the sub-operations can be either scheduled in parallel to each other or in consecutive cycles depending on the resource available.
3. *mult* represents general 2-input multiplication operation. The bitwidth of the product is the sum of the two input operands.
4. *delay* represents one clock cycle delay. It is similar to a delay element in logic design and can be used to create finite state machines. It has a single operand.
5. *shift and rotate* has one operand and one shift/rotate amount. The shift/rotate amount should be atleast $\log_2(op_{bitwidth})$, $op_{bitwidth}$ being the operand bitwidth.
6. *sel* represents an N-input to 1-output selection operation. In addition to the N-input operands, it should have one select signal of bitwidth atleast $\log_2(N)$.
7. *complex* represents the general class of LUT operations. The definition of the LUT operation is specified by the subtype in terms of the input

operands. For the example presented in Table 5.1, the LUT function is specified in terms of the input operands a , b and c . The number of inputs (N) and the bitwidths of each input ($bitwidth_i$) is constrained by the total input size for the LUT. The formula is $\sum_{i=1}^N \lceil bitwidth_i / granularity \rceil \times granularity$.

8. *load*, *store* are memory read and write operations respectively. The first input represents the address for load/store. For the case of store, the second input represents the operand to be stored in the memory. There is no restriction on the size of the address or the data to be stored. The mapping routine decomposes the data to be load/stored to appropriate number of memory locations determined by the size of the address bits.
9. *loadPR*, *storePR* are operations specific to the case where the MLB incorporates a large temporary storage. “*PR*” stands for page-register, a large storage (typically 2KB) which can be used for temporary data storage inside each MLB. *LoadPR* and *storePR* allows read and write to this temporary storage. Currently it is assumed that the bitwidth of the data loaded from or stored into the temporary register is less than the page-register granularity, a parameter which specifies the size of data that can be read or written during a single access to the page-register.

With the CDFG specified in terms of the above instruction set, the proposed application mapping methodology as illustrated in Fig. 5.1 maps the input application to the MBC framework. The essential steps of the software flow are described below.

5.1.2 Decomposition

Decomposition is the process of replacing a vertex with large number of inputs and outputs into multiple vertices with same type and subtype satisfying the input/output constraint for the LUTs as well for the custom datapath in case of mapping to the MAHA framework. Decomposition routines catering to a wide

Figure 5.1: Software flow for the proposed memory based computing model.

range of logic and arithmetic functions as well complex transcendental functions (115) is incorporated in the current version of the tool. Details of the decomposition routines which are currently incorporated in the tool can be obtained in Appendix 10

5.1.3 Fusion

Fusion involves opportunistic reduction in the total number of operations after decomposition (represented as vertices in the CDFG generated after decomposition) by combining multiple operations into a single operation which can be suitably mapped as a LUT or custom datapath operation. It incorporates three routines: i) fusion of random LUT based operations; ii) fusion of bit-sliceable op-

Figure 5.2: Vertices of an application CDFG after scheduling. For the example maximum number of execution levels inside each MLB is taken to be 8.

erations and iii) fusion of custom-datapath operations. Details of these individual routines can be obtained in Appendix 10.

5.1.4 Resource-aware scheduling

Multiple MLBs in the proposed MAHA framework interact following a cyclic schedule. If $Maxlvl_per_MLB$ is the maximum number of consecutive vertex level that can be accommodated in a single MLB, then following a cyclic scheduling policy, a vertex v in the CDFG with a global vertex level of M maps into the level $M \% Maxlvl_per_MLB$ and accommodated inside one of the MLBs. The idea is illustrated in Fig. 5.2 which shows how the vertices are scheduled for the input CDFG shown on the left side. Contrary to purely spatial reconfigurable fabric such as FPGA, the MAHA framework allows the user to easily tradeoff resource usage vs performance by simply specifying the number of custom datapath and memory operations which can be performed in parallel. These are specified by the mapping parameters $Maxcustom$ and $MaxLUTaccess$ respectively. Since the maximum number of parallel custom datapath and memory access operations are

Figure 5.3: Decomposed load and store operations with same enable signals are grouped together in same MLBs.

known beforehand, we employ a simple list-scheduling algorithm (116) to schedule the vertices such that maximum resource usage is not violated. From Fig. 5.2, we see that for $MaxLUTaccess=2$, it is possible to perform a maximum of two LUT access in parallel in one clock cycle. Thus if the MLB hardware supports two LUT access in parallel, all the *xor* sub-operations can be performed inside a single MLB in a single cycle. The number of cycles however increases by 1, if the parameter $MaxLUTaccess=1$.

5.1.5 Packing of partitions to multi-MLB framework

Following the resource aware scheduling of the CDFG vertices, before the actual *packing* of operations into one or more MLBs, intermediate steps known as *groupLoadStore* and *groupLoadStorePR* are executed. These steps are responsible for grouping decomposed load/store and loadPR/storePR operations into respective MLBs. Following criteria are satisfied while grouping these operations:

- After the load and store operations are decomposed into multiple memory read/write operations, these must be grouped together such that load/store operations pertaining to a range of address must be placed in a single MLB. Moreover, load/store operations for different datasets must be distinguished from each other. We use a field *enid* for the load/store operations to indicate the dataset to which the load/store operations operate on. After decomposition, if multiple load/store operations are created to manage a single dataset then unique *enids* are assigned to these pairs of smaller load/store operations such that they address the memory within a given MLB. The idea is illustrated in Fig. 5.3. The same policy is followed for grouping *loadPR* and *storePR* operations.

Following the packing of memory access operations such as loads and stores into one or more MLBs, the packing algorithm ensures that the vertices of other types are packed into the existing or additional MLBs without violating the hardware constraint of individual MLBs. The hardware resource constraints that must be satisfied during the *packing* step are:

- *CheckMem*: Total memory requirement for all the LUT operations and the load/store operations packed inside a MLB must not surpass the total memory available inside each MLB.
- *CheckIO*: The number of signals (considering the bitwidth of the signals) which are inputs and outputs to a MLB in a given clock cycle must be less than the number of the inputs and outputs available to each MLB.
- *CheckBank*: The number of parallel LUT operations inside a MLB in a given clock cycle must not exceed the maximum number of parallel memory accesses allowed in a MLB in a given clock cycle.
- *CheckInterim*: The temporary and the dummy registers present inside a MLB must be sufficient enough to hold all the feedforward signals for the vertices that are packed in a given MLB.

In addition, for specific MBC flows such as MAHA, the packing algorithm also ensures:

- *CheckCustom*: The number of custom datapath operations at each level of operation does not exceed the maximum number of custom datapath operations that can be simultaneously executed by the custom datapath. For example, operations $(a + b) \ll \text{shiftamt}$ and $c \ll \text{shiftamt}$ cannot be simultaneously executed within the datapath for a single MLB.
- *CheckPR*: For MBC framework with page register, it ensures that the total size of the temporary variables that are accessed via *loadPR* and *storePR* operations to be packed inside a MLB do not exceed the page-register size.

Pseudocode for the proposed packing heuristic is presented in Algorithm 1. A DAG $G'(P, E)$ with partitions (P) obtained from the partitioning step serves as the input to the packing algorithm. PL_i denotes the partition list for a MLB CL_i . The algorithm proceeds by topological sorting of P followed by processing of the partitions in the descending order of their levels. As illustrated in Algorithm 1, the temporary node list $temp_j$ must satisfy the constraints mentioned above before a partition p_i can be included in PL_j . We briefly describe the methodology for calculating the memory requirement for $temp_j$ in Algorithm 2. Here, $I(p_i)$ and $O(p_i)$ denote the number of inputs and outputs for each partition; fb denotes the number of fine blocks and cb denotes the number of coarse blocks required to map the given partition.

An example of packing operations into multiple MLBs is shown in Fig. 5.4. An important step of the packing heuristic is to determine the number of LUT and load/store operations that would fit inside the memory for a single MLB. Ideally a LUT or a dataset can begin or end at any address within a given slice of the function table. However, in order to access other LUTs, we will require adding an offset to the memory access address. The idea is illustrated in Fig. 4.1 which achieves the highest efficiency in packing LUTs/datasets within the function table. This however inordinately increases the memory address calculation time. A parameter $Utblky$ is therefore introduced to define the memory

Algorithm 1 *Packing*

```

1: Input: DAG representation  $(G'(P, E))$  of the partition set  $P$ , Design Parameters (refer to Fig. 6.1)
2: Output: Set of MLBs  $(CL)$  with the mapped partitions
3: Levelize the partitions  $p_i \in P$ 
4: Sort partitions topologically and store them in  $P_{sort}$ 
5: for  $i = 0$  to  $|P_{sort}| - 1$  do
6:   if  $p_i \in P_{sort}$  is unmarked then
7:     for  $j = 0$  to  $|CL| - 1$  do
8:        $temp_j = PL_j \cup p_i$ 
9:       if  $CheckMem(MFFS(temp_j)) == True$  then
10:        if  $CheckIO(MFFS(temp_j)) == True$  then
11:          if  $(CheckBank(MFFS(temp_j)) == True)$  then
12:            if  $CheckInterim(MFFS(temp_j)) == True$  then
13:               $PL_j = PL_j \cup p_i$ 
14:              Mark  $p_j | p_j \in temp_j$ 
15:            end if
16:          end if
17:        end if
18:      end if
19:    end for
20:    if  $p_i$  is unmarked then
21:      Create a new MLB  $m \ni PL(m) = MFFC(p_i)$ 
22:       $CL = CL \cup m$ 
23:      Mark  $p_j | p_j \in MFFC(p_i)$ 
24:    end if
25:  end if
26: end for

```

access granularity along the rows. LUTs or datasets mapped to the *function table* can only start from the row addresses at intervals of $Utblky$. $Utblky$ and $Utblkx$ together define the unit block for the function table. A LUT or a dataset cannot occupy a space smaller than that defined by this unit block. While checking for the space occupied by a set of vertices to be mapped to the memory, the number of unit blocks occupied by each is first calculated, followed by the sum of the

Figure 5.4: Decomposed and fused vertices of a CDFG are packed into multiple MLBs.

Algorithm 2 *Memory Calculation*

- 1: **Input:** List of vertices $PL = \bigcup p_i$
 - 2: **Output:** Check for memory violation
 - 3: **for** $i = 0$ to $|PL| - 1$ **do**
 - 4: $mem_i = 2^{I(p_i)} \times O(p_i)$
 - 5: $fb = \lceil mem_i / (Utblkx \times Utblky) \rceil$
 - 6: $cb = cb + fb$
 - 7: **end for**
 - 8: **if** $cb > Row_subarray \times Col_subarray$ **then**
 - 9: PL violates memory requirement
 - 10: **end if**
-

individual requirements. If the total number of unit blocks is less than the maximum number that can be fitted into the function table, then the vertices under consideration are packed into a single MLB. Algorithm 2 explains the steps of memory requirement calculation.

5.1.6 Placement

Following the packing of the operations into multiple MLBs, the MLBs must be placed and routed to estimate the power and performance of the input application mapped to the MBC framework. The software flow branches here depending whether the application is being mapped to a stand-alone reconfigurable

framework or to the MAHA framework. The placement procedures for these two branches are described below:

5.1.6.1 Placement for stand-alone reconfigurable computing

A stand-alone reconfigurable computing framework should be able to map applications from diverse domains and therefore comprises of the generic LUTs and fine-grained elaborate programmable interconnect. In modern FPGAs, these LUTs are clustered together in a group of 8-10 (46, 53) with both intra-cluster as well as inter-cluster communication. MBC also mimics a clustered FPGA architecture, where the individual LUTs are mapped to the function table. We have therefore used the popular VPR (63) toolset for placing and routing a multi-MLB netlist. Our toolflow is interfaced with the VPR toolsuite through the *.net* file format compatible with VPR which is generated from our toolflow. The file generated from our software flow is called *output.net*. For the FPGA framework *.net* file typically contains the following:

- Description of the inputs and outputs to the LUTs packed into multiple clusters in the FPGA. Clocks and global signals are also distinguished.
- Communication across LUTs inside a cluster.
- Communication across LUTs from one cluster to another.

The *.net* file generated from the MBC flow contains the following information:

- Description of the inputs and outputs of the LUTs packed inside the function table of the different MLBs.
- Intra-MLB communication.
- Inter-MLB communication.

The caveat here is however, VPR does not understand multi-cycle or time-multiplexed operation specific to the MBC framework. To place and route the multi-MLB netlist through VPR, we therefore unroll all the operations in time

and provide an abstraction for time-multiplexed interconnect. This is achieved in the following manner:

- *Modifications to the FPGA Architecture Description File:* Placement and routing of clustered FPGA architecture using the VPR toolsuite requires a FPGA architecture description (117). This description contains the following important information: i) position of input and output pins on each cluster; ii) maximum number of LUTs per cluster; iii) maximum input size for each LUT; iv) connection block flexibility (F_c) and switch block flexibility (F_s); v) resistance and capacitance for the programmable interconnect for inter-cluster communication vi) sequential and combinational delays inside each cluster and vii) area of each cluster. In order to adapt the FPGA architecture description to that for the MBC framework following considerations must be made: i) the position of the input/output pins on the cluster description as well as the values of F_c and F_s are kept the same as in the architecture description file; ii) since MBC supports variable size LUTs, the maximum number of LUTs that can be mapped to the function table of a MLB is quite large depending on the size of the function table. The maximum number of LUTs inside a MLB is therefore computed based on the minimum LUT size and is provided in the MLB architecture description file; iii) maximum LUT input size is also modified according to the MLB specifications; iv) area of cluster is replaced by the area for a a MLB; v) in comparison to the FPGA architecture, area and delay values for the programmable interconnect are re-calculated to account for the increased area of each MLB tile and hence the increase route length.
- *Modeling the time-multiplexed programmable interconnect:* Let $Input_{i,k}$ and $Ch_{i,j,k}$ denote the number of inputs to MLB_i and the outputs from MLB_i to MLB_j in clock cycle k . In order to model a time-multiplexed programmable interconnect, the maximum values of $Input_{i,k}$ and $Ch_{i,j,k}$, $\forall k$ is computed and assigned to the respective MLBs in *output.net*. This ensures that only $Max\{Ch_{i,j,k}|\forall k\}$ routing channels are used to route the signals from MLB_i to MLB_j . Similarly the inputs for all the LUTs mapped to a MLB

Figure 5.5: a) Important steps of the simultaneous scheduling-placement-routing heuristic; b) MLBs after placement. For the example, the number of modules at each level are 4(Banks), 1(Subbanks), 1(Mats) and 4(Subarrays) respectively.

must come from the output of other LUTs mapped to the same MLB or $Max\{Input_{i,k}|\forall k\}$ MLB inputs. Note that through packing it is ensured $MLB_Input \geq Max\{Input_{i,k}|\forall k\} \geq Max\{Ch_{j,i,k}|\forall k\}$.

The MLB architecture description file and the multi-MLB netlist *output.net* serve as inputs to the VPR routing tool. Placement and routing using VPR is then performed with no constraint on the maximum number of routing channels. The VPR tool is not currently called from inside the software flow. The user needs to run it separately with the architecture description and the *output.net* file generated from our flow.

5.1.6.2 Placement for MAHA framework using a hierarchical interconnect

As described earlier, existing on-chip/off-chip memory can be instrumented to create a reconfigurable framework on demand. In this MAHA framework, sub-arrays are instrumented to create MLBs and the large bandwidth that already exists inside the memory is used for inter-MLB communication. The interconnection network that exists inside the memory is however hierarchical, resembling

a H-tree (102). A hierarchical placement and routing algorithm was therefore proposed which satisfies the requirement specific to the MAHA framework.

Given the number of modules in each level of the memory hierarchy and their I/O bandwidth, the mapper places the MLBs in a hierarchical fashion such that the number of inputs and outputs crossing each module is minimized. To realize the same, we have implemented a bi-partitioning approach based on Fiduccia-Matheyses algorithm where MLBs are first partitioned among the first level modules. This is followed by distribution among the second level modules and is continued till each MLB has been mapped to the lowermost memory module (a block). The cost function for the bi-partitioning approach is taken to be $\sum_{i=1}^N \sum_{j=1}^N (-1)^k \frac{b_{i,j}}{l_{i,j}}$, where N denotes the total number of MLBs, $b_{i,j}$ denotes the number of edges between MLB_i and MLB_j and $l_{i,j}$ denote the difference in the level between the source and sink operations inside the two MLBs. $k = 1$ when both MLB_i and MLB_j are inside the same partition of a bi-partitioning step and $k = 0$ when they are in two different partitions. Effectively, two MLBs which communicate infrequently or if the communication is between vertex levels spaced wide apart are placed into different modules.

Routing of primary inputs and outputs to the DFG and the inter-MLB signals is performed in the following hierarchical order:

- Routing of primary inputs to each MLB for all levels of the cyclic schedule.
- Routing of signals which cross the first level of memory hierarchy for all levels of the cyclic schedule.
- Routing of signals which cross only the second level of memory hierarchy for all levels of the cyclic schedule. This procedure is repeated for each level of the memory hierarchy.
- Routing of primary outputs from each MLB for all levels of the cyclic schedule.

The order simplifies the static switch-box allocation across multiple clock cycles. A signal is routed from the primary input to a memory block on $Channel_i$, only

if corresponding channels at upper levels of memory hierarchy is free as well. A channel is considered to be available for routing only if: i) No other signal is scheduled on the channel in the same clock cycle. ii) If the switch box which drives the channel is set in a manner such that the source and destination modules are same for the given signal. An already set switchbox indicates it is used to route a signal with the same source and destination for another cycle of the cyclic schedule.

Fig. 5.5(a) summarizes the essential routines of the placement and routing step. In order to address the scenario where the placement or routing fails due to insufficient routing resources, we have implemented a simultaneous placement-routing-scheduling routine which re-schedules vertices so as to minimize the routing congestion for a given execution level. During the process of re-scheduling, the execution level with maximum congestion is first identified. Then vertices with most inputs outside the MLB are iteratively moved to the next execution level. The dependant vertices are also shifted from their respective levels. This is continued until the routing succeeds or the vertex returns to the original level in the cyclic schedule. Fig. 5.5(b) shows the placement of the MLBs shown in Fig. 5.4 for a representative hierarchy of banks, subbanks, mats and subarrays. ‘NULL’ denotes empty memory blocks.

5.1.7 Report generation

Following the successful placement and routing (P&R) step, reports are generated for estimating the area, power and performance of the mapped design. Similar to P&R, the report generation is also distinct for the two flows.

5.1.7.1 Estimation of energy and performance for MBC based stand-alone reconfigurable computing

Following are the important steps to the report generation for MBC based stand-alone reconfigurable framework:

- Area Estimation:* The area of the MLB can be approximated as the sum of the areas of the individual MLB components. The formula is given as:

$$Area_{MLB} = Area_{controller} + Area_{SchedTable} + Area_{Regfile} + Area_{Dummyreg} + Area_{Muxtree} + Area_{FunctionTable}.$$
- Delay Estimation:* As mentioned earlier, the delay for an application mapped to the MBC framework is a function of both intra-MLB and inter-MLB delays and is calculated as: $Delay_{total} = Max(Delay_{inter-MLB}, Delay_{poscyc}) + Delay_{negcyc}$, where $Delay_{inter-MLB}$ represents the worst case inter-MLB delay after the application is mapped to the MBC framework. $Delay_{poscyc}$ and $Delay_{negcyc}$ represent the positive and negative half cycle intra-MLB delays. $Delay_{poscyc}$ consists of temporary register and dummy register read time and formation of effective addresses to access the function table. This delay is accurately calculated from SPICE simulation. The negative half cycle delay is dominated by the function table access time. Since the function table is modeled as a standard 2-D dense memory array, we have used CACTI (102) in estimating the area, energy and power for different sizes of function table. The worst case inter-MLB delay is obtained from the critical path reported by VPR after placing and routing the multi-MLB netlist. A script was written to parse the MLB to MLB delay in the critical path report and flag the worst case inter-MLB delay. With these three parameters, the total time period for the application mapped is ascertained.
- Energy Estimation:* For MBC based stand alone reconfigurable framework, currently we only report the MLB energy (dynamic + leakage), i.e. the total energy expended in the MLBs per input vector and do not report the routing energy. The total energy per vector is calculated as:

$$Energy_{total} = \sum_{i=1}^N \sum_{j=1}^M (Energy_{controller} + Energy_{Schedtab} + Energy_{Regfile} + Energy_{Dummyreg} + Energy_{Muxtree}) + \sum_{i=1}^K Energy_{FunctionTable}.$$

As evident from the above expression, the total energy per input vector is a function of the total number of MLBs (M) as well as the total number of cycles (N) required to execute a single vector of the mapped application. Energy is

expended in the controller, schedule table, register file, dummy register and muxtree in all the MLBs in all clock cycles N . However, the energy expended in the function table is a function of the total number of LUT operations (K) actually required by the mapped application to evaluate a single input vector. In our framework, the logic energy is calculated through SPICE simulations while the memory energy is obtained from CACTI. Currently the routing energy is not estimated since that would require propagating the logic values through the MLBs. Since we currently do not support functional simulation of the multi-MLB framework for stand-alone reconfigurable computing, it is not possible to propagate the logic values at MLB outputs. This will be an extension to the future version of the tool.

5.1.7.2 Estimation of design overhead for the MAHA framework

The area, performance and energy estimates for the MAHA framework are arrived at in a manner similar to the stand-alone reconfigurable framework. Considering the custom datapath present inside each MLB, the area, performance and MLB energy for the mapped application is modified in the following manner.

- The area for the custom datapath is now added to the total MLB area. The schedule table area is also increased in order to store the microcode relevant to custom datapath operation. The number of temporary registers is also increased to store the outputs from the custom datapath. Furthermore, the muxtree also increases in size to select the operands for the custom datapath.
- Since the custom datapath operations occur in the negative half cycle in parallel to the function table access, the delay for the custom datapath operations is unlikely to affect the performance in case of MAHA framework.
- The total energy is now augmented by the energy of the operations which are mapped to the custom datapath. The schedule table access energy and the muxtree energy also increases with the size of the custom datapath width.

In addition to single MLB area, multi-MLB performance and total MLB energy, our software flow also reports the following information for the application mapped:

1. Number of vertices in the original CDFG, vertices after partitioning and fusion.
2. Break down of the vertex types.
3. Size of reconfiguration data that is required to on-demand configure the MAHA framework from a simple storage system to the computing platform. This is calculated as $\sum_{i=1}^M (Reconfig_{FunctionTable_i}, Reconfig_{SchedTab}) + Reconfig_{PI}$, where $Reconfig_{FunctionTable_i}$ denotes the data required to configure the function table for MLB_i . $Reconfig_{SchedTab}$ denotes the size of the schedule table reconfiguration data and depends only on the schedule table size, irrespective of the MLB number. $Reconfig_{PI}$ denotes the reconfiguration data for the programmable interconnect in MAHA framework.
4. The number of routing switches used at each level of hierarchy in the hierarchical PI framework is also reported in case of MAHA.
5. It reports the total interconnect area on the basis of the number of the total number of elements at each level of hierarchy (banks, subbanks, mats or subarrays) for a hierarchical interconnect.
6. Based on the number of routing switches used at each level of hierarchy, it estimates the total routing energy for each input vector as $\sum_{i=1}^L (SwitchCount_i \times Switchenergy_i)$, where L denotes the total number of levels in the hierarchical interconnect, $Switchcount_i$ denotes the number of switch used in level i of the hierarchy and $Switchenergy_i$ denotes the energy per switch for switch at level i .

Report generated by the MAHA software flow is written in *report.dat*.

5.1.8 Bitfile generation and functional validation

The bitfile generation routine is specific to the MAHA software flow. It accepts the placed and routed netlist and performs the following important functions:

- It generates the control or select bits for the programmable switches used to route the signals at different levels of the hierarchical interconnect.
- Since the connection-box both at input and output of a MLB is dynamically scheduled, it generates the schedule table entries which select appropriate inputs from input channels in a given clock cycle. Similarly it generates the microcode to steer the outputs from the function table and the custom datapath to proper output channels.
- In addition to input and output control signals, it also generates schedule table entries which correspond to: i) enables for the custom datapath and memory operations; ii) register enables for writing into the register file and iii) starting location of LUTs being accessed in each clock cycle.
- In addition, it performs the important task of liveness analysis for intermediate variables and register allocation. If a signal is *live* over multiple clock cycles, it moves the signal from the temporary register to an appropriate location in the dummy register so that its value is not overwritten in the following clock cycle.

The bitfile generated by the tool for a given application can be directly loaded into the MAHA verilog model. Functionality in the MAHA mode is then validated by specifying the input vectors for the mapped application.

5.1.9 Design space exploration

The tool allows the user to perform design space exploration of the MBC framework. The user may vary a number of MBC architectural parameters and note their impact on the area, performance and energy for both MAHA as well as the stand-alone reconfigurable framework. These parameters are:

- i. Number of elements as well as the interconnect bandwidth at different levels of the hierarchy for the hierarchical interconnect of the MAHA framework.
- ii. Width of the custom datapath inside each MLB.
- iii. Number of parallel LUT access inside each MLB.
- iv. Maximum number of consecutive levels of execution inside a single MLB.
- v. Maximum number of parallel custom datapath (*Maxcustom*) and LUT operations (*MaxLUaccess*) allowed for the mapped application.

In order to perform design space exploration, the user is required to:

- Run the tool with *-e* option.
- Provide an explore file which contains the area, performance and energy values for different MBC components for each design space configuration. For example, if the maximum number of parallel LUT accesses inside a MLB varies as 1,2 and 4 and the maximum number of consecutive levels of execution varies as 2, 4 and 8, then the explore file must contain the area, performance and energy values for the MLB components corresponding to the configurations 1x2, 1x4, 1x8, 2x2, 2x4, 2x8 and 4x2, 4x4 and 4x8 respectively.

At the end of design space exploration, area, performance and energy for the mapped application corresponding to different design points can be found in the file *explorer.es.txt*. The user may then select the best configuration depending on the design constraint.

The software framework described above serves several important purposes. These are summarized below:

- i. Given a MLB architecture specification, it maps the CDFG for the input application to a multi-MLB netlist.
- ii. It places and routes the multi-MLB netlist.

- iii. Given the area, performance and energy estimates for the MLB and the PI, it calculates the total area, energy and performance for the mapped application.
- iv. It allows the user to vary the MLB architectural parameters and arrive at an optimal MLB configuration depending on whether the design constraint is area, performance or energy.

6

MBC design space exploration

MBC design space exploration means finding the appropriate values of the MBC design parameters such that it improves one or more design constraints such as area, performance and energy. Since MBC is a hardware reconfigurable framework which advocates spatio-temporal execution of the input application mapped across multiple MLBs, the design space to explore primarily consists of two components: i) compute block configuration and ii) organization of the programmable interconnect. Till now, we have primarily focussed our attention towards the design of individual MLBs, the main reason being it is a temporal computing block which is still hardware reconfigurable. This is unlike *clusters* in modern FPGAs which are fully spatial and can accommodate a small number of LUTs. As discussed before, multi-cycle computing inside a single MLB reduces the multi-MLB communication requirement significantly. It is due to this reason innovations in the MBC routing fabric has been left for future investigations. Currently, for MBC based stand alone reconfigurable computing, we rely on an elaborate programmable interconnect similar to a FPGA while for the MAHA framework, we assume the inter-MLB communication is achieved through a hierarchical interconnect network such as *H-tree*. In particular, this chapter makes the following contributions:

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

- It explores different values for the MLB design parameters and investigates the impact of each on the area, energy and performance for individual MLBs as well for different applications mapped to multiple MLBs. Estimates for area, energy and performance are obtained from detailed SPICE simulations at scaled technology nodes.
- It explores implementation approaches for different logic components of the MLB and compares the design overheads across these approaches.
- Optimizing a memory design for read and write presents contradictory requirements. The memory array for MBC based stand alone reconfigurable computing is dominated by read. Hence, we present novel memory designs where the memory cells are optimized for improved read performance and energy at the cost of write which is infrequent.
- It compares the energy efficiency of the MBC framework against a baseline FPGA framework.
- Lastly, we propose a programmable logic array (PLA) based application mapping technique which uses content-addressable memory (CAM) to hold the PLAs for the partitions into which the application is decomposed. Compared to a LUT based application mapping technique, such a mapping reduces the area requirement exponentially and leads to considerable improvement in performance and energy for the MBC framework.

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

This section explores the design space for MBC based stand-alone reconfigurable computing framework used to map random logic. Important design parameters were identified and their effect on the geometric mean of performance and energy across benchmarks were noted.

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

MLB Parameters		
Granularity	Register Width	Register Count
Dummy Reg. Count	# of Memory Slices	Max. Level per MLB
Custom Datapath Size	Ut. Memory block height	Ut. Memory block width
# of MLB Inputs		# of MLB Outputs
Scheduling Parameters		
Max parallel datapath operations		Max parallel LUT operations
Additional MAHA Parameters		
Total Memory Size (B)	Memory Line Size (B)	Memory Bus Width
# of levels of memory hierarchy		# of blks in each level
Interconnect bandwidth at each level		Interconnect selection granularity

Figure 6.1: MLB design parameters for MBC based stand alone reconfigurable computing and additional design parameters for MAHA. Scheduling parameters allows the designer to trade-off between performance and resource usage.

Figure 6.2: a) Breakdown of MLB tile area and comparison with FPGA tile area.
b) Estimating the increase in switch area and delay due to higher MLB tile area.

6.1.1 Benchmarks and simulation setup

MCNC and ISCAS'85 and ISCAS'89 benchmarks (118) were selected for validating the effectiveness of the MBC based stand-alone reconfigurable computing framework. These benchmarks are mostly composed of random logic and offer

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

different degrees of routing complexity to the mapping tool. These benchmarks were mapped to the MBC framework following the software flow described in the previous chapters. In addition the FPGA architecture description file was modified to allow the VPR toolsuite to place and route the ‘*output.net*’ file that is generated from our software flow. Due to increase in the MLB tile area, it is necessary to accurately estimate the route delay in case of a MBC based reconfigurable platform. However, for accurate estimation of such delay, parameters such as i) Tile area; ii) Switch Area and iii) Switch Delay need to be determined for the MBC framework. Fig. 6.2 shows the breakdown of the MBC tile area and compares the MLB area against the tile area for a baseline 10-cluster 7-LUT FPGA. The MLB tile area was estimated for different combinations of the MLB design parameters (refer to Fig. 6.1), of which only 6 are depicted in Fig. 6.2(a). Increase in the MLB tile area demands resizing of the multiplexors and the buffers in unidirectional wires. Sizing was performed for representative resistance and capacitance values for 65nm interconnect model, considering different tile areas for different MLB configurations (Fig. 6.2(b)). From Fig. 6.2(a) and (b) we note that a MLB configuration with $Max_LUT_input=12$, $Utblkx=4$, $Log_num_slices=3$, $Row_subarray=64$ and $Col_subarray=32$ leads to 21 times increase in the tile area along with 10 times increase in switch area and 50% increase in switch delay. Another configuration with $Max_LUT_input=12$, $Utblkx=2$, $Log_num_slices=3$, $Row_subarray=32$ and $Col_subarray=32$ however, incurs only 8 times increase in tile area and 28% increase in routing delay for the same switch area compared to the baseline FPGA model. Note that for nanoscale technologies, it is beneficial to employ LUTs of larger input size. It is in this context that we wanted to demonstrate the benefit of mapping applications to larger LUTs and subsequently mapping these LUTs to dense 2-D memory arrays. For the spatial FPGA model, the framework with largest LUT for which models are freely available is for a 7-input LUT. In order to compare between the spatio-temporal MBC model and the fully spatial FPGA at scaled technologies, we have therefore used 12-input 4-output LUTs in case of MBC and 7-input 1-output LUTs in case of FPGAs. Please note that, compared to a 6-input LUT based FPGA, the improvement

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

Table 6.1: MBC design space exploration - I

Const: Utblkx=4, Slices=8, Num_DI_subarray=32 Dummy_reg_count=24, MEM=2KB				
Var.	Dependent Param.			
# of Partition ips	# of Partitions	Mem Req. (KB)	# of Logic Levels	Max Route Delay (ns)
8	227.76	11.1	4.717	0.3
10	166.67	28.93	3.68	0.34
12	124.12	79.82	3.25	0.28

Table 6.2: MBC design space exploration - II

Const: Max_LUT_input=12, Slices=8, Num_DI_subarray=32 Dummy_reg_count=24, MEM=2KB			
Var.	Dependent Param.		
# of Partition ops	# of Partitions	Mem Req. (KB)	Max Route Delay (ns)
1	287.46	40.36	0.37
2	166.9	66.81	0.28
4	124.12	79.82	0.29

with the MBC framework would be more significant.

6.1.2 MLB architecture exploration

6.1.2.1 Choice of number of partition inputs and outputs

Following trends were observed (refer to Table 6.1) on varying the partition input size for standard benchmark circuits: i) As the number of partition inputs is increased maximum number of logic levels in the benchmark circuit is reduced. Total number of partitions also reduces which leads to decrease in the inter-MLB route delay. ii) On the other hand, total memory requirement increases nearly exponentially. Going from a 8-input partition to a 12-input partition,

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

Table 6.3: MBC design space exploration - III

Const: Max_LUT_input=12, Utblkx=4 Num_DI.subarray=32, Dummy_reg_count=24, MEM=2KB			Const: Max_LUT_input=12, Utblkx=4 Slices=8, MEM=2KB Dummy_reg_count=24		
Var.	Dep Param.		Var.	Dep Param.	
# of Mem Banks	# of MLBs	Max Route Delay (ns)	# of MLB Input	# of MLBs	Max Route Delay (ns)
4	24.15	0.34	24	24.15	0.32
6	19.06	0.29	32	17.18	0.28
8	17.18	0.28	40	14.74	0.35
10	16.93	0.33	48	13.2	0.36

Table 6.4: MBC design space exploration - IV

Const: Max_LUT_input=12, Utblkx=4, Num_DI.input=32, Slices=8, MEM=2KB			Const: Max_LUT_input=12, Utblkx=4 Num_DI.input=32, Slices=8 Dummy_reg_count=24		
Var.	Dep Param.		Var.	Dep Param.	
# of Dummy Reg	# of MLBs	Max Route Delay (ns)	Mem per Bank	# of MLBs	Max Route Delay (ns)
8	18.59	0.31	1KB	18.53	0.32
16	17.93	0.28	2KB	17.18	0.28
24	17.18	0.28	4KB	17.18	0.28
32	16.88	0.29	8KB	17.18	0.28

one would expect a $2^4 = 16$ times increase in the total memory requirement. However, this expected increase is offset by the fact that: i) there is a linear reduction in the total number of partitions, ii) not all the partitions have 12 inputs. Our simulation results suggest that the combined effect of these two cause only a $8.76X$ increase in the memory requirement for standard benchmark circuits when partition input size is increased from 8 to 12 (refer to Table 6.1).

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

Memory requirement increases as 1.21 times for $\sim 2X$ increase in the number of partition outputs. From Table 6.2 we note that across all the benchmarks, the minimum routing delay is achieved for $Utblkx = 2$. The final MLB architecture however supports partitions with output count ($Utblkx$) up to 4 bits wide to minimize the number partitions evaluated and hence improve the energy.

6.1.2.2 Choice of number of slices in function table

Number of slices in the function table (denoted by Log_num_slices) dictate the maximum number of partitions that can be evaluated in parallel for a given cycle. With an increase in the value of Log_num_slices , the total number of MLBs required to map a given design is reduced. The minimum routing delay is however achieved for $Slices = 8$ (refer to Table 6.3). Thus in the final MLB architecture the function table is organized into 8 slices which supports up to 8 parallel partitions in a single execution cycle.

6.1.2.3 Choice of number of MLB inputs and outputs

Choice of MLB input is guided by the objective of maximizing the number of parallel partitions in the MLBs. From Table 6.3, we note that for small values of $Num_DI_subarray$, the number of MLBs required is more and therefore increases the routing delay. As the number of MLB input is increased, number of MLBs reduces drastically, indicating the increase in the number of parallel partitions in each MLB. If $Num_DI_subarray$ is excessively increased, the number of signals that needs to be routed to each MLB is also increased, thus increasing the maximum routing delay for a set of benchmarks. From Table 6.3, we note that the routing delay is minimum for $Num_DI_subarray = 32$. As in the case of a clustered FPGA architecture, number of outputs from each MLB in a given cycle is given by ‘Maximum number of partitions that can be evaluated in parallel $\times$ Maximum number of outputs from each partition’ For, $Slices = 8$ and $Utblkx = 4$, we have $Num_DO_subarray = 32$.

6.1.2.4 Choice of the number of dummy registers

Since the partitions are evaluated in a leveled manner inside each MLB, it is necessary to use dummy registers to accommodate for signals that cross more than one level of execution. We noted that as the number of dummy registers (*Dummy_reg_count*) is gradually increased, the total number of MLBs is reduced gradually. For *Dummy_reg_count* greater than 24, the reduction is sufficiently small (refer to Table 6.4). The final MLB architecture therefore supports a maximum of 24 feedforward signals in each cycle.

6.1.2.5 Choice of memory per slice of the function table

With increase in memory size, memory access time increases as $\propto \log_{10}$ (102, 119) for memories less than 64KB. Higher memory storage for the function table at each MLB would therefore only lead a negligible increase in the MLB cycle time. Table 6.4 shows the effect of varying the memory per slice (denoted as *MEM*) of the function table. From our simulations, we observe that only when *MEM* < 2KB, there is a 7.2% increase in the total number of MLBs required to map the benchmark circuits. As *MEM* is increased beyond 2KB, no improvement in MLB count or route delay is observed for *Utblkx* = 4. For *Utblkx* = 2, improvement in routing delay ceases after *MEM* = 1KB. The final value of *MEM* is therefore restricted to only 1KB.

From the trade-off analysis provided in Table 6.1 - Table 6.4, we select the design parameters for the final MBC architecture as: i) *Max_LUT_input*=12, ii) *Utblkx*=2, iii) *Num_slices*=8, iv) *MEM*=1KB, v) *Dummy_reg_count*=24, vi) *Num_DI/DO_subarray*=32.

6.1.3 Simulation results

Performance for the proposed MBC framework was validated against the baseline 7-input LUT and 10-LUT clustered FPGA model with 39 inputs. FPGA architectures for 65nm Predictive Technology Model (PTM) were obtained from (117).

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

For multi-MLB interconnect framework, we have assumed F_s (switch-block flexibility) and F_c (connection-block flexibility) to have the same values as in our baseline clustered FPGA model. The maximum routing delay for our mapped netlist determines the inter-MLB cycle time (T_{cycle}). The total time for evaluating a given benchmark is calculated as $T_{total} = (\# \text{ of Logic Levels}) \times T_{cycle}$. Total power expenditure for the proposed framework is estimated as $P_{total} = (\# \text{ of Partitions}) \times P_{partition}$, where $P_{partition}$ denotes the power required to evaluate a single partition. Since the MBC framework evaluates a design over multiple cycles, the total time for evaluating the mapped design is $\# \text{ of Cycles} \times CycleTime$. We have also compared the Energy Delay Product (EDP) for FPGA against that for a MBC framework at supply voltages 1.1 and 0.8V. It is to be noted here that while delay comparison with FPGA considers both logic and routing delay in the MBC framework, energy comparison considers only the logic energy in the two frameworks. We could not obtain the energy numbers for MLB interconnection since activity generation for MBC framework and interfacing that with power-aware VPR (120) is not yet supported. However, we expect significant savings in routing energy due to the considerable reduction in the total number of nets compared to a FPGA framework.

6.1.3.1 Delay and power results

6.3 shows the break up of the MLB cycle time and the power required to evaluate a single partition. From Fig. 6.3, we note that the MLB cycle times at 65nm technology node with 1.1V supply voltage is $680ps$. This includes the placement and routing delay overhead. From Fig. 6.3, we observe that the delay through the intra-MLB routing architecture dominates the cycle time followed by the total memory access delay. The total power required to evaluate a single partition is calculated to be $1.38mW$. This includes power from reading and decoding the schedule table outputs. For estimating the EDP improvement at a lower supply voltage, we have also estimated the cycle time and power requirement at 0.8V. From Fig. 6.3, we note that the MLB cycle time at 0.8V increases to $1132ps$, while the total power requirement for a single partition reduces to $0.89mW$. In

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

Figure 6.3: Breakdown of a) delay and b) energy for a single MLB at 65nm technology node. c) alu4 benchmark circuit mapped to c) FPGA and d) MBC frameworks respectively.

order to investigate the technological scalability of the proposed framework, MLB cycle times and power requirements were also estimated using PTM 45nm (45) technology model file. From Fig. 6.3, we note that at 45nm, MLB cycle time improves to $560ps$ and the power requirement reduces to $0.78mW$. It is important to note here that the delay for the internal routing architecture scales at a higher rate (30.5%) from 65nm to 45nm compared to the routing framework used to connect the MLBs or the CLBs as in the case of FPGA.

6.1.3.2 Design overhead for PTM 65nm

Table 6.5 compares the delay, energy and EDP values for mapping the benchmark circuits to the MBC platform as well as to the FPGA framework for two different voltage points. After mapping the benchmark circuits to 12-input LUTs, benchmarks such as *apex4*, *ex5p* and *bigkey* can be evaluated in a single cycle and there is no overhead for inter-MLB communication. From Table 6.5, we note that the average improvement in delay for all the benchmarks is 56.41% . From Table 6.5,

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

Table 6.5: Comparison of MBC and FPGA frameworks for PTM 65nm

Ckts	% Improv. in # of LUT	% Improv. in # of Clusters	% Improv. in Delay		% Incr. in Energy		% Improv. in EDP	
			@ 1.1V	@ 0.8V	@ 1.1V	@ 0.8V	@ 1.1V	@ 0.8V
c3540	71.62	71.05	44.38	18.44	232.71	55.88	-85.05	-27.13
c5315	75.89	60.71	14.93	-24.74	-	-	-	-
c6288	86.44	86.30	56.07	41.5	72.84	-19.02	24.07	52.63
c7552	77.34	57.14	33.9	3.08	-	-	-	-
alu4	94.98	95.45	48.85	16.56	-46.57	-74.97	72.67	79.11
apex2	78.30	78.05	33.98	12.08	130.19	7.85	-51.97	5.18
apex4	99.43	98.86	85.28	75.5	-95.29	-97.79	99.31	99.46
des	87.53	80.29	52.44	30.27	-30.86	-67.61	67.12	77.41
ex5p	97.47	96.88	67.77	46.35	59.27	-25.38	48.67	59.97
misex3	89.78	89.89	52.36	22.29	-0.12	-53.21	52.42	63.64
seq	82.46	80.16	30.77	-12.92	89.48	-11.23	-31.17	-0.24
spla	87.92	85.43	53.33	31.56	-26.7	-65.66	65.79	76.5
pdcc	90.62	89.84	66.51	45.38	-12.37	-58.95	70.66	77.58
s1423	71.28	63.16	82.37	74.16	336.76	104.63	23.02	47.11
s5378	80.94	64.29	48.8	31.81	222.62	51.15	-65.2	-3.07
s38417	76.80	58.31	76.46	65.49	136.91	11	44.24	61.69
s38584	84.10	48.19	47.17	13.83	-	-	-	-
bigkey	91.19	83.52	88.42	80.72	106.71	-3.16	76.05	81.32
elliptic	66.58	48.65	88.07	80.53	173.44	28.11	67.37	75.06
Avg.	83.72	75.58	56.41	34.31	84.31	-13.65	29.87	51.64

†Benchmarks mapped with $MI=24$ to achieve smaller channel width; – Power-aware VPR could not run for these circuits.

*Cutmap reduced the partition count and total energy requirement for these benchmarks.

we note that on an average, the MBC framework requires 84.31% more energy compared to the FPGA. We also have compared the EDP estimates to investigate the energy efficiency of the two frameworks. From Table 6.5, we note that on an average, MBC improves EDP by 29.87% compared to the FPGA framework at 1.1V. From Table 6.5, we note that at 0.8V, on an average the proposed framework still offers a 34.31% improvement in performance. The average improvement in energy is 13.65% . The average improvement in EDP for MBC over FPGA is 51.64% at 0.8V. Energy efficiency for MBC is more evident at lower supply voltages.

6.1 Design space exploration for MBC based stand-alone reconfigurable computing

Figure 6.4: Scaling of logic and routing delay components across technology nodes for FPGA and MBC frameworks.

6.1.3.3 Improvement in resource usage

From Table 6.5, we note that a migration from the 7-input LUT in FPGA to a 12-input LUT mapping for MBC reduces the total LUT count by 83.72% on an average. Considerable savings (75.58%) is also obtained in the total number of computing elements (FPGA clusters and MLB for MBC framework). Local execution and time-multiplexed use of programmable interconnects in the MBC framework leads to 68.87% savings in total wirelength compared to the baseline FPGA. This improvement is clearly depicted in Fig. 6.3(c-d) for a benchmark circuit mapped to both MBC and FPGA frameworks.

6.1.3.4 Technological scalability of performance

For the case of MBC, only 8.19% of the total delay is contributed by external routing. Since delay for general-purpose interconnects improves at a lesser pace compared to the logic delay or delay through intra-MLB interconnect, we expect the performance improvement attained in MBC to be more marked for future technology nodes. The FPGA delays at 45nm were estimated using the scaling trend for Altera Stratix family. From 90nm to 65nm, the routing delay scaled by

a factor of 0.87 , while the logic delay increased by a factor of 1.11 . The overall delay scaled by a factor of 0.86 . This reduction in the routing delay and increase in the logic delay is due to the change in FPGA logic-block architecture from Stratix II to Stratix III family (121). Fig. 6.4 shows the technology scalability from 65nm to 45nm technology nodes. For benchmark circuits mapped to Stratix FPGA, the average net delay contribution at 65nm is 77%, which we assume to scale at a value of 0.87 while the logic delay (23%) is assumed to scale at a factor of 1.11 . Similarly, the contributions from logic and net delays for MBC at 65nm are 92% and 8% respectively. Since the MLB cycle time improves from 680ps at 65nm to 560ps at 45nm, the logic delay for the proposed framework scales at a value of 0.82 while the scaling factor for the routing delay is taken to be same as that for FPGA (0.87). With these scaling trends, we note that MBC framework offers 10% better scalability compared to FPGA platform (Fig. 6.4).

6.2 Design space exploration for MAHA framework

In this section we explore the design space for the MAHA framework. The design choices made for such a framework directly affect the integration density for the memory array which is instrumented to realize the MAHA framework. The important design choices to be made are: i) width of the custom datapath inside each MLB; ii) optimal number of slices and levels inside each MLB which allows flexibility in mapping designs as well as incurs minimal area overhead; iii) the hierarchy organization of the memory array which is suited to reconfigurable computing and iv) interconnect bandwidth which will allow inter-MLB communication for the mapped application.

6.2 Design space exploration for MAHA framework

Table 6.6: Simulation setup for MAHA hardware exploration

Benchmarks		Constant Params		Variable Params	
Domain	Name	Name	Value	Name	Range
Security	AES (122)	Row_subarray	128	Datapath_width	8,16,32
	SHA-1 (123)	Col_subarray	16384	Log_num_slices	0,1,2
Image Proc.	Motion Estimation (9)	Granularity	4	Log_num_levels	2,3,4
	ATR(9)	Reg_count	8	Maxcustom	1,2,4,8,16
	Color interpolation (124)	PR_size	1KB	banks	1,2,4,8
Signal Proc.	2-D DCT(125)	Utblk _y	64	subbanks	1,2
	DWT(126)	Utblk _x	16	mats	1,2
Infomatics	Smith Waterman (127)			subarrays	1,2,4,8
	Max entropy			Buswidth	64,128,
	Classification (128)				256,512,
	Census				1024

6.2.1 Benchmarks and simulation setup

Contrary to benchmarks based on fine-grained random logic used to validate the MBC based stand-alone reconfigurable computing platform, we choose a number of applications selected from different domains to validate the effectiveness of the MAHA framework. *Critical sections* or *hotspots* from these algorithms are mapped to the MAHA framework which in this case acts as a hardware accelerator. These benchmarks are:

- i. *Advanced Encryption Standard for 128bits* or *AES-128* and *Secure Hash Algorithm 1* or *SHA-1* from the *Security* domain.
- ii. *Motion Estimation*, *Automatic Target Recognition* or *ATR* and *Color Interpolation* from the *Image processing* domain.
- iii. *Discrete Wavelet Transform* or *DWT* and *Discrete Cosine Transform* or *DCT* from the *signal processing* domain.
- iv. *Smith Waterman*, *Maximum Entropy Classification* and a synthetic application of *Census* data which belong to the *Informatics* domain.

Details of the simulation setup used for this exploration are provided in Table 6.6. As listed in Table 6.6, we have assumed that the memory size per subarray

Figure 6.5: Methodology for exploring the hardware design space of the MAHA framework.

of the instrumented memory is $Row_subarray \times Col_subarray = 256KB$.

6.2.2 MLB architecture exploration

Fig. 6.5 shows the methodology for exploring the hardware architecture of the MAHA framework and arriving at the final memory configuration which would support the MAHA framework. As illustrated in Fig. 6.5, the final design choice is guided by the optimization goal, i.e. minimizing the area, energy or maximizing the performance of the mapped applications. Through synthesis and SPICE level simulations at PTM 45nm (45) technology node, we first find out the area, energy and performance for the individual components of the MLBs and the routing

Figure 6.6: a) Area and b) Energy overhead for different components of the custom datapath for different datapath widths.

switches which would be used to route the inter-MLB signals for each MLB and routing switch configuration. Then during exploration step of our software flow, each configuration, these area, performance and energy numbers are collated to obtain the total area overhead, energy requirement and performance for the mapped application. Finally based on the optimization objective, one of the configurations is chosen for the final implementation.

6.2.2.1 Choice of the datapath width

Different applications have different requirements for the width of the custom datapath. For example, *SHA-1* benefits from a 32-bit datapath while *DCT* requires a 16-bit datapath. Choosing a datapath of appropriate size is therefore a trade-off between performance and design overhead. We have varied the width for the datapath as illustrated in Fig. A.1 as 8, 16 and 32 and noted the corresponding impact at 45nm technology node. The results are provided in Fig. 6.6.

6.2.2.2 Choice of number of levels of execution inside MLB

Increasing the number of execution levels inside the MLB increases the temporality inside the MLB, at the cost of increasing the *schedule table* size. We have varied the number of levels as 4, 8 and 16 and noted the impact on the *schedule table* area and energy.

6.2.2.3 Choice of number of slices in function table

Number of parallel LUT accesses from the *function table* are denoted by the parameter *Log_num_slices*. The actual value for the number of parallel LUT access is $2^{\text{Log_num_slices}}$. Increasing this design parameter, increases the number of parallel operations inside the MLB at the cost of increase in hardware overhead. It directly increases: i) the *schedule table* area since each row of the *schedule table* has to store more instruction opcodes and ii) the *muxtree* area which has to select more operands for the parallel operations. The increase in area and energy for the *schedule table* and *muxtree* for different values of *levels* and *slices* in the MLB design are shown in Fig. 6.7(a) and (b) respectively.

6.2.2.4 Choice of the memory hierarchy and interconnect bandwidth

Memory arrays are conventionally organized for maximum integration density and/or minimum energy dissipation. It is however important to explore the optimal memory organization for reconfigurable computing. For example, while some applications might require more local communication among its sub-operations, other might demand more global communication. For the first case, the number of *subarrays* and the bandwidth at each *subarray* must be high. The second situation however demands more number of *banks* and large *Buswidth* value. The memory organization and the *Buswidth* thus directly impacts the routability for the mapped application. Fig. 6.8 shows the percentage of benchmark circuits

Figure 6.7: a) Area and b) Energy overhead for different sizes of the *schedule table* and *muxtree*.

6.2 Design space exploration for MAHA framework

Figure 6.8: Routability for designs mapped to MAHA framework with different hierarchy and *Buswidth* configurations. The configurations are given as *Banks,subbanks,mats,subarrays,Buswidth*.

which are routable for different memory organizations and *Buswidth* values. Note that as the *Buswidth* is increased, the routability improves. Moreover, we notice that for the same number of physical resources (i.e. $Banks \times Subarrays \text{ per bank}$) it is more beneficial to provide more number of *Banks* than *Subarrays per bank*.

6.2.3 Variation in energy-efficiency with varying MLB design parameters

MLB parameters such as datapath width, number of levels and number of slices have considerable effect on i) area for each MLB; ii) number of MLBs; iii) total MLB energy; iv) total latency per input vector; v) total interconnect area and vi) total interconnect energy. While the actual impact on area and energy depends on the area and energy for the *function table* access and therefore the exact memory technology used, it is possible to note the impact on the number of MLBs and performance in cycles for the mapped applications as the design parameters are changed. Fig. 6.9(a) and (b) shows the effect on latency (in terms of cycles) and MLB count for benchmark *SHA-1* as the MLB design parameters are varied. For a particular set of design parameter values; the minimum latency and MLB count points have been plotted. Missing point at a particular configuration indicates such a configuration did not go through *packing* or *routing* due to insufficient logic or routing resources. As expected, *slices=2* has lower latency and lower MLB count compared to *slice=1* and *slice=0*, since increasing the number of

Figure 6.9: Variation in a) latency and b) MLB count for *SHA-1* with variation in i) levels; ii) slices and iii) datapath width.

slices, increases the number of parallel operations per clock cycle. A similar effect going from *datapath=16* to *datapath=32*. As the number of levels is increased, the MLB count decreases, since each MLB is able to accommodated more operations. Finally, latency should not ideally change with the number of levels. However, for *slice=1*, we note an increase in latency due to increase in *levels*. The reason being, the for *level=4*, the number of vertices in each MLB increases which increases the requirement for MLB I/O. Hence, a *slice=1,level=4* can only pass routing if the number of parallel operations is restricted, i.e. *Maxcustom* is less which negatively impacts the latency.

6.3 Optimization in application mapping

This section describes the software opportunities for improving the energy requirement of the MBC framework.

6.3.1 Content-aware mapping

The main computing fabric for the MBC framework is memory. For memory technologies such as SRAM and STTTRAM, it is possible to skew the memory cells such that memory cells containing 0/1 have better read energy compared to cells storing 1/0 (91, 92). In such scenarios, it is possible to intentionally skew the LUTs stored in the 2-D memory array (function table) to contain more 0/1 than the other. This leads to an improvement in the average memory access energy. We have therefore proposed a content-aware application mapping methodology which exploits this asymmetric nature of memory cells and amplify the energy savings by storing more logic ‘0’ than logic ‘1’ or vice versa. Note that the concept of skewing LUT content has been explored for diverse reasons in the context of FPGAs. In (8), the authors attempt to improve the soft-error tolerance of FPGA configuration bits by storing more logic zeros in the LUTs. In (129), the authors have proposed to reduce the leakage power consumption in FPGA by correct selection of polarity for the configuration bits. Here we present a greedy heuristic which reduces active power consumption in the MBC framework by preferentially skewing the ratio of logic ‘0’s to logic ‘1’s in the LUTs.

6.3.1.1 Heuristic for content-aware mapping

Algorithm 3 presents the pseudocode for maximizing the percentage of logic ‘0’s stored in the LUTs mapped to the memory array. The procedure to maximize the percentage of logic ‘1’s follows similarly. Input to Algorithm 3 is the hypergraph representation $(G(V,E))$ of the partitioned netlist and the truthtables for the individual partitions. The partitions are first leveled following their topological order in the netlist. Then beginning from partitions present in the first level,

6.3 Optimization in application mapping

Algorithm 3 *Content-aware Mapping to maximize the percentage of logic ‘0’ in the LUTs*

```
1: Input: 1) DAG representation  $G(V,E)$  of the partitioned netlist. 2) Truthtable  $T_j$  for the individual
   partitions ( $Part_j$ ).
2: Objective: To maximize the percentage of logic ‘0’ in the truthtables of the individual partitions.
3: Levelize partitions. Let the total number of levels be  $L$ .
4: for  $i = 1$  to  $L$  do
5:   for  $Part_j \in Level_i$  do
6:     if Output from  $Part_j \notin$  PO or Flop Inputs then
7:       if in  $T_j$ , # of ones  $>$  # of zeros then
8:         Flip 0’s and 1’s.
9:         Rearrange the 0 and 1 locations for the LUTs at the fanout of  $Part_j$ 
10:      end if
11:    end if
12:  end for
13: end for.
```

the truthtable of the individual partitions are examined. If the truthtable is found to contain more logic ‘1’s, then each LUT location is inverted. This is only done for LUTs which do not drive any primary output of the circuit or input to any internal state element. This is done to avoid reversing the polarity of the primary or state element output. In order preserve the correctness of the logic in the following levels, truthtable for the LUTs in the fanout of the modified LUT are rearranged. This rearrangement involves a simple reordering of the 0 and 1 locations inside the truthtable and does not modify the total 0 or 1 count.

6.3.1.2 Improvement in logic ‘0’ count

We have validated the effectiveness of the proposed mapping approach using a set of standard circuits chosen from the ISCAS’85, ISCAS’89 and MCNC benchmark suites mapped to 12-input 4-output LUTs. Table 6.7 shows the percentage of logic ‘0’ count in the LUTs for the target circuits before and after content-aware mapping. From Table 6.7, we note that the proposed mapping heuristic achieves

6.3 Optimization in application mapping

almost 49% increase in the percentage of logic ‘0’ count stored in the LUTs.

Table 6.7: Improvement in stored logic ‘0’ count achieved through skewing of LUTs

Ckts	# of Partitions	% of Logic ‘0’s		% Impr
		Before Opt	After Opt	
c3540	80	44.44	73.25	28.81
c5315	85	35.17	75.04	39.87
c6288	77	17.71	82.87	65.16
c7552	109	28.45	74.68	46.23
alu4	44	27.45	91.98	64.53
apex2	254	33.87	90.24	56.37
apex4	5	46.68	64.8	18.12
des	115	24.02	75.99	51.97
ex5p	16	46.51	96.51	50
misex3	88	21.78	94.99	73.21
seq	220	27.24	90.06	62.82
spla	204	25.5	88.83	63.33
pdc	241	27.36	91.24	63.88
s1423	41	54.21	68.3	14.09
s5378	106	42.76	82.01	39.25
s38417	794	31.22	77.33	46.11
s38584	745	31.36	81.42	50.06
bigkey	160	18.61	81.55	62.94
elliptic	102	42.92	81.55	38.63
Avg.				49.23

6.4 Preferential memory design for MBC frameworks

Since memory is primarily used for computation in a memory based reconfigurable framework, memory access is dominated by read rather than write, which occurs only during the process of application mapping. For a given technology, optimizing a memory cell for both read and write pose contradictory requirements (70). However, exploiting the read-dominated memory access pattern in reconfigurable frameworks, it is possible to come up with a novel memory cell design which offers better read power and performance at the cost of increased write power and performance. Dense memory arrays with the new memory cell would therefore offer better energy efficiency when used for computing during normal operation. Details of these circuit-level optimizations corresponding to CMOS based SRAM technology and emerging STTRAM technology are provided in this section.

6.4.1 Case study - I : SRAM array

6.4.1.1 A novel SRAM cell design

In order to improve the read performance and minimize the read power, we propose a novel 6-T SRAM cell (Fig. 6.10(a)) which employs one-sided write and dynamic read methodology (91). This improves the EDP during normal operation of the reconfigurable framework at the cost of higher write power during infrequent reconfigurations of the platform. The proposed SRAM cell benefits the MBC framework in terms of increased SNM, more tolerance to process variability and reduced read power consumption. Also the proposed cell being essentially a 6-T structure the area overhead associated is less than in 7T or 8T structures proposed earlier (130). However, writing to the cell is single-ended and requires word line boost up methods thereby consuming more write power than the conventional 6-T structure. So the proposed structure's energy efficiency over the

Figure 6.10: a) The proposed 6-T SRAM cell; b) Pathological case showing the flow of static current from an unselected cell to the selected cell; c) Pulse generation circuit for read operation.

conventional case is a direct function of the read write ratio. The proposed cell considers reading by controlling the source of the access transistor for read. This concept has been explored earlier in (131). However, the solution proposed in (131) suffers from the fact that the bitline for the cells are coupled via the high impedance source. This might lead to corrupting data of the unselected cells and giving false reads. We eliminate this problem by maintaining the source of the access transistor at low impedance and using a pulsed read mechanism. This decouples the cells effectively. More details on the read/write mechanism of the proposed SRAM cell are provided in (91).

6.4.1.2 Read-stability

The proposed memory structure has an improved read-stability. This is due to decoupling of the read and writes such that the cell being read from is isolated from the bitline so that there is no issue of the cell flipping while reading. The SNM curves in Fig 6.11(a) illustrate the read-stability advantage of the cell at

supply=1V. At lower supplies, the distinction is even more prominent as shown in Fig 6.11(b). This is because of the fact that due to the smaller supply voltages, read disturb will dominate even more for standard 6-T cell whereas the charge at the node 'X' still remains decoupled for proposed cell structure.

6.4.1.3 Read energy as a function of the pulsed read

In order to address the pathological case as illustrated in Fig. 6.10(b) and minimize the static power dissipation for the worst case condition, we have proposed a pulsed read mechanism. Details of the pulsed read mechanism can be obtained from (91). The amount of static power dissipated in the proposed memory cell is a direct function of the length of the pulse. For reliable operation of the proposed cell and to achieve power savings over a conventional 6-T cell, we have selected a pulse width of 150ps.

6.4.1.4 Operation at higher frequency

Fig. 6.11(c)-(d) indicate that the proposed cell is clearly a better choice for low voltage and high frequency operations. Even considering 200mV of necessary drop instead of the customary 100mV drop for the differential detection across SRAM, the cell has 2X lower access time than the 6-T structure. This essentially means an increased frequency of operation is possible for the proposed cell. More details on high frequency of operation of the proposed SRAM cell can be obtained from (91).

6.4.1.5 Writability

Writing to this cell being single ended, to ensure reasonable writing time, we overdrive the write bit line and the word line. The cell voltage is maintained at 0.6V whereas the W_{BL} and W_L are maintained at 1V while writing. Whereas the unselected cells of the selected column do not have problems for such a scheme this can give rise to stored data being flipped for the unselected cells along selected

Figure 6.11: a) The proposed cell offers a higher SNM compared to the conventional 6-T SRAM at nominal voltage; b) The improvement in SNM becomes more marked at lower supply voltages; c) a) The proposed memory cell has improved access time compared to the conventional 6-T SRAM cell. The improvement in access time is higher for lower supply voltages; d) The proposed cell also achieves a 40% reduction in the read power at nominal supply voltage.

row. This may be avoided by maintaining the unselected cells along the selected row at higher voltage ($V_{dd}=1V$). More details on the writability of the proposed cell can be obtained in (91).

6.4.1.6 Asymmetric behavior

The proposed memory cell is asymmetric with respect to the power consumed during read '1' and read '0' operations. Due to higher read power during the read '1' operation, it is intended that the memory location being accessed contains

6.4 Preferential memory design for MBC frameworks

Figure 6.12: Reading from a cell storing logic ‘0’ requires less read power compared to a cell storing logic ‘1’. The total read power thus decreases as the probability of the memory locations storing ‘0’ increases.

logic ‘0’ rather than logic ‘1’. Furthermore, for the read ‘1’ scenario, it is desired that the unselected rows in the same column have higher number of logic ‘0’ to reduce the static power dissipation during the pulsed read operation. Fig. 6.12(a) shows the reduction in read-‘1’ power as the number of cells storing ‘0’ in the same column increases. This asymmetric behavior of the memory cell in storing ‘0’s and ‘1’s is exploited by the content-aware mapping heuristic proposed earlier to skew the LUTs contents for improving the read energy.

6.4.1.7 Improvement in EDP for MBC and EMB based heterogeneous FPGAs

For the proposed MBC based stand-alone reconfigurable framework, we have noted that for conventional 6-T SRAM based MBC framework, the memory array contributes 29.5% to the MLB cycle time and 24.3% to the MLB power per cycle. With the preferentially designed SRAM cell, we note that the MLB cycle time improves by 12.7% and the power improves by 5.2% respectively. Since the proposed SRAM cell improves the performance and power for each memory access, an EMB based heterogeneous FPGA (86, 87) can greatly benefit from the

use of the proposed memory design. Standard benchmark circuits were mapped to 12-input LUTs using the *Heteromap* mapping algorithm proposed in (93). Fig. 6.12(b) shows the delay and energy results for the combinational MCNC benchmarks mapped to the EMB based heterogeneous FPGA framework. From Fig. 6.12(b), we note that the read-optimized memory cell along with content-aware mapping approach achieves a *36.21%* improvement in EDP in a EMB based heterogeneous FPGA framework. Since the routing energy remains the same for the two scenarios, we have only compared the logic energy with and without the proposed co-design approach.

6.4.2 Case study - II : STTRAM array

STTRAM (75, 80, 92) is a promising non-volatile memory technology which has a number of distinct advantages over the other prevalent memory technologies. Having essentially a 1T-1R structure it forms a dense array with high integration density and being magnetic in nature is tolerant to particle hits. In these features it is better than the SRAM. Being non volatile, it does not require refreshing like DRAM. It has a high write endurance, which makes it a better candidate than Flash memories. Another feature of STTRAM is its scalability which makes it an attractive option at scaled dimensions. In the nanometer nodes, leakage is one of the primary forms of energy dissipation. The STTRAM structure has zero standby leakage. All these attributes make STTRAM extremely attractive as a reconfigurable computing fabric. We have demonstrated that a spatio-temporal MBC architecture realized with 2-D STTRAM arrays, a STTRAM memory cell optimized for read and a content-aware application mapping approach can be integrated into an unified design flow to realize an energy efficient reconfigurable framework (92). In the proposed non-volatile MBC framework, the function table inside each MLB is realized using STTRAM arrays. This offers the following benefits: i) since the function table holds the configuration for the partitions, it occupies the maximum area inside a MLB. A small footprint of the Magnetic Tunneling Junction (MTJ) device (75) ensures that the area occupied by this

6.4 Preferential memory design for MBC frameworks

Figure 6.13: a) Design of STTMRAM cell for MBC framework to achieve optimal read energy; b) read energy with varying write probability.

memory array is minimized; ii) non-volatile nature of the STTMRAM array ensures that configuration bits stored in the function table are retained when power is turned down; iii) high read performance and low read power for the STTMRAM array results in considerable EDP improvement for a STTMRAM based non-volatile MBC framework.

6.4.2.1 Novel STTMRAM cell design

Exploiting the read dominated nature of the memory access in the MBC framework, the STTMRAM cell can too be optimized for read operation at the cost of write. Similar to the 6-T SRAM cell, the design space for STTMRAM is constrained by the readability and writability conditions i.e. tunnel magnetoresistance (TMR) ratio and write current requirement. Given a MTJ, the choice of the access MOSFET width (W) and its wordline voltage (V_{WL}) can be used to navigate the design space of TMR and write-current. A key aspect of the solution is its dependence on the read-write probability. Fig. 6.13(a) shows two solutions corresponding to write probabilities of 0.5 and 0.1, respectively. A larger write probability means a solution with larger width and smaller V_{WL} as write has a quadratic dependence on V_{WL} . From read perspective, a solution with lower width is preferred due to less leakage power dissipation. For MBC with read-dominant access pattern, the $W - V_{WL}$ configuration corresponding to equiprobable condition is not an opti-

mal choice as it dissipates higher read energy (Fig. 6.13(b)). Hence we choose the optimal energy point corresponding to low write probabilities which provides much lower read and total energy at the expense of increased write energy. Fig. 6.13(b) shows 24% saving in total energy for write probability of 10^{-5} for an 8-bit 64x64 memory array with a read access time of 400ps.

6.4.3 Improvement in EDP for STT RAM based MBC

We have performed simulations with MTJ at 65nm node with Resistance-Area product $30\Omega - \mu m^2$. The sizes of the MTJ devices have been taken as $50 \times 90 nm^2$ which requires approximately $60\mu A$ of switching current assuming current density of $10^6 A/cm^2$. The high and low resistance states are represented by $11.1k\Omega$ and $6.67k\Omega$, respectively. For the required STT RAM characteristics, we use the MTJ device characteristics as reported in the paper. The resistive values for the parallel and anti-parallel states at the read and write voltages are of consequence to us. The abstracted resistive behavior of the MTJ is simulated in conjunction with NMOS device at 65nm predictive technology model. The simulation for the STT RAM cell is done using HSPICE for all our measurements. The transistor was designed so as to be able to drive the switching current under both write '0' and write '1' conditions. To obtain the solution for varying write probabilities, first a host of simulations with the high and low resistance is performed for a range of V_{WL} and W . The solution space has to be extracted from the generated design space considering the constraints on minimum TMR and switching current. In this work we consider minimum TMR and switching current requirements of 0.34 and $60\mu A$, respectively. Corresponding to this extracted feasible design space of $V_{WL} - W$, we evaluate the read, write, active leakage and total energy of STT RAM array. The energy evaluation considers the read and write probability ratios. The $V_{WL} - W$ combination which gives the minimum energy is identified as the optimal energy solution. For demonstration of the results we selected write probabilities of 0.5 and 0.1 to evaluate the design points. The read and write energies for '0' and '1' are computed for these design points.

Figure 6.14: Improvement in: a) delay and b) Energy Delay Product (EDP) for STTTRAM MBC over conventional SRAM-based FPGA.

Delay and energy requirement for the CMOS elements of the MLB were obtained through SPICE simulations using predictive models at 65nm technology node (45). Delay and energy estimates for the MBC framework with the read optimized STTTRAM cell was obtained from the integrated automation flow described in Chapter 5. These were then compared against that for a 7-input LUT, 10-LUT cluster FPGA model at 65nm technology node (117). Fig. 6.14(a,b) shows the improvement in performance and energy-delay product for STTTRAM based MBC over the baseline CMOS FPGA model. As we note from Fig. 6.14(a), for standard benchmark circuits on an average the MBC framework improves the performance by 45.4%. Fig. 6.14(b) compares the EDP values between the two frameworks. The non-volatile MBC framework achieves a 5% improvement in EDP over the CMOS FPGA framework. The performance and EDP computation includes the cell optimization for read operation. The EDP improvement is further enhanced through the content-aware mapping step which skews the LUT contents to have more logic ‘1’ than logic ‘0’s. As a result of this content-aware mapping, the average EDP improvement was calculated to increase from 5% to 6.64%.

6.5 PLA based application mapping in MBC

Although the integration density for present and emerging memory technologies are expected to significantly improve in the future (132), a judicious use of the memory available is always welcome since it i) reduces the total memory requirement, which leads to faster cycle time, b) improved energy/vector requirement and c) potential to map larger application at each computational unit of a memory-based computing framework. Although most functions can be efficiently represented using a PLA-based representation, a reconfigurable platform using such a representation requires a different memory structure such as Content Addressable Memory (CAM), which has built-in search capabilities. Modern CAM implementations are usually large with word length ranging from 36 to 144 bits and address space from 7 to 15 bits (133). In spite of large size, such CAM designs allow access times as low as 0.25ns (134). Since a CAM-based PLA representation of a function can lead to better utilization of the storage space, it is worthwhile to investigate the effectiveness of CAM structure for memory based computing.

Although, efforts (135) have been made to configure a small storage at each CLB (in a conventional SRAM-based FPGA) both as a CAM and a LUT for efficient functional representation, the limited number of inputs to a CLB fail to exploit the high integration density offered by modern CAM designs. For efficient utilization of the die real estate, it is therefore desirable to have a large content addressable storage, which can be configured as multi input-output PLAs. Embedded System Block (ESB) of the APEX20K from Altera Corporation incorporates such an embedded memory, which can perform as a CAM-based CPLD (Complex Programmable Logic Devices) structure (136). However, such an implementation maps only the onset or offset of a function and cannot exploit the optimization obtained by consideration of don't care terms. We have therefore proposed a novel implementation of the memory based computing paradigm that leverages on the high integration density of modern CAMs. The idea is to realize the *function table* inside the MLB using a dense 2-D CAM array and store the optimized PLA representation of the partitions in the CAM array. Similar to LUT

Figure 6.15: NOR based T-CAM cell implementation and its encoding.

based MBC framework, these partitions are evaluated in a topological order over multiple clock cycles. We have also investigated a hybrid CAM-LUT based function representation which further reduces the memory area by selectively storing some partitions in the CAM, while others in LUT.

6.5.1 Function representation using CAM based MBC framework

6.5.1.1 Optimized PLA using Ternary CAM (T-CAM)

In order to evaluate a partition in memory given its PLA representation, we need to match the inputs to the PLA representation that is already stored in the function table. One hurdle for such an optimized implementation is that for many entries in the ON-set of a function, one or more function inputs may have ‘X’ (don’t care) values. We address this challenge by modifying the conventional CAM structure to store and match ‘X’ values regardless of the input bit, similar to (133). Such a CAM structure is often referred to as Ternary CAM (T-CAM). However, in order to store 3 logic values (0, 1 and X), each memory cell must contain two storage bits. Thus, the data stored in a T-CAM cell can be matched against either 1, 0 or X on the input lines. Fig. 6.15 shows the NOR-based implementation of a single T-CAM cell with the corresponding encoding for the logic levels. Based on the observation that a collection of T-CAM cells have a

structure similar to the ‘AND’ plane of a PLA, it is possible to construct a PLA structure in the memory with the ‘OR’ plane being constructed from the normal SRAM cells, which hold appropriate values for the output response.

6.5.1.2 Design and organization of T-CAM

The organization for a given memory bank is shown in Fig. 6.16, which can store up to ‘n’ partitions, where each partition has a maximum of ‘k’ product terms. During each cycle, information stored in the schedule table is used to enable the match lines for one of the ‘n’ partitions stored in a given memory bank. Within each partition, it is possible for more than one match to occur, thereby enabling more than one match line. One of the access transistors for the 6T-SRAM cell is gated by the match line, so that data is read out only if a match occurs. In case a matching PLA entry of a given function has logic value of ‘1’, the node ‘V’ is discharged through the corresponding NOR leg. The drivers for the input data and the current sense amplifiers are shared for each memory bank, thereby reducing the interface logic required. Note that partitions with product terms less than ‘k’ have $D = D_c = 0$, which always leads to a mismatch in the unused match lines.

6.5.1.3 A hybrid CAM-LUT approach

As illustrated in (135), a PLA representation is good for complex inter-related function while LUT based representation as in traditional FPGAs is suitable for small complex gates. Such a scenario is often encountered for circuits that require many “XOR”-like logic function leading to exponentially large number of product terms in the PLA representation. Hence, a framework containing both PLA and LUT based representation is advantageous for memory-efficient realization of all classes of functions. We propose using hybrid CAM/LUT-based storage for the memory based computing framework. The partitioning algorithm was augmented to allow the user to choose a LUT based implementation for the partitions for which a CAM-based implementation of the product terms requires

Figure 6.16: Proposed T-CAM based realization of individual partitions.

a larger area compared to the corresponding LUT area. Thus, a hybrid approach can potentially improve the area requirement for the total memory to which the application is mapped.

6.5.2 Simulation results

6.5.2.1 Delay and power estimation

Spice simulations were carried out with PTM 70nm technology model (45) for the CAM-based MLB framework. Simulation results for the example MLB with associative storage indicate that the delay for each stage of the pipeline is within 0.6ns for four memory banks, each with 16KB of memory. The total power consumption for a cycle time of 0.6ns and input switching activity of 12.5% (default switching activity for Quartus v7.0 simulator by Altera) was estimated to be 10.46mW.

6.5 PLA based application mapping in MBC

Table 6.8: Comparison of memory requirement, area, delay and energy/vector for 12X12 partitions

Ckts	Mem. Req (KB)		Area		Delay (ns)		Energy/Vector(pJ)		Leak Pwr(μ W)	
	LUT	CAM	LUT	CAM	LUT	CAM	LUT	CAM	LUT	CAM
C880	37.6	1.11	301	11.4	3.5	3.5	29.57	36.61	2.98	0.49
C1908	20	3.41	160	35.3	2.5	3	21.13	31.38	1.58	1.52
C2670	81.7	1.93	653.6	19.7	5	5	42.25	52.3	6.47	0.85
C3540	130.9	2.68	1047	27.2	8.5	8.5	71.83	88.91	10.36	1.17
C5315	111.4	4.13	891.4	42.3	9	9	76.05	94.14	8.82	1.83
C7552	180.1	7.83	1441.1	80.3	11.5	11.5	97.18	120.29	14.27	3.47
C6288	216.2	12.9	866.4	130.8	9.5	10	80.28	104.6	8.57	5.65
DCT	44	15.83	352	168.85	106	110	895.70	1150.6	3.48	1.67
FIR	46.75	16.76	374	178.83	14.5	16	122.52	167.36	3.71	1.77
Avg Improv.	+22.58X		+16.94X		-4.37%		-29.2%		+61.8%

Ckts	Mem. Req. (KB)	% improv.	Active area	% improv.	partitions converted to CAM
C880	1.11	0.00	11.3	0.88	23
C1908	3.17	7.04	32.5	7.93	19
C2670	1.93	0.00	19.7	0.00	36
C3540	2.68	0.00	27.2	0.00	61
C5315	4.13	0.00	42.3	0.00	69
C7552	7.63	2.55	76.8	4.36	85
C6288	12.6	2.33	126.9	2.98	111

Figure 6.17: a) Comparison of memory requirements between LUT and CAM-based implementations; b) Memory and area requirement with hybrid CAM-LUT implementations.

6.5.2.2 Improvement in memory requirement

Memory requirement was noted for both ISCAS-85 benchmark suite, DCT and FIR applications when mapped to i) LUT and ii) CAM based implementation of the multi-cycle computing framework. Table 6.8 compares the memory require-

ment, delay and energy/vector for the two scenarios. From Table 6.8 and Fig. 6.17(a) we can observe that a CAM-based implementation entails an exponential saving in memory requirement compared to a LUT-based approach. We have also observed that with CAM based implementations, it is possible to map partitions with larger input and output sizes (16×16) with significant savings in area (251X), with nearly the same speed-up as offered by a corresponding LUT based implementation (-7.62%). To observe the impact of hybrid CAM/LUT based application mapping, we performed simulations with several ISCAS-85 benchmarks. Table 6.17(b) shows the memory requirement (in KB) and area (estimated as active area, $W \times L$) with the proposed approach. As we can observe from Table 6.17(b), the hybrid approach can further improve the memory saving for some circuits, which consist of partitions with large number of product terms.

7

MBC using on-chip cache

Modern processors incorporate multiple levels of memory hierarchy in the form of on-chip caches. Level 1 (L1), level 2 (L2) and sometimes level 3 (L3) caches are present on the processor die. Typically L1 instruction and data caches are 32KB-64KB, L2 unified caches are typically 512KB-2MB while L3 caches are of much larger size (24MB) (68). In this chapter, we demonstrate how the on-chip cache in a processor can be used to perform computation on demand by storing the results of Boolean functions in LUT format. We note that such on-demand computation can be used for improving reliability of the processor.

7.1 Reliability improvement

Although technology scaling provides the capability of integrating billions of transistors in a modern processor, it gives rise to important issues such as high defect rate and device parameter variations. These issues have emerged as major barriers to future technology scaling (137). Increasing defect rate and device parameter variations in sub-90nm technology regime leads to reduced yield (138). In order to adapt to both manufacturing defects and Process-Temperature-Voltage (PTV) induced parameter variations, an effective solution would be to develop a

system which can dynamically detect and correct defects and variation-induced failures. A number of circuit and system level techniques have been proposed in this context (138, 139), (140), (141). These techniques however typically incur large performance overhead (139), (140) or do not address manufacturing defects (141). In this paper, we propose an architecture-level solution for improving processor yield and reliability in presence of both manufacturing defects and parameter variation induced reliability issues. The proposed scheme allows on-demand transfer of activity from functional units such as integer Arithmetic Logic Unit (ALU) and Floating Point Unit (FPU) to the embedded memory of the processor. Here we assume that under a fixed delay target, one or more logic units are non-functional while any hard defect or variation induced parametric failure in memory has been addressed using suitable redundancy (142) or re-mapping (138) techniques. In such a scenario, the memory can compensate for hard manufacturing defects as well as parametric failures (such as delay failures) in logic units. Moreover, the method can be used to address reliability problems arising from temporal parameter variations, such as variations due to thermal stress, by dynamically transferring activities of a functional unit (FU) to memory when the FU experiences high temperature.

The key idea is to realize the functionality of different execution units in a

Figure 7.1: Almost 60% of the integer instructions have bitwidth less than 16-bits for SPECInt2000 benchmarks.

processor, such as adder or multiplier using on-chip memory, which acts as a shared reconfigurable computing resource. We demonstrate that on-chip cache in a processor can be used to perform computation on demand by storing the results of Boolean functions in lookup table (LUT) format. The performance loss due to the memory based computation however remains within tolerable limits for benchmark applications. There are primarily two reasons for that: 1) the operands for a specific instruction (e.g. add/multiply) experience high locality of reference across different clock cycles thus requiring loading only part of the LUTs in the cache; 2) Most of the operations in the functional units are narrow-width (143). For these operations, the operation latency can be comparable to their regular functional latency using dedicated hardware. Fig. 7.1 illustrates the observation regarding abundance of narrow-width operands for integer operations in SPECInt2000 benchmark applications. It shows that over 40% operations have bit-width of 8 or less. Memory based computation can be effective for these operations since smaller operand width translates to fewer memory accesses. The proposed solution seamlessly integrates with the conventional pipelined execution flow in a modern processor. An additional hardware unit, which we refer as “glue logic”, is required for interfacing with the cache to construct the effective virtual and physical addresses during memory based computation. However, the overhead due to the glue logic is significantly smaller compared to replication of the entire functional unit.

7.1.1 Overall scheme

In order to study the effectiveness of the proposed memory based computation framework, we have applied it to realize the functionality of the integer execution unit (adder and multiplier). Fig. 7.2 shows the sequence of actions followed by the decision making hardware inside schedule and issue logic to bypass original functional unit for memory based computation. Once a need for bypassing the normal execution unit has been detected the processor will issue an indication for the Operating System (OS) to load the result tables for that particular operation

Figure 7.2: Flow chart showing different steps of the proposed memory based computation framework using on-chip cache.

in a section of the main memory. The OS returns the pagebase address where the results for the lookup table realization of the operation are being stored in the main memory. This pagebase address will be used for further access to the main memory to load the result tables to the on-chip cache. The pagebase address is used to form the physical address required to access the on-chip cache as well as the main memory. Note that the OS is responsible for mapping the virtual memory address space into the physical memory address space. Since the range of operands supported by MBC is known during the time of the design, the range of physical address space which holds the results of the MBC operations is also fixed during the design of the processor. The OS must ensure that no virtual address from the program that is being executed is mapped to the physical address space which holds the results for MBC operations. Details of integration of the proposed

Figure 7.3: Formation of virtual and physical addresses from the source operands for a particular operation.

scheme with the dynamic pipeline of a superscalar out-of-order processor can be found in (89).

7.1.1.1 Formation of effective address

The formation of virtual and physical addresses is illustrated in Fig. 7.3. Let us consider the addition of two 32-bit operands X and Y . Each of the operands can be considered as a combination of four 1-byte operands (X_0, Y_0) , (X_1, Y_1) , (X_2, Y_2) and (X_3, Y_3) . The addition result for such 1-byte operands can be stored in the memory in the form of lookup tables. The addition results can then be obtained by accessing the memory with an address that contains the information regarding the source operands. For the first access, we will consider that the operands are X_0 and Y_0 . The operands X_0 and Y_0 when concatenated form the special virtual address required for memory based computation. Since this virtual address cannot be part of the virtual address space defined by the compiler, a separate MBC mapping table is required to map this virtual address to its physical counterpart. The normal route for virtual to physical address translation through the Translation Lookaside Buffer (TLB) is therefore bypassed. When the requirement for

memory based computation arises for the first time, the CPU raises an exception and the exception handler of the OS returns with address for the pagebase which contains the results for the corresponding operation. For example, in case of addition, this would be the address for the page in the main memory which contains the addition results of all possible 8-bit operands. This idea is illustrated in Fig. 7.3 where the MBC mapping table stores the page base address for the operations which are intended to be supported using memory based computation. The page base address stored previously from the OS is concatenated with the page offset to form the complete physical address that selects a line of a page in the main memory. The selected line contains the addition results of the operand in the page offset with all possible 1-byte operands. Thus, the whole page of addition results is loaded into the main memory only once. After that for each combination of operands, all the addition results for the operand which is smaller of the two is loaded into the L1 cache from the main memory. Here, we utilize the commutative property of addition in that fetching the addition result for one of the operands will suffice. Note here that since two operands of size 8-bits have been considered for virtual address generation, for page table of size 64KB or more, the operands only need to be concatenated with the pagebase address for physical address generation. However, for smaller pagebase sizes, the pagebase address would require an addition with $2 * Operand_Size - Page_Offset_Size$ bits from the operands.

It is to be noted here that loading of the ADD/MULT results table would follow the same steps as loading of instruction/data into the main memory and from the main memory into the processor cache. When MBC is invoked for the first time, virtual to physical address mapping will encounter a page fault. The exception handling routine of the OS would load the corresponding pages into the main memory and load the virtual to physical translation into the page table. Subsequent access to the same virtual address would bring the mapping and the data into the MBC mapping table and the on-chip cache, respectively. The proposed activity transfer scheme requires additional hardware that generates the virtual and the physical addresses for accessing the cache and the main memory,

respectively. In order to exploit the commutativity property of the addition and multiplication operation, we use a 32-bit comparator for comparing the operands and aligning the larger operand to form the most significant bits. The same comparator can also be used for the multiplication procedure. Additional hardware required for multiplication includes i) a 32-bit shifter to obtain partial products of appropriate weight before they are added and ii) 32-bit priority encoder in order to minimize the number of memory accesses for the generation and addition of partial products in case of narrow width operands. More details on the functionality of the *glue logic* can be obtained in (89).

7.1.2 Cache and memory organization

Each line of the page in the main memory contains the addition result of one 8-bit operand (say, X_0) with all 8-bit operands ($Y_0, Y_1, \dots, Y_N$). Two such sets are present, one for the condition that input carry is zero and the other for input carry one. The page offset is used to access the addition results for both cases of the input carry. The input carry is then used to select the proper addition results. On a cache miss, the memory line containing the addition results of one of the operands (X_0) is brought into the cache and written at the location pointed to by the corresponding physical address. Cache line is accessed using the tag and index section of the physical address. The second operand (say, Y_0) which forms the block offset section of the address is used to access the result of addition for the two operands (X_0 and Y_0).

The hierarchy of cache access has been shown in the Fig. 7.2. The L1 cache is accessed with the physical address; miss on the L1 cache triggers an L2 cache access. A hit on the L2 cache causes the return of data which is also written into L1 cache. The valid bit is set on a successful write operation. If a free write location is not found in the L1 cache at the given physical address, then the data present in that location is evicted from L1 and written to L2 cache following a Least Recently Used (LRU) policy.

Figure 7.4: Implementation of memory based addition using carry-select addition procedure.

7.1.3 Memory based addition procedure

A scheme based on carry-select addition of two 32-bit operands using memory based computation is shown in the Fig. 7.4. If one of the operands is zero, the addition is completed in one cycle. If not, the 32-bit operands are bit-sliced into 8-bit operands. For each set of 8-bit operands, the addition result for both input carry zero and one is looked up from the cache. The input carry is then used to select one of the two results. The same operation is repeated for all the 8-bit operands. Thus the entire addition procedure is completed in two steps, a memory lookup and subsequent carry select addition using the 8-bit operand addition results. Note that due to the commutative property of “add” ($a + b = b + a$), total memory required to store all the “add” results is halved and comes to 64KB.

Considering the result for all the sub-operands (X_i, Y_i) are present in the on-chip memory, the worst-case evaluation time for two 32-bit operands is 4 cycles.

Although this evaluation time is more than that of respective functional units, due to the fact that most of the operations (almost half of the integer operation) are narrow width (143), the average penalty in performance is not significant. The exact latency of operation depends on number of memory accesses as well as the number of cycles required to access the memory. The later is determined by the location of relevant LUT in the memory hierarchy. dependent instructions, the schedule and issue logic follows the same procedure as satisfying LOAD dependencies for LOAD operations.

Fig. 7.5 shows the required modifications in the processor schedule and issue logic. The configuration unit creates a configuration vector based on the defect map obtained during testing phase or the dynamic temperature profile obtained from the temperature sensor. For permanent defects, the faulty execution units are detected during manufacturing test and the defect map generated is stored in a small N-bit (where N is the number of functional units) Read-Only Memory (ROM). For a superscalar processor, operands issued to the FUs are read from the Reorder Buffer (ROB). The logic for selecting the ready operands from the issue window is implemented as a tree of priority encoders (144). For the proposed

Figure 7.5: Modified issue logic with *Req* signal masking.

approach, when FU_k is defective or is under thermal stress, Req signal to the corresponding FU is masked (Fig. 7.5) and any instruction which is not serviced by the faulty/busy units is transferred to MBC unit. In our simulations using HSPICE, the delay overhead in the issue stage due to the small masking logic was found to be negligible. The memory based multiplication is achieved based on memory based partial product formation followed by memory based addition procedure. Details can be obtained in (89).

7.2 Simulation results

7.2.1 Test setup

To check the effectiveness of the proposed scheme, we implemented the activity transfer mechanism in an out-of-order superscalar processor architecture using SimpleScalar Version 3.0 (145) compiled for Alpha binaries. For this implementation, we considered the variable latency aspect of memory-based ADD/MULT operation in the same manner as done for load operations. Given a set of operands for memory based ADD/MULT operations, we also determine the position of their result in the memory hierarchy and schedule an event based on the number of cycles required to retrieve the result. Similar to the load/store instructions, the dependant instructions in the RUU are waked up when the result for variable latency ADD/MULT is available. The baseline configuration of the processor is taken from (146). The effectiveness of the proposed framework was validated for two different scenarios: 1) improving the reliability of operation under temperature variation and 2) improving manufacturing yield of a processor when some functional units become inoperative due to manufacturing defects. More details of our simulation setup is provided in (89).

7.2.2 Tolerance to thermal stress

We have measured the performance loss in case of different Spec2000 benchmarks due to MBC. Our observation is that when the proposed computational framework is used for temperature management in the integer ALU, the average increase in CPI is 3.9%. In this case, we assume that existing on-chip L1 and L2 caches is used for MBC. The impact on performance due to MBC is mainly two-fold: 1) increased operational latency compared to execution in the functional units and 2) higher capacity misses in the cache due to reduction in cache size. When an additional on-chip memory is devoted for on-demand computation, the average CPI degradation was observed to be 1.57%. From this, we can infer that:

- For processors targeted towards memory-intensive applications, an additional embedded memory can be incorporated for yield and reliability improvement without considerable performance impact.
- This additional memory still acts as a reconfigurable unit which can map one or more failing units and incurs less hardware overhead compared to the duplication of individual FUs.

Note that the CPI degradation in case of MBC is however much better over the DVFS scheme (139), which slows down the entire processor thus incurring a performance penalty of over 10%. More details on the individual benchmarks is provided in (89).

7.2.3 Tolerance to manufacturing defects

We have considered two defect scenarios: i) 2 adders and 1 multiplier are defective and ii) 4 adders and 1 multiplier are defective out of the 6 adders and 2 multipliers present in the baseline configuration. Fig. 7.6(a) and (b) compares the performance of a processor with and without MBC when multiple FUs are considered inoperative and an extra on-chip memory hierarchy is present to support MBC operations. Fig. 7.6(a) and (b) correspond to Configuration 1 and 2 mentioned above. In the proposed scheme, activities of these defective functional

Figure 7.6: Performance and power results for 2-add, 1-mul ((a) and (c)) and 4-add, 1-mul ((b) and (d)) defective scenarios.

units are dynamically transferred to the memory. As observed from Fig. 7.6(a), if we do not employ memory based computation, then the performance overhead for a faulty processor with configuration 1 is 5.5%. In case of configuration 2, the processor suffers a much higher performance overhead of 42.1% due to increased number of defective units (Fig. 7.6(b)). However, when the proposed activity transfer scheme is incorporated in the processor architecture, the loss in performance is considerably less (0.79% and 4.82%, respectively).

The smaller performance overhead for the proposed computation framework is due in part to the high degree of correlation that exists among the operands thereby reducing the number of L2 and main memory accesses. Besides, large percentage of operands have narrow width (less than 16 bits) which helps to reduce the number of memory accesses. Fig. 7.7 supports both observations for the benchmark applications. From Fig. 7.7, we note that on an average only

Figure 7.7: Number of memory bypass and the L1 miss expressed as a percentage of the total number of MBC operations.

12% of the MBC operations suffer an L1 miss which shows that the bit-sliced operands experience high locality of reference. Fig. 7.7 also plots the percentage of memory bypasses, which shows the impact of narrow-width operands. For example, a memory based add operation for two 32-bit operands would take 4 memory accesses in the worst case. If both operands are less than equal 8-bit, the number and percentage of memory bypasses are 3 and 75%, respectively. For the benchmark applications, we note an average of 71% memory bypasses. This matches with the observation in Fig. 7.1 which shows that addition for 60% of the operands can be achieved within 2 cycles. Using the above L1 miss rates and the bypass ratio, one can arrive at the estimate for worst-case evaluation time ($EvalTime_{worst}$) for adding two 32-bit operands. This estimate is close to 5 cycles as shown below.

$$EvalTime_{worst} = 4 \times [0.3 \times (0.88 \times 1 + 0.12 \times 6 + 0.001 \times 100) + 0.7 \times 1] \sim 5.$$

Table 7.1: Comparison of area overhead between MBC (due to glue logic) and resource duplication

Approach	Module	Individual	Total
MBC		Area(μm^2)	19324
	32bit shifter	11280	
	32bitPriority Encoder	3541	
	32bit Comparator	4503	
Resource Duplication	32bitAdder	10135	204555
	16bit Multiplier	194420	

7.2.4 Overhead in MBC scheme

As observed from Table 7.1, additional hardware requirement for implementation of the MBC scheme is only 9.5% of the duplicated functional units, when we use the existing cache. We have also noted the power overhead due to the proposed memory based computation framework. The results were collected using Wattch Tool Set Version 1.0 (147). Fig. 7.6(c) and (d) show the power overhead results for configuration 1 and 2. For memory based computation, we considered the power due to additional memory accesses and power dissipation in glue logic. From the results we note that for configuration 1, the proposed scheme incurs an average power overhead of 0.24%. However, a higher power overhead (average 1.23%) is incurred when the proposed scheme substitutes for 4 defective adders and 1 defective multiplier.

7.2.5 Yield improvement results

Fig. 7.8 shows the yield improvement due to the proposed activity migration scheme in case of such manufacturing defects. A chip is assumed to fail if the performance overhead due to the failing functional units is more than 10%. For

this analysis, we considered a specific implementation of integer and floating point units obtained from (148) which consisted of 600K transistors. We also considered an appropriate size of memory for MBC organized into 32X32 blocks with built-in redundancy of 2 columns per block. Defects were inserted into the transistors following a random distribution for a range of defect rates. In case of memory, some defects were tolerated using redundant columns. Depending on the number of failing units, the original chip suffers a performance degradation which is obtained from our simple scalar simulations for the benchmark suite. Even for low defect rate (region A in Fig. 7.8), the percentage of chips that do not meet the target performance can be appreciable. For these low defect rates, however, the performance overhead for the MBC scheme was lower than the 10% tolerance. In this region, the proposed scheme can improve the yield by almost 60%. As the defect rate increases, more functional units fail. Besides, less memory blocks become usable with increased device failure. As a result, the overhead in MBC also increases. Hence, yield for the proposed scheme also starts to degrade in region B. However, the chip failure rate becomes considerably better than the baseline case. For still high failure rates, the failure rates for the two schemes approach the same value.

Figure 7.8: Variation in chip failure probability for different device failure probabilities with and without the proposed approach.

Figure 7.9: a) Percentage power savings achieved for MBC operation by the introduction of Result CAM; b) Percentage CPI increase; and c) Percentage power saving results for MBC framework using Result CAM and the original cache, compared to the baseline CPU and MBC without Result CAM.

7.2.6 Use of result CAM

The performance and power overhead of the MBC operation can be considerably reduced by the use of a very small fully-associative memory which stores the addition/multiplication results for the most recently used operands. We refer to this memory as “Result CAM”. The Result CAM enjoys a high hit ratio due to the fact that the 64-bit input operands are broken up into 8-bit sub-operands which has high locality of reference. The power required to access Result CAM is almost 6X lower (obtained from Wattch (147)) than that required for accessing the L1 cache for MBC. As a result, introduction of the Result CAM considerably improves the Energy-Delay Product (EDP) for memory based computing. Fig. 7.9(a) reports the percentage improvement in power for a MBC framework using the Result CAM. The average power improvement for an 8-entry Result CAM is 71%, while for a 32-entry Result CAM, the percentage power improvement reduces to 65%. However, the use of a 32-entry Result CAM results in a higher hit ratio (avg. 61%) compared to the 8-entry Result CAM (avg. 33.5%). Fig. 7.9(b) and (c) shows the percentage CPI and power increase for the scenario when MBC uses a 32-entry Result CAM along with the original cache hierarchy

for the processor and a LRU policy. In these simulations, we consider the fact that original load/store operations in the load-store queue (LSQ) are delayed due to MBC operations accessing the original cache. More benefits of using the *ResultCAM* such as decrease in the memory traffic are listed in (89).

8

Improving the reliability of operations in MBC frameworks

Like any other computing framework at nanoscale technologies, memory based computing is also prone to defects and failures in the nano-devices. In this chapter we describe two approaches to mitigate the effect of failures in memory, which is the primary computing fabric in memory based computing. The first approach proposed for MBC based stand-alone reconfigurable computing, tries to preferentially map an input application so that unreliable memory blocks are avoided for computation. The second approach uses error-correction codes for mitigating runtime failures in on-chip memory which can be used for computing.

8.1 Mitigating the effect of parametric failures in MBC frameworks

Since MBC uses large, high-density memories for computation, reliable operation of the framework under increasing process variations becomes a major concern.

8.1 Mitigating the effect of parametric failures in MBC frameworks

Variation may potentially cause memory access failures or flipping of stored data during read-out (70), which leads to incorrect execution of the mapped application. Moreover, in order to reduce the power requirement, memory core is conventionally operated at lower supply voltages. Although this minimizes the active and leakage power consumption, read and access failure probabilities increase significantly at low operating voltages (70). In order to address variation induced memory failures in the MBC framework, we note that MBC operation is dominated by read, while write only occurs occasionally during reconfiguration. Exploiting the read-dominant memory access pattern in MBC, we propose a preferential memory cell sizing approach that makes the memory more robust to read and access failures. The resultant decrease in write stability is addressed by low-overhead circuit techniques during reconfiguration.

8.1.1 Effect of Parameter Variations on QoS

Parametric variation in nanoscale technologies can lead to incorrect execution of applications mapped to the MBC framework. Such errors in execution may arise from any one of the following sources: i) *Failure in memory*, ii) *Failure in Control Unit* and iii) *Failure in programmable interconnects*. In (149), we discuss in detail the effect of each of these failures on the MBC operation. As pointed out in (149), memory failures due to degraded cell stability is most difficult to address among all the parametric failures that may lead to Quality of Service (QoS) degradation at the MBC output.

Variation in process parameters (both systematic and random) have emerged as a major design challenge at both circuit and architecture level design (137). Such variations can cause mismatch between adjacent transistors in a SRAM cell, which may eventually lead to a failure of the cell itself. These failures include: i) *Access Failure* which results due to an increase in the cell access time; ii) *Read Failure* due to an unwanted flipping of the cell content while reading; iii) *Write Failure* due to the inability to successfully write to the memory cell and iv) *Hold Failure* which results due to the application of a supply voltage

8.1 Mitigating the effect of parametric failures in MBC frameworks

lower than the minimum voltage required by the cell to hold the data at nominal condition. Since, the above failures are caused by variation in device parameters, they are often referred to as parametric failures (70). Although alternate failure mechanisms exist for memory cells (70), we focus on the effect of parametric failures on MBC and their mitigation.

Let us consider a memory model consisting of 16KB of memory, divided into 128 blocks, each with 1024 cells. The memory cells in each block is organized into 32 rows (N_{ROW}) with 32 cells (N_{COL}) in each column. Note that depending on the maximum number of partition inputs and outputs specified during partitioning, a single memory block can store the entire LUT or only a part of it. For example, for mapping a LUT with 12 address bits and 8 data bits, 32 similar blocks will be necessary. The appropriate 8 bit selection will be performed through a combination of row-decoding and column-multiplexing. The number of redundant columns in each block (N_{RC}) is 2. We also consider pulsed wordline and bitline isolation architecture for reducing bitline swing and a nominal V_{dd} of 0.9V. We performed simulations with this memory using HSPICE for PTM 45nm LP models (45) considering both systematic and random variation in V_T . Details of the simulation can be obtained from (149).

From the Monte Carlo (MC) simulations performed on the cells inside each block, we noted the parameters V_{read} , V_{trip} , T_{access} and T_{write} for each cell. These parameters defined in (149) are commonly used to characterize the vulnerability of memory cells to parametric failures (70). As pointed out in (150),(151), a “weak” memory cell can undergo a read/write or access failure under supply voltage variation, high temperature or coupling noise. It is therefore necessary to identify such “weak” cells through post-silicon calibration techniques. In order to quantify the vulnerability of individual memory blocks, we introduce block level reliability metrics obtained by collating V_{read} , V_{trip} , T_{access} and T_{write} parameters from individual memory cells. The metrics are:

- Indicator for Read Stability denoted as $I(V_{trip} - V_{read})$
- Indicator for Write-ability denoted as $I(T_{write})$

8.1 Mitigating the effect of parametric failures in MBC frameworks

- Indicator for Access-ability denoted as $I(T_{access})$

In general the indicator $I(x)$ where $x \equiv V_{trip} - V_{read}$, T_{write} or T_{access} for a memory block can be derived by:

- Classifying the cells of the block into separate bins based on the value of parameter x
- Calculating a weighted average of the cells in each bin:

$$I(x) = \frac{\sum_{i=1}^N n_i * w_i}{\sum_{i=1}^N w_i} \quad (8.1)$$

In equation (8.1), w_i denotes the weight and n_i denotes the number of cells in the i_{th} bin. For our simulations, we have divided the cells in each block into 5 bins ($N = 5$) with $w_i = 2^{-i}$ values. The range of $x \equiv V_{trip} - V_{read}$ values for the bins are : i) $x \leq 0$, ii) $0 < x \leq 100mV$, iii) $100mV < x \leq 150mV$, iv) $150mV < x \leq 200mV$, and v) $x > 200mV$. Choice for the number of bins and the range of x values were primarily driven by the distribution of the $V_{trip} - V_{read}$ obtained in our simulations. For low supply voltages, considering a parametric variation scenario with $\sigma_{dV_{t_{sys}}} = \sigma_{dV_{t_{0_{rand}}} = 50mV$, $V_{trip} - V_{read}$ values for the ‘weak’ cells were mostly centered around a value of 50mV. Hence the 2nd bin

Figure 8.1: a) shows the variation in $I(V_{trip} - V_{read})$ for the memory blocks at fast corner. (b) Random mapping of DCT modules to memory blocks.

8.1 Mitigating the effect of parametric failures in MBC frameworks

was chosen for $0 < x \leq 100mV$. The weight assigned to each bin is primarily determined by the operating condition of the MBC framework and the relative vulnerability to failure for cells in each bin. Ideally the weights for the ‘weak’ cells should be exponentially large in order to distinguish them from cells with $x > 100mV$.

$I(x)$ for a given block serves as an indicator of the process corner to which the block has moved due to systematic variation. A block with a larger value of $I(V_{trip} - V_{read})$ is more prone to read disturb failures under increased environmental stress (i.e. reduced voltage and or higher temperature). Fig. 8.1(a) shows the inter-block distribution of $I(V_{trip} - V_{read})$ at the fast corner.

In order to observe the effects of variation induced failures in the memory on the performance of the MBC framework, a discrete cosine transform (DCT) was mapped to the MBC framework. Details of the mapping procedure is described in (149). LUTs for the DCT operation were first randomly mapped to the blocks of our memory model. Relative vulnerability to parametric failures for these blocks were determined through Monte Carlo simulations considering the systematic and random intra-die components mentioned above. For the distribution of process parameters obtained after Monte Carlo simulations, a reliability map of the memory blocks is generated on the basis of their $I(V_{trip} - V_{read})$ values. This reliability map serves as an input to the proposed variability-aware preferential mapping approach. Note that:

- At lower supply voltages, under parametric variation, the failure probability for the SRAM cell increases significantly (70). However, the relative vulnerability of the blocks due to systematic variation remains unchanged.
- Due to its input dependence, an average of the PSNR at the DCT output over a length of time fails to capture the worst-case scenario. A worst-case PSNR for multiple input vectors is therefore a better metric than a simple *time average*. However, due to the probabilistic nature of the parametric failures, estimation of true worst-case degradation is infeasible. An average over multiple instances of the same memory design yields the average

8.1 Mitigating the effect of parametric failures in MBC frameworks

Figure 8.2: (a) Original image at input of DCT; (b) Image after inverse DCT, where 1D-DCT operation is performed using the MBC framework. ($Avg_{PSNR_{Rwst}} = 28.49dB$).

worst-case PSNR ($Avg_{PSNR_{Rwst}}$) which serves as a comparison metric in our analysis.

Fig. 8.2 shows the output image quality ($Avg_{PSNR_{Rwst}}$) for random mapping of the LUTs to the memory blocks. By random mapping, we mean that the critical and non-critical computations of the target applications are mapped to memory blocks without considering their relative vulnerability to parametric failures. The final image is obtained by taking IDCT of the DCT components obtained as outputs from the MBC framework. Under variation, due to read failures in the LUTs, these DCT components will be different from the values obtained from the MBC framework in a no-variation scenario. As shown in Fig. 8.2(b), due to random assignment, the final image quality suffers considerable degradation with an average worst-case PSNR of 28.49dB.

8.1.2 A variation-aware preferential design approach

To alleviate degradation in output quality due to variation induced memory failures in the MBC framework, we propose a three-step solution that involves joint

Figure 8.3: Major steps in the proposed methodology for robust and low-power operation in MBC.

circuit/architecture level optimizations. As we show later, the proposed approach can provide significant improvement in parametric yield under variations. The approach can also be used for power reduction using aggressive voltage scaling. As summarized in Fig. 8.3, the major steps in the proposed co-design approach are:

8.1 Mitigating the effect of parametric failures in MBC frameworks

- A preferential (skewed) design of memory cell using transistor sizing that increases the read stability at the expense of write failures (which can be compensated during occasional reconfiguration using several low-cost techniques such as lower cell supply during write) for the read-dominated MBC framework.
- Post-fabrication characterization of the memory and generation of the reliability map to store $I(V_{trip} - V_{read})$, $I(T_{write})$ and $I(T_{access})$ values for the memory blocks.
- A preferential application mapping approach that maps the critical computations to more reliable memory blocks under delay constraint.

8.1.3 Preferential mapping

Many signal processing applications (such as DCT, FIR filtering) consist of critical and less critical computations (152). A mapping algorithm which: a) partitions the computations into critical and less-critical bins; and then b) maps the critical computations to more reliable sections of the memory can achieve a substantial performance improvement over a random mapping approach.

8.1.4 Post-Si generation of Reliability Map

A number of techniques (153),(150),(151) have already been proposed to generate post-Si reliability map for embedded memories in order to cope with device variation induced parametric yield loss. These techniques either use March Test (99) to detect parametric failures or attempt to directly measure the read/write margin for memory cells (151). The trade-offs between these two alternative approaches are provided in (149). Since the motivation for reliability map generation is only to identify the relative variability of the memory blocks, an array based technique can be used for fast characterization of the memory blocks. With array-based test approaches, it is possible to determine the relative variability of

8.1 Mitigating the effect of parametric failures in MBC frameworks

the memory blocks on the same die. The reliability map used for preferential application mapping is stored on a non-volatile memory which is loaded with the relative reliability values of the memory blocks during manufacturing test. If an array based characterization is pursued for map generation with block size of 1KB, the total memory requirement to store the reliability map for a function table of size 16KB is only 64 bits. This follows from the observation that the $I(V_{trip} - V_{read})$ values can be encoded into 4 bits without significant quantization error (refer to Fig. 8.1(a)).

8.1.5 Impact of Preferential Mapping on Output Quality

We propose a heuristic-based preferential mapping approach that can substantially improve the output quality of the target application under parameter variation. If $c_i (i=1 \dots P)$ denotes the contribution of the i^{th} computation to the overall output quality and $r_j (j = 1 \dots N, N \geq P)$ denotes the reliability measure of the j^{th} memory block, then the mapping heuristic should attempt to maximize $\sum_{i=1}^P c_i r_j$. By reliability measure for the j^{th} memory block, we mean $r_j = \frac{1}{I_j(x)}$. A simple minded approach is to sort $C = \{c_i\}$ and $R = \{r_j\}$ in descending order of their values, and then to assign the computations in the order of their contribution to unassigned memory blocks with highest value of r_j .

As an example, in the DCT application, first adder stage and the precomputer blocks are assigned to minimum $I(V_{trip} - V_{read})$ blocks for more reliable operation. Since probability of read failure (P_{RF}) dominates over the other parametric failure mechanisms at low frequencies of operation, assignment of blocks have been based upon $I(V_{trip} - V_{read})$. However, the same can be achieved based on $I(T_{access})$, $I(T_{write})$ or on a weighted combination of all. Fig. 8.4(a) shows the memory block assignment using the preferential mapping approach. The unassigned memory blocks are highlighted in black. A comparison with Fig. 8.1(a) clearly shows that blocks with high $I(V_{trip} - V_{read})$ have been avoided through the proposed mapping approach. The output as shown in Fig. 8.4(b) achieves about 4dB improvement in PSNR over a random mapping policy.

8.1 Mitigating the effect of parametric failures in MBC frameworks

Figure 8.4: (a) Preferential mapping of DCT modules to memory blocks (b) Image after inverse DCT, where 1D-DCT operation is performed using the MBC framework. ($Avg_{PSNR_{Rwst}} = 32.48dB$).

8.1.6 Preferential Mapping Algorithm Under Delay Constraint

A preferential mapping approach which does not consider any delay constraint would attempt to provide the best output QoS for a given variation model. However, this may cause increased delay overhead due to larger spacing between the computing elements in the critical path. Thus, in order to achieve the optimal mapping (in terms of reliability) under a delay constraint, resource allocation should be a part of the placement and routing step. We have developed a heuristic-based post-processing of the delay optimal routed netlist that maximizes QoS without violating the delay constraint. Fig. 8.5 shows the timing-driven variability-aware preferential mapping process. It starts with ranking the computations in order of their criticality and the memory blocks in order of their reliability measures (defined as r_i for the i^{th} memory block). In descending order of criticality, each computation in the routed design is then considered for reassignment if it is mapped to a memory block with unacceptable r_i . The reassignment step first looks for an unused block with higher r_i and at the smallest Euclidean distance d_i from its current position. The impact on timing is calculated for each trial using incremental timing analysis. If the resulting placement

8.1 Mitigating the effect of parametric failures in MBC frameworks

Figure 8.5: Flowchart showing the major steps for the proposed heuristic-based preferential mapping approach under timing constraint.

exceeds the delay constraint, we attempt to swap the current computation with a less critical one which is assigned to a more reliable memory block. As before,

8.1 Mitigating the effect of parametric failures in MBC frameworks

Figure 8.6: a) Delay optimal placement of the DCT blocks obtained from VPR (63); b) $I(V_{trip} - V_{read})$ for the memory based computing elements used to map the DCT application; c) Final placement after the proposed preferential mapping under timing constraint; d) Final image after proposed mapping has a PSNR of 31.72dB.

computing elements at the smallest Euclidean distance are given higher priority and timing for the new placement is checked so that it does not violate the delay constraint. The major steps of the proposed heuristic is illustrated in Fig. 8.5.

Reliability map for the entire memory array as obtained from Monte Carlo simulations is shown in Fig. 8.6(b). With this reliability distribution, PSNR degradation at the DCT output was measured for the preferential mapping approach. Fig. 8.6(a) shows the original routed design from VPR and the output from the timing-driven preferential mapping heuristic presented in Fig. 8.5(c). The modified placement as shown in Fig. 8.6(c) has the same delay (4.78ns) as the delay-optimal placement shown in Fig. 8.6(a). However, the output PSNR

8.1 Mitigating the effect of parametric failures in MBC frameworks

Figure 8.7: Variation in failure probabilities with a) W_{PUP} , b) W_{PDN} and c) W_{AX} ; d) Proposed SRAM cell sizing methodology.

improves from a value of $28.49dB$ to $31.72dB$.

8.1.7 Preferential memory design

In a SRAM cell design, optimization of read and write stability present contradictory requirements. However, we can leverage on the read-dominated access pattern in the MBC framework to trade off read stability with write-ability of memory cells. Stability of occasional write operation in MBC during reconfiguration can be improved by dynamic change in operating conditions. Although a number of novel memory schemes have been proposed to improve the read stability of the SRAM cell (130), (154), in this paper we explore a sizing based approach which incurs negligible design overhead.

8.1.8 Preferential sizing of 6-T SRAM

The criteria followed for preferential sizing of the 6-T SRAM cell are: i) Minimize the read and access failure probabilities; ii) Minimize the increase in area of

8.1 Mitigating the effect of parametric failures in MBC frameworks

the memory cell. The cell area estimation methodology is the same as followed in (70). Fig. 8.7 shows the dependence of the read, write and access failure probabilities on the width of the pull up (W_{PUP}), pull down (W_{PDN}) and access (W_{AX}) transistors. Details of calculation for the various probabilities are provided in (149). From Fig. 8.7(a-c) we note that:

- Increasing W_{PUP} increases V_{trip} and reduces P_{RF} .
- Increasing W_{PDN} decreases V_{read} and thus reduces P_{RF} .
- Decreasing W_{AX} reduces P_{RF} .
- Increase in P_{AF} due to weakening of W_{AX} can be compensated by increasing W_{PDN} .
- First two optimizations increase T_{write} and P_{WF} .

The nominal memory cell was skewed based on the above observations following the methodology as outlined in Fig. 8.7(d). The old and new value for the transistor sizes are given in Table 8.1. With $L_{min} = 45\text{nm}$, the increase in area per cell was calculated to be 2.5% according to the formula provided in (70). Fig. 8.8 (a)-(c) show the distributions of V_{read} , T_{write} and T_{access} for the nominal and the skewed cell.

Table 8.1: Transistor widths for nominal and skewed memory cell

Cell	$W_{PUP}(\text{nm})$	$W_{AX}(\text{nm})$	$W_{PDN}(\text{nm})$
Nominal	100	150	200
Skewed	125	125	220

8.1 Mitigating the effect of parametric failures in MBC frameworks

Figure 8.8: Distribution of: a) $V_{read} - V_{trip}$, b) T_{write} and c) T_{access} for the nominal and the skewed memory cell. d) Shows the T_{write} distribution for cell with column-based lowering of cell supply voltage.

8.1.9 Compensation for Increased Write Failure Probability

From Fig. 8.8(a) and Fig. 8.8(c), we note that the skewed memory design achieves better tolerance to read and access failures at the cost of higher write failures (refer

Table 8.2: Cell failure probabilities for $V_{ddnom} = 0.9V$

Cell	P_{RF}	P_{WF}	P_{AF}	P_F
Nominal	3.28e-8	6.66e-16	2.92e-14	3.28e-8
Skewed	2.93e-10	2.77e-12	2.22e-14	2.95e-10
Skewed w/ lower WR Vdd	2.93e-10	3.33e-16	2.22e-14	2.93e-10

8.1 Mitigating the effect of parametric failures in MBC frameworks

to Fig. 8.8(b)). Since a higher wordline voltage degrades the half-select stability of the cells in the same row (155), we propose to use a column based lowering of cell supply to achieve better write-ability of the skewed memory cell. Note that such column based dynamic control of the supply voltage has been explored in earlier works (156) to improve both read and write stability while minimizing leakage power. Although such a scheme incurs area overhead due to complicated power grid and routing, it achieves nearly a 10X reduction in single bit fails by creating a 100mV voltage differential between the wordline and cell supply. In our simulation framework, supply voltage for the memory cell was reduced to 0.7V to improve T_{write} (Fig. 8.8(d)). Table 8.2 shows the overall failure rate for the nominal design, the skewed design as well as skewed design with lower cell supply voltage for write. A skewed design with lower cell supply for write reduces the total cell failure probability (P_F) by factor of 111.94.

8.1.10 Impact of cell sizing on output quality

Fig. 8.9(a) illustrates the fact that skewing leads to redistribution of the cells in the 5 bins for a given memory block. Since a heavy penalty is associated with each failing cell, skewing leads to an improvement of $I(V_{trip} - V_{read})$ values across all the blocks in the memory (Fig. 8.9(b)). Fig. 8.9(c)-(g), illustrates the PSNR improvement achieved through the proposed preferential mapping and memory design. Following points may be noted from Fig. 8.9(c)-(g).

- Fast and slow corners in Fig. 8.9(d) and (e) correspond to global fast and slow corners, i.e. for both NMOS and PMOS. A PSNR value of 40dB is used to represent a *no-degradation* case, although the PSNR is theoretically ∞ for no degradation.
- Preferential mapping for the original memory can achieve a significant improvement in PSNR for a range of $I(V_{trip} - V_{read})$ as high as 1.75. Considering that a cell moving from bin #1 to bin #2 reduces $I(V_{trip} - V_{read})$ by 0.25, this is equivalent to tolerating 5 more read failures in the memory block without increasing N_{RC} .

8.1 Mitigating the effect of parametric failures in MBC frameworks

Figure 8.9: a) Redistribution of memory cells into bins before and after skewing; b) Skewed design achieves better $I(V_{trip} - V_{read})$ for all blocks; PSNR values for i) original without mapping, ii) original with mapping, iii) skewed without mapping and iv) skewed with mapping for c) fast, d) nominal, e) slow, f) fast-N slow-P and g) fast-P slow-N corners. h) Most blocks have moved to a similar $I(V_{trip} - V_{read})$ region after skewing.

8.2 Mitigating the effect of runtime-failures in MBC frameworks

- Due to significantly smaller values of $I(V_{trip} - V_{read})$, the skewed design did not suffer from any degradation in output quality in our simulations. However, if the tolerance for $I(V_{trip} - V_{read})_{max}$ is reduced to smaller value (in other words considering a higher variation for the skewed design), a degradation of the PSNR values is observed at the output (Fig. 8.9(c)-(g)). Preferential mapping can again be applied to the latter case to improve performance.
- In the fast-N slow-P corner, the memory cell becomes most susceptible to read failures. This is indicated by a higher value of $I(V_{trip} - V_{read})$ which the system must tolerate to produce an acceptable PSNR at the output.
- Improvement in output PSNR due to preferential mapping after skewing is comparatively smaller than the mapping applied to the original design. The reason for this is evident from Fig. 8.9(h) which shows that skewing leads to large improvement in read stability for all memory cells, thereby reducing improvement with the mapping approach. The skewed memory cells however occupy larger area compared to the nominal design and experience increased probability of write failures. The preferential mapping approach therefore can provide either a software-only or a complementary solution to circuit-level optimization during design to tolerate the effect of parameter variations.

8.2 Mitigating the effect of runtime-failures in MBC frameworks

Since memory based computing using on-chip cache can be used to improve processor reliability, it is essential to protect the cache from both i) manufacturing defects as well as ii) runtime failures. In order to compensate for manufacturing

8.2 Mitigating the effect of runtime-failures in MBC frameworks

defects and variation induced failures, statistical design along with built-in redundancy have emerged as a popular design choice (70). In this section, we try to address the runtime failures in on-chip caches induced by “soft error”, voltage, or thermal noise and aging effects. Conventionally such runtime errors are addressed using single error correction double error detection (SECDED) codes. However, these codes have very limited correction capability, making them inefficient to protect memory in scaled technologies (sub-45 nm), which are particularly vulnerable to multiple-bit failures (157), (158). The requirement to tolerate multi-bit failures is accentuated by inter-die and intra-die variation in memory blocks which increases the vulnerability towards runtime failures.

8.2.1 Impact of parameter variation on runtime failures

Runtime failures in processor cache can be classified as: a) random failures due to voltage/thermal noise and b) contiguous failures due to soft error. Although the Soft Error Rate (SER) of an SRAM cell is found to remain constant across process technologies, due to the increase in the memory size per generation of the chip, the total number of soft errors increases with processor generation (159), (157). Moreover, as the SRAM footprint decreases with process scaling, the probability of MBUs due to a single particle strike increases (157),(158). Runtime failures in processor caches have been conventionally addressed by ECC. The level of ECC protection is typically different for L1 and L2/L3 caches due to their different design criteria.

Due to process parameter variations, memory cells which are marginally functional during manufacturing test can undergo runtime failures due to voltage or thermal noise, soft-error or aging effects. These cells (referred to as “weak” cells (160), (161) must therefore be protected using more ECC compared to cells which have suffered little variations. As argued in (162), memory cells which have either moved to the low threshold (V_t) or the high V_t region due to inter/intra-die variations are susceptible to read, write, access and hold failures. We have observed a similar trend in our simulations (refer to (49)).

8.2 Mitigating the effect of runtime-failures in MBC frameworks

Figure 8.10: a) Based on variability, the spectrum of NMOS and PMOS variation is divided into multiple ECC regions. b) ECC allocation for memory blocks on a die at worst-case inter-die corner.

Based on the above analysis, we have noted that a $\pm 10\% \Delta V_t$ variation from the nominal value is sufficient to cause the cell to be more vulnerable to one or more modes of runtime failures. As a result a uniform ECC allocation performed statically for all memory blocks during design time fails to account for the distribution of vulnerability across memory blocks after fabrication. Moreover, it can lead to overly pessimistic results if the worst-case vulnerability of a memory block is accounted for during ECC allocation. In this paper, we therefore propose a reliability-driven ECC allocation scheme that matches the relative vulnerability of a memory block (determined using post-fabrication characterization) with appropriate ECC protection. As shown in Fig. 8.10(a), the entire spectrum for PMOS and NMOS V_t variation in a SRAM cell can be divided into 3 regions based on the extent of ECC protection they should receive. We note that

8.2 Mitigating the effect of runtime-failures in MBC frameworks

Figure 8.11: (a) Major steps in variable ECC allocation; (b) Architecture for post-fabrication variable ECC allocation based on the process corner of the individual memory blocks.

a strong PMOS and a nominal NMOS ($\Delta V_t < +/ - 10\%$ of the nominal value) is most resilient to the failure mechanisms considered above. Cells with such configuration should therefore be assigned the least ECC protection ($t = t_{min}$ where t denotes the number of errors that may be corrected for a given block). Cells with PMOS and NMOS V_t variation less than $+/- 10\%$ of the nominal value should be assigned a higher ECC protection ($t = t_{med}$) due to their vulnerability towards soft error. Cells falling outside the above ΔV_t regions should be assigned the maximum ECC protection ($t = t_{max}$) to tolerate runtime failures arising from system noise, soft errors and aging effects.

8.2.2 Reliability-driven ECC allocation for multiple bit error resilience

8.2.2.1 Overall scheme

Fig. 8.11(a) shows the major steps of the proposed variability aware ECC allocation methodology in large caches. As seen from Fig. 8.11(a), the cache size (S), cache associativity (A), block size (B) and the maximum number of bit-errors that can be tolerated (t_{max}) serve as input specifications for the variable ECC allocation methodology. During the design phase, based on t_{max} , the number of required check bits (m) and the number of ways (w) required to store m check bits is calculated. Suitable μ -arch modifications to interchangeably use these ‘w’ ways for data and ECC storage are then incorporated into the cache. Finally, a low-latency and energy-efficient BCH encoder/decoder is implemented at the memory interface to detect and correct up to t_{max} bit-errors. Post-fabrication during manufacturing test, a reliability map is generated which characterizes the vulnerability of different sections of the cache to runtime errors. The baseline vulnerability is determined by the amount of parametric variation (measured in terms of read/write margins) present in the most reliable memory block. The baseline vulnerability is used to determine the value of t_{min} for the proposed ECC allocation scheme. For example in our simulations, for a bit failure probability of 10^{-6} , most reliable memory blocks have been assigned $t_{min} = 2$. Based on the reliability map, values of $t_{min} < t \leq t_{max}$ for other memory blocks are determined by their relative vulnerabilities to parametric failures. For variable ECC allocation during deployment, these t values are encoded into 2-bits and are stored on-chip in a small non-volatile memory. Based on these encoded values, variable number of ways in each set are dedicated to storage of ECC bits by asserting appropriate select signals to the ECC selection logic. During runtime, the ECC selection logic (present for each memory block) is responsible for distinguishing the data and ECC bits for each set being read.

Fig. 8.11(b) shows the corresponding L2 cache architecture to enable the proposed variable ECC allocation. For a given error rate, let the three levels of

8.2 Mitigating the effect of runtime-failures in MBC frameworks

ECC protection assigned to a L2 cache be $t_{min} = 1$, $t_{med} = 2$ and $t_{max} = 3$. $t = t_{min} = 1$ denotes the scenario where a single error in a memory word (32-b or 64-b) can be safely corrected using conventional SECDED scheme. Typically the SECDED protection is applied to the L2 data arrays while runtime failures in the smaller tag arrays are typically avoided by either conservative design of the tag array or simple parity assignment (163). Assuming 8 ECC bits per 64-b of L2 cache, SECDED scheme for 2MB L2 cache will require a storage of 256KB. Conventionally a separate storage is dedicated for storing these ECC bits on-chip (164). In the variable ECC assignment scheme, multiple-bit error correction facility in less reliable memory blocks is achieved using BCH codes. For example, $t = 2$ denotes a 2-b error correction and 4-b error detection scheme. Similarly $t = 3$ denotes 3-b error correction and a 6-b error detection scheme, all achieved using BCH codes. Higher values of t require extra ECC bits. For example, for each 64-b word, the number of ECC bits required for $t = 2$ and $t = 3$ are 14 and 21, respectively. In order to accommodate these extra ECC bits, we use one or more of the cache ways to store ECC bits in place of actual data bits. This is illustrated in Fig. 8.11(b). The derivation for the number of ways which needs to be devoted for storing the ECC bits is provided in (49).

The proposed scheme exploits the fact that modern L2/L3 caches typically have large number of “ways”, some of which can be used to store check bits for on-demand error tolerance. Thus, contrary to the 2-D error coding scheme proposed in (165), allocating one or more ways in the same row for ECC storage does not cause unnecessary read from multiple rows. Fig. 8.11 shows the encoder and decoder circuits for multiple-bit ECC protection. The encoder logic is used during the process of write (or store) to the L2 cache. At all other times, encoder logic corresponding to $t = 1, 2$ and 3 are gated off. Similar to an only SECDED scheme, any store shorter than the ECC word (64-b) requires a read-modify-write operation to the ECC stored in the way as well as to the ECC bits in the memory conventionally used for storing SECDED.

Decoding for error correction involves two components: i) logic for detection of error and ii) finding out the error location in order to achieve error correction. For

BCH encoding, the first component is denoted as *Syndrome detect*. The second component is further subdivided into *Berlekamp-Massey (BM) module* and *Chien Search module*. The correction circuit is invoked only if an error is detected. For realistic error rates, error-free read will be performed most of the time. Since the detection logic appears in the critical path for L2 read, we have implemented a single-cycle decoding logic for BCH. Moreover, the error correction logic is pipelined allowing a 64-b throughput per clock cycle. Details of this architecture which allows a post-fabrication ECC allocation is provided in (49).

8.2.2.2 μ -architectural modifications for variable ECC allocation

Fig. 8.12 shows the modifications to the L2 cache μ -architecture required to realize the variable ECC assignment. In addition to the address tag, an enable signal from the reliability map would now serve as an input to the *Address Tag Compare* logic (Fig. 8.12). Given the memory blocks are classified into 4 or less ECC bins, this signal would be only 2-b wide. This actually serves to invalidate the tag comparison operation for one or more ECC ways so that during read, data out is not selected from any of these ways. If there is a hit in any of the remaining data ways, the cache hit signal and the enable signal coming from

Figure 8.12: μ -architecture showing procedure to store and retrieve ECC bits from one or more cache ways.

Figure 8.13: States of the L2 ECC controller for error detection and correction.

reliability map can be used to separate the ECC bits from the data bits using an *ECC selection logic* as shown in Fig. 8.12. During write, the ECC bits stored in these ways is modified based on the incoming data to be written. Write occurs to the cache way for which the tag hit has occurred. In evaluating the performance for variable ECC, we note the number of runtime errors in the L2 cache that are either transferred to the processor pipeline during loads or are transferred to the main memory in case of a read-miss or a write-miss. Fig. 8.13 shows the L2 states during which the encoder, error detection and correction logic are invoked. Here we assume a write-back L2 with a write-allocate scheme. Details of the controller states is provided in (49).

8.2.3 Simulation setup and results

The effectiveness of the proposed ECC allocation scheme was validated for the following scenarios: 1) tolerance to random failures; 2) tolerance to contiguous failures; 3) simultaneously tolerating both random and contiguous failures; and 4) tolerating high failure rates at low voltage. Details of our simulation setup are provided in (49). For evaluating the tolerance to runtime random errors, a weighted random pattern generation tool was integrated with the SimpleScalar setup to insert random runtime errors into the L2 cache following a given bit failure probability (BFP). A BFP of 10^{-n} indicates a single bit failure out of 10^n bits. Considering a wide range of device parameter variations for PTM 45nm model, a large BFP range ($10^{-7} - 10^{-4}$) was observed in our SPICE simulations. We note that a similar range has been observed in (166). In our simulations, the Mean Time To Failure (MTTF) is taken to be 10^5 cycles. Hence, the error pattern is regenerated every 10^5 cycles. Note that the choice of MTTF has been made to facilitate the simulation process. A smaller or larger value of MTTF would not affect the average number of failures in a simulation window and hence is not expected to affect the choice of ECC or the energy/performance trends. The total number of errors in L2 which propagates into the processor pipeline or to the main memory was noted at the end of simulation. Energy values for the baseline processor and the proposed variable ECC scheme is obtained for 100nm CMOS process using Wattch (147).

8.2.3.1 Tolerance to random errors for $BFP = 10^{-4}$

Although we simulated for a wide range of BFP , here we present the results for $BFP = 10^{-4}$. Simulation results for other values of BFP are provided in (49). The baseline processor suffers from significant number of runtime failures for both worst as well as best-case reliability maps (refer to Fig. 8.14). For the proposed scheme, a $t = 2, 3$ and 4 distribution is able to correct 99.9% and 99.8% of all the failures that occur in the baseline processor. However, as shown in Fig. 8.14, it fails to account for all the errors. In order to ensure an error free operation, all

8.2 Mitigating the effect of runtime-failures in MBC frameworks

Figure 8.14: Number of failures in a) Baseline and b) Proposed Scheme with $BFP = 10^{-4}$.

the memory blocks must be protected with $t_{min} = 4$ for the die with the worst-case and $t_{min} = 3$ for the die with best-case reliability maps respectively. When the proposed scheme with $t_{min} = 4$ is used to protect all memory blocks in a die with worst reliability map, the average increase in CPI and energy are 3.58% and 3.8% respectively. For the best reliability map, the CPI and energy increase are 2.47% and 2.53% respectively (refer to Fig. 8.15). Average increase in L2 miss rate are 17.14% and 17.12% respectively.

8.2.4 Tolerance to soft-error induced contiguous errors

For protection against soft error induced failures, we assume that the baseline processor is protected using SECDED along with 4-way bit interleaving. Such a system can correct up to 4 contiguous errors and detect up to 8. This suffices to cover the maximum number soft-error induced contiguous failures (4, 5 or 7) corresponding to $N_c = 16, 32$ or 64 , where N_c refers to the number of memory cells between two well taps (158). However, as pointed out in (165), bit interleaving comes at additional power, delay and area overhead. Power overhead in an

8.2 Mitigating the effect of runtime-failures in MBC frameworks

Figure 8.15: Percentage increase in a) CPI and b) Energy for the proposed scheme with $\text{BFP} = 10^{-4}$.

interleaved cache, which occurs primarily due to access of unwanted words in the same row has been shown to be more than 100% of a non-interleaved one (165). It is therefore interesting to see the trade-off involved in tolerating multiple-bit errors using on-chip BCH based ECC as opposed to bit-interleaving.

For memory designs with $N_c = 16$ or 32 , an ECC distribution with $t = 4, 5$ and 6 suffices to cover the maximum number of contiguous errors that may occur. Penalty is however paid in terms of higher conflict miss and increased CPI overhead since $2 - 3$ ways in each set are used for storing the ECC instead of data. Fig. 8.16 shows the CPI overhead for the individual benchmarks for worst and best-case reliability maps. The average CPI overhead in these two cases are 3.52% and 3.29% , respectively. Although the L2 miss rate increases significantly, the variable ECC allocation actually leads to energy savings over the baseline processor which employs bit interleaving for tolerating multiple-bit contiguous errors. Total energy consumption for the baseline processor was estimated assuming the fact that bit interleaving doubles the energy requirement for each L2 access (165). For the proposed scheme without bit interleaving, energy calculation considers the overhead due to $t = 4, 5$ and 6 ECC logic. Fig. 8.16(b)

Figure 8.16: Percentage increase in a) CPI and b) Energy when the proposed scheme is used for soft-error tolerance with $t_{min} = 4$.

shows the energy overhead for the individual benchmarks. Average savings for the worst and best maps are 4.33% and 4.13%.

8.2.5 A unified solution to tolerate random and contiguous errors

Here we discuss the trade-offs involved with schemes that can serve as unified solutions to tolerating both random as well as contiguous errors in memory words.

8.2.5.1 $t_{min} = 4, 5$ and 6 distribution with extra storage

Variable ECC allocation with $t = 4, 5$ and 6 distribution leads to inordinate increase in L2 miss rate for a set of benchmarks. This can be addressed if additional storage is available for storing the extra parity bits required by BCH encoding rather than using one or more ways to store the ECC bits. Assuming an additional storage for SECDED is already present, storage required for storing 20 extra parity bits corresponding to $t = 4$ is calculated to be 31% of the L2 area.

Figure 8.17: Overhead in: a) CPI and b) energy when the proposed scheme with $t_{min} = 2$ is used along with bit-interleaving for soft-error tolerance.

For blocks with $t = 5$ and $t = 6$, we assume that the additional parity bits occupy 1 out of 8 ways in L2. This area increase is with respect to a L2 cache without bit interleaving scheme. If this area overhead can be tolerated then, the average L2 miss rate increase is a minimal 3.68%. After accounting for this additional energy overhead, a $t = 4, 5$ and 6 distribution with extra memory leads to an average energy savings of 6.64%. Results for individual benchmarks are shown in Fig. 8.17(b).

8.2.5.2 $t_{min} = 2, 3$ and 4 distribution with bit-interleaving

A variable ECC allocation with $t = 2, 3$ and 4 and 4 way bit interleaving tolerates both: i) random errors with error rate as high as 10^{-4} ; ii) up to 8-bit contiguous errors in $t = 2$ blocks and up to 16-bit contiguous errors in $t = 4$ blocks, making it amenable to tolerance of high soft error rate and scalable to future technology nodes. With respect to a L2 cache without bit interleaving, this scheme does not incur any extra area overhead since extra ECC bits for $t = 2, 3$ and 4 are

Figure 8.18: Percentage energy savings at 100nm and 70nm technology nodes when the on-chip L2 cache is operated in a low power mode with the variable ECC scheme accounting for the increased failure rate in low power mode.

stored in ways. The average CPI overhead is only 1.56% (refer to Fig. 8.17(a)). With respect to an only bit-interleaved L2, this scheme incurs an average 1.57% increase in energy requirement (refer to Fig. 8.17). The average L2 miss rate however increases by 11.64% .

8.2.6 Energy saving with dynamic voltage scaling

The proposed multi-bit error tolerance scheme provides an opportunity to dynamically allocate ECC for the memory blocks. Such dynamic adaptation can be used to achieve power saving by combining voltage scaling for the memory cells with higher ECC protection. A 200mV reduction in supply voltage reduces the read, write and leakage powers by 55.86% , 33.15% and 55.7% , respectively.

This energy saving due to low power operation outweighs the overhead in providing extra ECC protection for $BFP = 10^{-4}$. Using Cacti we first estimate the ratio of leakage to dynamic power for a 2MB cache. For 100nm node this ratio is 1.03 . The total energy consumption including the leakage overhead for the L2 is then estimated for both nominal as well as low power operating conditions. After considering the energy overhead for $t = 4$ ECC protection for all memory blocks, average energy savings at 100nm node with the proposed scheme is calculated to be 5.93% (Fig. 8.18). Higher savings in energy is achieved for 70nm technology

8.2 Mitigating the effect of runtime-failures in MBC frameworks

node. At 70nm node, savings in read, write and leakage energy increases to 64.76%, 48.77% and 62.8%, respectively. The ratio of leakage to dynamic energy also increases to 5.17. Taking into account all these factors, the average energy savings at 70nm is 20.58%. Note that in order to provide increased protection during low power operation, the cache must go through a number of transitions before it fully operates in the low power mode. These actions ensure that correct ECC bits are stored for the memory blocks. Details of this transition is provided in (49).

Off-chip hardware acceleration using MAHA framework

Since MBC model is independent of the underlying memory technology, MAHA can be realized using both on-chip (such as SRAM or embedded DRAM) as well as off-chip memory (such as Flash or emerging memory technologies such as STTRAM (75), RRAM (77) etc) technologies. Although the actual implementation may vary from one technology to another, the working principle will remain the same. For both on-chip and off-chip scenarios, MAHA is intended to function as a hardware accelerator. This implies compute or data-intensive tasks or both for which software execution is inefficient in terms of performance and/or energy would be migrated to the MAHA framework. A large number of hardware accelerators for domain specific acceleration (24) as well as acceleration of general purpose applications (21) have been proposed to date. All these accelerators however suffer from the same Von-Neumann bottleneck as faced by the conventional CPU. In this chapter we demonstrate the use of MAHA as a hardware accelerator which improves the energy-efficiency for diverse applications by migrating the task to the memory. In summary, the key contributions are as follows:

- Application characteristics are identified which define the condition under which an application will benefit in energy-efficiency using MAHA based off-chip hardware acceleration.
- Hardware modifications to the conventional CPU system in order to realize the MAHA framework using off-chip non-volatile memory are proposed. The memory technology considered for this purpose is CMOS-compatible NAND Flash memory. Design considerations specific to NAND Flash technology are also investigated.
- Modifications to the conventional CPU based software flow to allow application execution using off-chip MAHA framework are proposed.
- Applications from diverse application domains were successfully mapped to the off-chip MAHA framework and the improvement in energy-efficiency was noted for each.

9.1 Background and motivation

In this section we re-emphasize on the performance and energy bottleneck of the Von-Neumann computing model, highlight current computing hardware models which aim at mitigating the above bottleneck and provide the motivation for an off-chip MAHA framework.

9.1.1 Von-Neumann bottleneck

The term “Von-Neumann bottleneck” famously coined by John Backus in his 1977 ACM Turing award lecture refers to the limited data transfer rate between the processor and the memory compared to the large amount of memory available (10). Since the inception of digital computer this bottleneck has been both a performance limiter and more importantly an energy bottleneck for the entire system. Since the transfer rate is not increasing accordingly to Moore’s law

(16), the relative time required to fetch a volume of data proportionate with the compute resources on-chip is increasing with every technology node (68).

9.1.1.1 Performance bottleneck

The performance bottleneck of the Von-Neumann model is a well-known fact. The disproportionate improvement in CPU speed and the memory size compared to the data transfer rate between the two puts a serious limitation on the effective processing speed when the CPU is required to perform minimal processing on large volume of data for data-intensive applications. To alleviate this performance bottleneck, multi-level caches and branch prediction algorithms are common feature for modern microprocessors (107). The situation is however expected to exacerbate further with technology scaling. While the total number of compute units is expected to increase by $\sim 15X$ from 2011 to 2022, the off-chip data access bandwidth is only expected to improve by $1.4X$ over the same time period (16). For a *aggressive strawman* system outlined in (68), the effective number of processor clock cycles required to execute a single instruction (t_{proc}) can be computed as :

$$\begin{aligned}
 t_{proc} &= \frac{1}{N_{issue}} \times (1 + g \times (L1_{lat} + (1 - L1_{hit}) \times L2_{lat} \\
 &\quad + (1 - L1_{hit}) \times (1 - L2_{hit}) \times localDRAM_{lat} \\
 &\quad + (1 - L1_{hit}) \times (1 - L2_{hit}) \times (1 - localDRAM_{hit}) \\
 &\quad \times globalDRAM_{lat})) \\
 &\text{or} \\
 t_{proc} &= \frac{1}{N_{issue}} \times (1 + g \times T_{mem})
 \end{aligned} \tag{9.1}$$

In equation 9.1, the number of cycles in an execution unit is assumed to 1. $L1_{hit}$, $L2_{hit}$, $localDRAM_{hit}$ and $globalDRAM_{hit}$ represent the hit percentage in L1, L2, local DRAM and global DRAM respectively. For these parameters, typical values of 0.8, 0.9, 0.6 and 0.4 obtained from (68) have been used in our

calculations. Similarly, $L1_{lat}$, $L2_{lat}$, $localDRAM_{lat}$ and $globalDRAM_{lat}$ stand for the access latencies of L1, L2, localDRAM and globalDRAM respectively. Typical values for these are 1, 8, 100 and 10,000 processor cycles respectively. For a given application, ‘ g ’ represents the fraction of instructions with memory reference. N_{issue} represents the number of instructions issued in a given clock cycle. On-chip caches mitigate the performance bottleneck to a large extent by exploiting the ‘spatial’ and ‘temporal’ locality of operands in a program. However, with other factors remaining constant, the increasing gap between number of compute elements on-chip and bandwidth to off-chip memory will degrade the off-chip memory access latency significantly. Novel packaging methodologies which enable 3D stacked memory chips with high-speed signalling can be potential solutions in such scenarios (68).

9.1.1.2 Energy bottleneck

Although the performance bottleneck of the Von-Neumann computing model is well-researched, implications of its energy bottleneck are being recently studied (68). The energy bottleneck primarily arises due to the movement of instruction and data back and forth between physically separate memory and compute engine. On-chip caches, though beneficial in masking the performance bottleneck, are major contributors to energy bottleneck (167). As rightly put in (168), with the current computing models, *“it is possible to hide the latency of off-chip access but not its energy”*. Similar to equation 9.1, the average energy per instruction for a single processor (e_{proc}) is given as:

$$\begin{aligned}
e_{proc} &= E_{compute} + g \times (E_{L1} + (1 - L1_{hit}) \times E_{L2} \\
&\quad + (1 - L1_{hit}) \times (1 - L2_{hit}) \times E_{localDRAM} \\
&\quad + (1 - L1_{hit}) \times (1 - L2_{hit}) \times (1 - localDRAM_{hit}) \\
&\quad \times E_{globalDRAM}) \\
&\text{or} \\
e_{proc} &= E_{compute} + g \times E_{mem}
\end{aligned} \tag{9.2}$$

In equation 9.2, $E_{compute}$ represents the energy expended in the processor due to instruction fetch, register file read and in the execution units. Typical values at 45nm are $15\text{-}20pJ$. E_{L1} and E_{L2} represent the L1 and L2 cache access energies, with typical value of $40pJ$ and $400pJ$. Similarly $E_{localDRAM}$ and $E_{globalDRAM}$ represent the access energies for local and global DRAM with typical values of $1380pJ$ and $13800pJ$. All these estimates were obtained from (68). One important point to note here is, the energy to actually read and write into the off-chip memory is very small ($\sim 15\%$) compared to the energy expended in the interconnect during the transport of data and instruction ($\sim 85\%$) (168). A computing framework which minimizes the movement of data is therefore expected to reap significant savings in total energy consumption.

9.1.2 Existing non-Von-Neumann architectures

The area of research which addresses performance and energy bottleneck arising from Von-Neumann computing model, boasts of a large number of novel and exciting solutions. Following are three main-stream solutions which have gained prominence over the past decade. These are: i) Graphics Processing Unit (GPU); ii) Reconfigurable processors and co-processors and iii) Processor-in-memory.

9.1.2.1 GPU

The advent of GPGPU (General-purpose computing on graphics processing units) has revolutionized the field of high-performance computing. GPUs which were conventionally used for computer graphics can be used to perform applications which were traditionally handled by the CPU. Key hardware concepts which empowers the GPU for high-performance computing are: i) addition of programmable stages and higher precision arithmetic to the rendering pipelines which allows software developers to use *stream processing* on non-graphics data; ii) support for large datasets, high parallelism with minimal dependency between data sets indirectly improves the arithmetic intensity and helps to hide the memory access latency. The GPU enjoys a very high memory bandwidth ($\sim 80GB/s$ with ~ 400 pin DRAM interface for *Nvidia Tesla* (169)) along with a tiered storage bandwidth hierarchy and algorithms running on the GPU take advantage of this bandwidth to achieve dramatic performance improvement. On the software side, development of programming frameworks like *CUDA* and *openCL* (170) have allowed the software programmers to exploit the hardware parallelism in the GPU to address the data and task-level parallelism in their applications. For the above reasons GPUs are now being commonly used in application domains ranging from digital image processing, video processing, scientific computing etc.

Although modern GPUs embrace the idea of *throughput computing* (171) to improve performance (Von-Neumann performance bottleneck) in the post voltage-scaling era, they fail to address the Von-Neumann energy bottleneck as described in the previous subsection. In the GPUs, best energy efficiency is achieved only for data-parallel applications which are heavy in floating point calculations. A survey for mobile GPUs reveals that the lowest TDP (thermal design power) for a single GPU core is as high as $12W$ (172). For applications which do not satisfy the above characteristics, energy efficiency drops prominently.

9.1.2.2 FPGA

Modern FPGAs integrate ~ 4 billion transistors (1.5 times the transistor count for a 10-core Intel Xeon processor) and the spatial computing model allows its performance to scale following Moore’s law. A large pool of compute resources comprising of LUT based configurable logic blocks and custom logic connected with an elaborate framework of programmable interconnects make FPGA amenable for mapping applications of varying complexity from diverse domains. Advancements in reconfigurable hardware design have been supplemented with software innovations (72, 109, 173) which allow application description in high-level languages to be directly translated to a hardware image. All these technological advancements have allowed FPGAs and similar reconfigurable frameworks to surpass the GPUs in both performance and energy-efficiency for a variety of application domains (100, 174). The common observation in these works is that FPGA is more performance and energy-efficient for applications which involve low level hardware control operations and can benefit from pipelining. GPUs on the other hand are beneficial for streaming applications with high demand for memory bandwidth and floating-point performance (100). Although FPGAs have been successful in mitigating the performance bottleneck associated with a single-CPU model, they still suffer from poor memory bandwidth (typically 20-30 GB/s as compared to ~ 160 GB/s for GTX GPU family from Nvidia) and Von-Neumann energy bottleneck. Although the power consumption for the FPGA depends on the actual application mapped, it is typically tens of watts (16).

9.1.2.3 PIM architectures

The perception that memory and I/O bandwidth and energy can be ultimate bottlenecks towards performance and energy scaling had led to the development of the concept *processing in memory* (74, 83). The main idea is to move the processing power to the proximity of the off-chip memory, thereby mitigating the Von-Neumann bottleneck. Different implementations of the same have however been proposed. *IRAM* or *Intelligent-RAM* (74) incorporates a vector processor

along with DRAM which takes advantage of the wide processor-DRAM interface to process large volume of data in a parallel and energy-efficient manner. Processing-in-memory or *PIM* as outlined in (82) proposes a tiled architecture where each tile consists of a simple CPU such as MIPS R3000 communicating with 4 smallest possible slices of DRAM. The *Gilgamesh* project (175) looks to extend the *PIM* computing paradigm to a massively parallel architecture for better scalability, adaptability and fault tolerance. *Computational RAM* (83) proposes to integrate fine-grained reconfigurable logic in close proximity of the DRAM array to on-demand transform the memory array into a SIMD computing machine for DSP applications. Other efforts (176) include incorporating an embedded DRAM into the conventional FPGA framework for minimizing reconfiguration latency and facilitate logic-memory bandwidth.

Most of the above investigations were performed in 90's. The primary reason of them being not a part of mainstream computing for over a decade is the sustained reduction in processor energy consumption ($\propto \alpha^3$, α being the scaling factor) for each technology generation. This has allowed the processor energy to be minimized to such a point that the total energy for the system is now dominated by the off-chip access energy. As a result experts (68) are speculating the return of disruptive computing paradigms such as *processing in memory* to virtually eliminate the performance and energy bottleneck due to off-chip memory access.

9.1.3 Why do we another computing framework?

With the plethora of computing fabrics existent today, the obvious question is whether we need a new one. The primary motivation for a new computing hardware is derived from a quantitative comparison of the energy-efficiency between the conventional uni-processor model, the frameworks described above and a computing framework that resides together with a non-volatile memory which is finally responsible for storage of data. A number of earlier works (16, 177, 178)

have analyzed the energy and performance benefits of emerging computing models which include symmetric multicores (e.g. Intel Corei7), heterogenous multicores (IBM Cell Processor) and heterogenous computing platforms integrating GPU/FPGA/ASIC along with a general-purpose processor (e.g. Chimaera (21)). These works however do not separately assess the impact of the memory hierarchy on the performance and energy of the total system. Here we therefore try to formulate the performance and energy for a memory based computing platform and identify key parameters which determine the domain of applications for which such non Von-Neumann computing machines would be of practical use.

To start with, we define the following important parameters:

g - fraction of total instructions with memory reference (loads and stores).

f - fraction of total instructions actually transferred to MBC.

c - fraction of instructions translated to actual instructions in memory based computing framework. Note that unwanted loads and stores are removed during a translation from the host processor's instruction set to the instruction set for the MBC framework. It is dependent on how the kernel is mapped to the MBC framework.

o - fraction of instructions in the memory based computing platform which result in an output which needs to be transferred to the host processor. We note that for applications such as Secure Hash Algorithm (SHA) or color histogram, this fraction can be negligible.

e_{mbc} - average energy per MBC instruction. Typical value of 50pJ has been used for further calculation.

e_{txfer} - energy expended in the transfer of an output from the MBC framework to the host processor. Typical value of 10,000 pJ has been used for calculation.

t_{mbc} - ratio of cycle time for the MBC framework to that for the host processor. Typical value of 15 is used for further calculation.

n - fraction speedup due to parallelism in the MBC framework. Typical value of 0.01 has been used assuming 100 computing elements in parallel.

t_{txfer} - time taken in terms of processor clock cycles to transfer an output from MBC to the host processor. Typical values are of the order of 10,000 processor

cycles.

With these definition, the average time to execute an instruction in a system with a host processor and the MBC framework can be formulated as:

$$T_{sys} = T_{mbc} + T_{proc} + T_{txfer} \quad (9.3)$$

where

$$\begin{aligned} T_{mbc} &= t_{mbc} \times (I_{f>g}(f) \times (f - g + f \times c \times n) \\ &\quad + I_{f \leq g}(f) \times f \times c \times n) \\ T_{proc} &= (1 - f) \times t_{proc} \\ T_{txfer} &= t_{txfer} \times (I_{f>g}(f) \times ((f - g) \times o + f \times c \times o) \\ &\quad + I_{f \leq g}(f) \times f \times c \times o) \end{aligned}$$

where

$$I_A(x) = \begin{cases} x & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

In equation 9.3, T_{mbc} , T_{proc} and T_{txfer} denote the fraction latencies in the MBC framework, due to processor execution and in the transfer of the resultant output from the MBC platform to the processor. As seen from equation 9.3, the first and the third terms increase with f , indicating the increase in both compute time in MBC as well the time required to transfer the resultant output to the host processor. The increase is however more prominent once the fraction of task f transferred to the off-chip compute platform is more than the fraction g dominated by memory references. The latency for the first component however reduces as the number of parallel computing units is increased (value of n reduces). The second component which denotes processor execution time decreases with increasing f since it has to share less burden of computation. A similar

Figure 9.1: Variation in total energy for a system with an off-chip MAHA framework for $g=0.7$ with c and o varying as a) 0.005; b) 0.05 and c) 0.5. d) Times improvement in energy compared to a single processor execution for varying values of g with $c=o=0.05$.

expression for the energy of the resultant system is given below:

$$E_{sys} = E_{mbc} + E_{proc} + E_{txfer} \quad (9.4)$$

where

$$E_{mbc} = e_{mbc} \times (I_{f>g}(f) \times (f - g + f \times c) + I_{f \leq g}(f) \times f \times c)$$

$$E_{proc} = (1 - f) \times e_{proc}$$

$$E_{txfer} = e_{txfer} \times (I_{f>g}(f) \times ((f - g) \times o + f \times c \times o) + I_{f \leq g}(f) \times f \times c \times o)$$

Similar to latency, it also has three components, with the MBC and the transfer

energy increasing with f while the processor energy reduces with increasing f . From, equation 9.3 and equation 9.4, it is evident that for appropriate values of g and other parameters, either the terms increasing with f or the one decreasing with f will dominate. It is however interesting to note that for a range of values of c and o , there exists an f at which minimum energy, latency and hence energy-delay product (EDP) can be achieved for the system. Fig. 9.1(a-c) shows the three components and the total system energy. For Fig. 9.1(a-c), $g=0.7$. In Fig. 9.1(a-c), the values of c and o have been taken to be 0.005 , 0.05 and 0.5 respectively. As clearly evident from Fig. 9.1(a-c), as the effective reduction of load and stores achieved by the MBC framework reduces (c increase) and the fraction of outputs to be transferred to the processor also increases (o increases), the total energy increases significantly. For smaller values of c and o , the energy savings however dominate over the other two components. For intermediate values of c and o , minimum energy consumption is achieved for f close to g . Fig. 9.1(d) shows the total energy saving (compared to the value in equation 9.2) as a function of f for different values of g with $c=o=0.05$. As we may note from the figure, the maximum energy saving reduces as g reduces and is observed at smaller values of f . Given the nature of an application (g and o known), knowledge of its mapping to the MBC framework (c and n known) and other parameters known, it is therefore possible to estimate the value of f for which maximum energy savings can be obtained by off-chip computing.

A similar trend is observed too in the total latency of executing the application. Fig. 9.2(a) shows the ratio of the effective number of cycles required to execute a single instruction for the MBC system to the cycles required in a single processor. As we may see from Fig. 9.2(a), for $c=o=0.05$, there exists a value of f when the system achieves maximum performance improvement over the baseline system. Similar trend was observed for $c=o=0.005$. However, for $c=o=0.5$, the transfer latency dominates the total latency, thus nullifying any performance improvement achieved through parallel speed in the MBC framework. Combining the performance and energy trends for the MBC system, we derive the behavior for energy-efficiency (expressed in terms of energy-delay product or EDP) of

Figure 9.2: a) Improvement in performance as g is varied with f for $c=o=0.05$ and $n=0.01$. Variation in energy-efficiency (EDP) for b) $c=0.005$, c) $c=0.05$ and d) $c=0.5$.

the total system. Fig. 9.2 shows the EDP of the total system for $c=o=0.005$ (Fig. 9.2(a)), 0.05 (Fig. 9.2(b)) and 0.5 (Fig. 9.2(a)). As expected, the EDP improvement progressively diminishes as the data-intensive nature (value of g) of the application is reduced. Finally, from Fig. 9.2(a) and (b) we note that for EDP improvement > 1 , for a given g , there exists an f for which maximum EDP improvement can be observed. In this work we show that there exists several data-intensive applications in different application domains which bear the characteristics mentioned above and the proposed memory based hardware re-configurable framework can be employed to achieve significant improvement in energy-efficiency for these set of applications.

Figure 9.3: a) Results from published works demonstrate scaling of NAND Flash access time with technology; b) Comparison of NAND Flash memory with emerging non-CMOS memory technologies.

9.1.3.1 Difference with existing frameworks

An off-chip MAHA framework based on a non-volatile memory technology is a completely different approach as pursued by previous hardware frameworks in mitigating the Von-Neumann bottleneck. The distinctions are as follows:

- *A computing framework in non-volatile memory:* Unlike conventional CPUs, GPUs and FPGAs integrate large amount of on-chip distributed memory and compute blocks. Since the data for such devices is fetched from the main memory, their performance will eventually be throttled by the off-chip I/O bandwidth which scales poorly compared to the logic integration density (16). *Processing-in-memory* systems mitigate that bottleneck to a large extent by migrating the application to be executed to DRAM based main memory. But main memory is volatile and data which is being processed only resides in the off-chip non-volatile storage. Movement of data from the non-volatile storage to main memory requires thousands of processor clock cycles and incurs significant energy overhead. It is easy to comprehend

that the natural transition to achieve the best performance and energy is to migrate towards a computing framework which can be integrated with the non-volatile memory itself. Till date, the operating characteristic of conventional magnetic storage had meant that such integration was quite impossible. But with the emergence of CMOS-compatible non-volatile memory technologies such as NAND Flash and other CMOS-compatible non-volatile memory technologies such as PCRAM, STTRAM and Memristor, such an integration if made possible would reap enormous performance and energy savings. The off-chip MAHA framework investigated in this work attempts to realize the same for the popular non-volatile NAND Flash technology (106, 179). Flash has excellent integration density which has already surpassed that of commercial DRAMs and is expected to reach $100\times$ the value of logic integration density by 2020. The integration density for multi-level cell (MLC) Flash is again predicted to be higher compared to single-level cell (SLC) Flash memory. In addition to advantage in integration density, Flash offers low read access times ($\sim 20\text{ns}$) which is expected to improve in future technology nodes (refer to Fig. 9.3(a)). In addition to Flash, non-volatile memory technologies such as PCRAM, Memristor and STTRAM have gained considerable attention. Based on the data provided in (47), we have compared NAND Flash, PCRAM, Memristor and STTRAM memory technologies on the basis of the following criteria: i) memory cell size; ii) number of bits/cell; iii) write endurance; iv) write energy/bit; v) write latency and vi) read latency. Summary of our comparison is provided in Fig. 9.3(b). As seen from Fig. 9.3(b), future non-volatile memory technologies will offer improved write endurance and write latency while remaining competitive with Flash technology in terms of cell size and read latency. All the above observations make computing in non-volatile memory a worthwhile investigation.

- *Memory instrumentation:* A typical NAND Flash architecture consists of a linear array of a large number of memory blocks (typically 4096). Implementing a computing framework within the Flash memory has several

challenges. One of the most important being deciding upon the nature of the compute framework. Irrespective of its nature, the compute framework is constrained by the fact that it may only small amount of logic to the existing flash memory array so as not to compromise the integration density of the memory array. A conventional CPU based software reconfigurable framework requires complex fetch and decode logic in addition to data memory for storing temporary data. Note that for a flash based computing framework, it is not desirable to use the flash for storing temporary data since writing to Flash memory is costly in terms of performance and energy. A more desirable choice is a hardware reconfigurable framework which can map an application spatially and can leverage on the parallel memory resource. A fine-grained fully spatial framework such as a FPGA will however incur significant design overhead in terms of both logic and routing resources. A spatio-temporal framework, which preserves memory integration density and greatly minimizes the requirement for routing resources is therefore a better design choice. We therefore propose to instrument the flash array to realize a flash memory based MAHA framework. Note that the organization for the flash memory is not hierarchical as in the case of on-chip volatile memories such as SRAM array. To realize the MAHA framework, we impart a hierarchical abstraction to the linear array by constructing a hierarchical interconnect framework around it. The issue with using the flash memory for temporary storage is solved by using the wide page register which is present at each block of the flash memory.

- *Using non-volatile memory as a vector co-processor:* In the proposed off-chip MAHA framework, the Flash memory array at each compute block not only acts as a storage for data being processed but also acts as the *function table* used to map the LUTs for the input application mapped. Note that the LUTs are written once during the process of the reconfiguration. The cost of writing is therefore amortized over the duration of execution. In the proposed framework, the Flash memory blocks therefore act as vector co-processors which can map complex applications and evaluate them over

multiple clock cycles. This clearly distinguishes the proposed framework from PIM projects which memory for storage purposes.

9.2 Off-chip MAHA using NAND Flash technology

In order to show the viability of the proposed MAHA framework as a malleable off-chip hardware accelerator, we have chosen *CMOS logic-compatible Single-Level Cell(SLC) NAND flash* as the candidate non-volatile memory technology. The choice for SLC flash is mainly driven by the availability of open source area, power and delay models (180). In this section, we briefly describe the memory based computing model and the proposed hardware reconfigurable framework. Then the details of flash instrumentation and modifications to the CPU-flash interface are presented.

9.2.1 Modifications to Flash array organization

In order to realize a multi-MLB framework we incorporate minimal modifications to the existing flash memory array organization so as not to adversely affect its integration density. Specifications for the baseline flash configuration is provided in Table 9.1. Modifications are primarily required for the following components:

1. *Blocks*: Typically flash memory is organized into multiple blocks, each of which is subdivided into multiple pages. During read, contents of the entire page is first read into the page register and then serially transferred to the flash external interface which is typically 8/16 bits (181). In the proposed MAHA framework, we instrument 1 out of every N memory blocks to operate as a compute element on demand. The value of N is chosen depending on the overhead of MLB logic. The instrumented memory block is hereafter referred to as *MLB block*. As illustrated in Fig. 9.4, the MLB logic includes the i) schedule table; ii) PI inside MLB and iii) a small control logic to select schedule table entries and iv) custom datapath.

9.2 Off-chip MAHA using NAND Flash technology

Figure 9.4: a) Modifications to conventional flash memory interface to realize MAHA framework; b) Instrumented flash memory array for on-demand reconfigurable computing; c) Hierarchical interconnect architecture to connect a group of MLBs. Statically programmable switches are realized using multiplexors.

We propose to use the page register for storing intermediate data inside each MLB. The memory present in each block is used either as the function table to store the LUTs or simply for storing the data to be processed. For a normal flash where page sizes are typically 2KB, the entire page is read. However, for MAHA operation, where the memory inside a MLB block is primarily used to store LUTs, a wide memory read is not required. Hence, for MAHA operation, we propose a *narrowread* scheme (182) which allows 16 to 64-b read instead of a 2KB read. The idea as illustrated in Fig. 9.5 is to insert an additional series transistor for each bitline. This series transistor is enabled by the output of a 8to256 decoder which is used to select 64 bits out of 2KB page. As during normal operation, the page decoder is used to select one of the 128 pages. To summarize, the following design parameters determine the area, delay and energy overhead for each MLB and are carefully chosen only after considering the implications on system-level area, delay and energy for a set of benchmark applications.

- Number of consecutive levels of execution allowed inside each MLB.
- Width of the custom datapath.
- Number of bits read from the memory in *narrow read* mode.
- Datapath granularity inside each MLB.

2. *Inter-Block PI*: In addition to design changes to memory blocks, configurable switches are required to connect multiple MLBs. As illustrated in Fig. 9.4, for inter-block communication, we assume a hierarchical interconnect architecture (*fat tree*) where individual blocks can communicate directly with local neighbors and with distant neighbors only by traversing up the hierarchy. A multiplexor is assumed to be present at each input of a block to select the source of the incoming data. Multiplexors which function similar to FPGA switch-boxes provide full-connectivity and drive signals over a length of $B/2$, where ‘B’ is the width of individual blocks. As described earlier in Chapter 4, the multiplexors to be statically scheduled.

Figure 9.5: Modifications to the flash array architecture to support narrow read operation. The 8to256 decoder is responsible for selection of a set of columns from the entire page size.

9.2.2 Modifications to CPU-Flash interface

Minimal modifications are necessary at the CPU-flash interface to convert the existing non-volatile storage into the proposed MAHA framework. Fig. 9.4 shows the integration of a flash based non-volatile storage into a typical computer system. Apart from the flash memory array, a number of logic structures are responsible for controlling the read and write operations to the flash (181). The Flash Translation Layer (FTL) converts the logical address of a location to its corresponding physical address. The command register is responsible for latching the operation request commands. Bits of the status registers are used to indicate the status (READY/Busy) and success/failure of read, write and erase opera-

Figure 9.6: Control engine external to the flash memory array synchronizes load operation inside memory array and provides address for read/write operation to the write buffer.

tions inside a flash. Depending on the type of operation request, appropriate address bits are selected for block and page decoding. Finally, the write buffer (also referred to as data or cache register) is used to store data before they are written into the flash memory. The NAND flash typically has 8-bit or 16-bit I/O bandwidth. In case of 8-bit I/O, all 8 bits are used for both instructions and data but in different cycles. In case of 16 bits I/O bandwidth, only the lower 8 bits are used for transferring instructions while the data transfer uses all the 16 bits. Modifications required to convert the existing framework into the proposed hardware accelerator are summarized below:

- Along with the address and the data bits, the CPU must send a 2-bit *mode control* signal to the instrumented flash to distinguish between the following modes of operation: i) normal mode (memory read and write); ii) schedule table load mode; iii) PI configuration mode and iv) MAHA execution mode.
- In the MAHA execution mode, the flash memory array behaves like an array of compute machines closely interacting with the storage elements. Since the number of compute elements is finite, large loops to operate on large datasets cannot be fully unrolled inside the array. This large dataset is therefore iteratively processed which requires decomposing a single large loop into smaller loops. As illustrated in Fig. 9.6, each of the smaller loops is responsible for loading data from one or more local MLBs, while the multiplexor select signal is periodically incremented by the outer loop controller. In our architecture, the outermost loop controller is controlled by a small control engine (CE) which resides outside the flash memory array. This control engine is simply a resettable loop counter which can be preset to a count threshold during the process of application mapping. The count threshold for the loop counters realized inside the flash array using MLBs are also preset during application mapping. As illustrated in Fig. 9.6, to realize loops, it is necessary to store the output of each iteration in temporary registers which is implemented using the page registers already present inside each block.

9.2.3 Overhead estimation

We have modeled in detail the overhead due to instrumentation of the Flash memory. Table 9.1 shows the configuration for the baseline Flash architecture. Typical values for this organization were obtained from (106). Area and energy estimates for each logic component of the MLB as reported in Chapter 6 are used to estimate the instrumentation overhead. Area and energy for the routing switches in the hierarchical interconnect was calculated after sizing them in a

9.2 Off-chip MAHA using NAND Flash technology

Table 9.1: Flash organization and performance (106)

Technology	45nm
Total Size	1 GB
Organization	2048 bytes * 128 pages * 4096 blocks
Cell Size	$0.0081\mu m^2$
Read access time	30ns
Program Time	$800\mu s$
Page register Area	$5903\mu m^2$
Flash Block Area	$17827\mu m^2$
Page register read energy (normal mode)	141pJ
Flash read energy (normal mode)	6.29nJ
Page register read energy (narrow mode)	4.5pJ
Flash read energy (narrow mode)	24pJ
Block decoder area	$27086\mu m^2$

manner such that they could drive an effective distance of each side of a Flash memory block (256KB). Note that though the page access time for the flash memory at 45nm technology has been reported to be 30ns (106), based on our trend analysis as reported in Fig. 9.3(a), we have taken 20ns as the typical page access time for the Flash memory. Note that, although it is possible to operate the MLB logic at much faster speed, we have assumed 20ns for MLB cycle time so as not to modify the block read circuitry. Energy consumed in the flash array is modeled following the *FLASHPOWER* model as presented in (180). Compared to the full page read during normal memory operation, energy saving during narrow read primarily occurs due to:

- Reduction in total bitline capacitance that is pre-charged and then discharged during read.
- The wordline driven by the page decoder is segmented and gated by the output of the 8-to256 decoder which selects 64-bits during narrow read operation. The total capacitance value as charged by the wordline drivers is therefore also reduced.

9.2 Off-chip MAHA using NAND Flash technology

Figure 9.7: Area overhead for each Flash block due to instrumentation for different MLB configurations.

- The block decoder is not used during MAHA operation and is therefore excluded from the energy consumption calculations for MAHA mode.
- There is however an increase in the energy due to additional 8to256 decoder required for narrow-read operation.

At 45nm, 20ns gives enough time slack for the logic. Hence, the logic was therefore synthesized with minimum area constraint in order to minimize the area overhead for the instrumented array. Moreover, we have used a low-power PTM 45nm model for our simulation to reduce the MLB energy. For CMOS compatible SLC nand flash memory, area, delay and energy models were obtained from existing literature (180, 181). These were used to estimate the area overhead and energy savings for the proposed *narrow read* scheme. Following the design exploration strategy for the MAHA framework as outlined in Chapter 6, each of the benchmark applications were mapped to different configurations of the instrumented Flash, overall area, delay and energy overhead values were noted and then optimal solution was chosen based on design constraint. For NAND-Flash based MAHA framework, we tried to minimize the area-delay product.

9.2 Off-chip MAHA using NAND Flash technology

Table 9.2: Common MLB design configurations for our benchmark suite

Banks=8, subbanks=1, mats=1, subarrays=2, Buswidth=512			
Datapath	Slices	Levels	Latency(s)
16	0	4	2.52e-7
16	2	3	2.4e-7
16	2	4	2.4e-7
32	0	2	2.41e-7
32	1	2	2.24e-7
32	1	4	2.24e-7
32	2	2	2.4e-7
32	2	2	3.4e-7
32	2	4	2.4e-7

9.2.3.1 Overhead for MAHA

Based on the page register, flash block area and block decoder area, we tried to estimate the area overhead due to the additional MLB logic at each block. Fig. 9.7 shows the area overhead in a Flash block due to instrumentation. Here we assume that 1 in every 4 Flash blocks are instrumented to form a MLB, so that area overhead per block is minimized. As we note from Fig. 9.7, a 32-bit datapath inside the MLB incurs significant area overhead compared to other configurations. The final configuration is however chosen based on cumulative mapping results of the benchmark suite mentioned in Chapter 6.

9.2.3.2 Selection of final configuration

The final configuration, which includes both the MLB configuration as well as the hierarchy and *Buswidth* configurations are selected in the following manner:

- All the benchmarks are run using the *explore* mode of our software flow.
- Arrive at common configurations for which all of them pass *packing* and *routing* step.

9.3 Integration with CPU based software flow

- If there are multiple hierarchy and *Buswidth* configurations corresponding to each MLB design point, choose the configuration with minimal interconnect area/energy overhead.
- Choose the MLB design configuration which offers the minimum latency.

Table 9.2 shows the final MLB configurations obtained corresponding to *Banks=8*, *subbanks=1*, *mats=1*, *subarrays=2*, *Buswidth=512*. From Table 9.2, we note that for the selected benchmarks, 32-b datapath does not lead to any appreciable improvement in latency. Hence, a smaller area overhead configuration *emphDatapath=16*, *Slices=2* and *Levels=3* is chosen as the final configuration. At this MLB and hierarchy configuration, the area overhead for the instrumented Flash array is calculated as:

$$\frac{((Area_{ArrayInstrumented} + Area_{BlockDecoder} + Area_{PI}) - (Area_{FlashArray} + Area_{BlockDecoder})) \times 100}{Area_{FlashArray} + Area_{BlockDecoder}}.$$

According to our calculation the total area estimate for the original array is $90.71mm^2$. The area for the instrumented flash array including the overhead for each MLB, the block decoder and the inter-MLB interconnect is estimated to be $104.51mm^2$, which is 17.67% more than the original flash array estimated. Typically the array size is bigger in actual implementation. The layout area for a 2GB 4-level MLC NAND Flash at 45nm technology node has been reported to be $120mm^2$ (179). For such an array size, the area overhead for instrumentation is expected to much less. Note that here we have considered impact on area for only the memory array, the block decoder and additional programmable interconnects. If the area for write buffer, command and status register are considered, the area overhead due to flash instrumentation would reduce further.

9.3 Integration with CPU based software flow

Since the MAHA framework is intended to work as an accelerator on demand, it is desired that it seamlessly integrates with the conventional software flow. For the proposed off-chip acceleration scheme using the MAHA framework, we follow the conventional approach of identifying the compute/data intensive section of a given

Figure 9.8: MAHA software architecture and its integration into conventional software flow. Figure shows the analogy between MAHA and conventional GPGPU software flow.

application through profiling, followed by mapping of the application kernel to the hardware acceleration unit (36, 183). In this respect, we draw analogy from the popular GPGPU computing platform (170), specifically the compilation process of CUDA-C via the Nvidia CUDA-C compiler, NVCC. For the case of both MAHA and GPU, compilation begins with a C file that contains user directives for off-loading to the hardware accelerator. The region of code marked by these directives is then parsed out of the code and processed by another tool set. In the case of CUDA-C, the GPU instructions are first separated from the CPU instructions, and the two parts are compiled separately. For non-target-specific compilation, the C code is processed with GCC, and the GPGPU instructions are processed with a series of custom tools that result in PTX, or Parallel Thread

Execution code, a pseudo-assembly code geared towards a generic Nvidia graphics card. At run-time, the graphics driver translates the PTX code into target-specific instructions, which can then execute a specific kernel on the GPU within the framework of the host C code. The two sets of code are merged into one binary file (termed as *fatbin*), which a user may run on a CUDA-enabled system.

Similarly, the MAHA C Compiler, or MCC, parses out the section of the code targeted to execution on the MAHA framework. Typically this section is a specific function that is designated for acceleration using custom compiler directives. A data flow graph is then generated from this section in a format that is compatible the MAHA application mapping tool. The compiler also generates code for the I/O and memory operations which are used to synchronize the CPU and the MAHA code execution. The synchronization code is inserted back into the original C code as target-specific assembly, replacing the original function call with the necessary I/O instructions. Fig. 9.8 shows the conventional GPGPU software architecture on the left and the entire MAHA software flow on the right. During runtime, the CPU executes the instructions in the compiled code till it receives a call to the MAHA routine. As part of the routine, it invokes the DMA unit, responsible for transferring the reconfiguration bits for the MAHA framework from main memory to the off-chip reconfigurable framework (Fig. 9.4). Assuming no data dependency, the CPU and the MAHA frameworks execute concurrently. Upon completion of MAHA execution, the CPU receives an interrupt and the resultant output is transferred from the MAHA framework to the CPU.

9.4 Simulation results

Based on the final configuration selected for Flash based MAHA framework, we have estimated the energy-efficiency (measured in terms of energy-delay product or EDP) with the off-chip memory based computing framework and compared it against the energy-efficiency in a conventional processor based execution. The energy efficiency was compared for different sizes of the input dataset. The size of the dataset was limited by the number of MLBs (1024) in a 1GB instrumented

Table 9.3: Processor configuration and energy estimates

Processor Configuration		Energy estimates	
Processor	2-way in-order issue, 32 LSQ, 2 integer ALUs, 2 integer mul/div units, 2 FP ALUs, 2 FP mul/div units, 2 Wr/Rd ports	E_{Instr}	20pJ
Branch Prediction	Combined, 32-entry RAS, 2K 4-way BTB, 8 cycle mis-prediction penalty	$E_{IL1Access}$ $E_{DL1Access}$	39pJ
Caches	32KB 2-way 2 cycle I/D L1, 512KB, 4-way 12 cycle L2	$E_{L2Access}$	390pJ
Main Memory	100 cycle latency, 8-byte wide bus	E_{L2Miss}	1380pJ

Flash memory. The final configuration selected for Flash instrumentation requires 16 Flash blocks connected to each other using a hierarchical interconnect architecture. The total number of parallel DFGs that can be simultaneously executed inside the 1GB Flash is therefore 64. The dataset size is limited by the data that can be loaded and operated on by these 64 DFG instantiations.

9.4.1 Processor energy and performance estimates

The performance and energy for the processor based execution model was obtained as follows:

- The description for each of the benchmark was written in C.
- The C-file was compiled for a portable instruction set architecture (PISA) using the *sslittle-na-sstrix-gcc* cross-compiler which comes with the *Simplescalar 3.0 binutils*.
- The compiled benchmarks were simulated using the *Simplescalar 3.0* tool-suite for the baseline processor configuration as described in Table 9.3 for the entire length of the program. The number of instructions, simulation cycles, number of L1 instruction and data cache accesses, number of L2 cache accesses and number of L2 misses were noted for each.

9.4 Simulation results

Table 9.4: Performance and energy results for processor based execution of benchmark suite

Benchmark	Data Size (Bytes)	Instr Cnt	Cycle Cnt	IL1 Access	DL1 Access	L2 Access	L2 Miss	Total Energy (pJ)
AES	1.64E+04	1.46E+08	2.36E+08	1.48E+08	9.72E+07	2.85E+03	2.01E+03	1.25E+10
	6.55E+04	5.82E+08	9.45E+08	5.91E+08	3.89E+08	1.30E+04	3.55E+03	4.98E+10
	2.62E+05	2.33E+09	3.78E+09	2.36E+09	1.55E+09	4.60E+04	9.69E+03	1.99E+11
	1.05E+06	9.32E+09	1.51E+10	9.45E+09	6.22E+09	1.78E+05	9.99E+04	7.98E+11
ATR	1.64E+04	7.50E+08	1.232E+09	7.55E+08	3.14E+08	2.59E+04	5.86E+03	5.67E+10
	6.55E+04	3.16E+09	5.202E+09	3.19E+09	1.32E+09	1.00E+05	1.98E+04	2.39E+11
	2.62E+05	1.30E+10	2.138E+10	1.31E+10	5.44E+09	7.58E+05	1.33E+05	9.82E+11
SHA	3.12E+05	8.19E+07	1.29E+08	8.23E+07	3.61E+07	2.20E+04	1.19E+04	6.28E+09
	6.24E+05	2.29E+08	3.64E+08	2.30E+08	1.02E+08	4.22E+04	2.17E+04	1.76E+10
Motion Est.	1.64E+04	3.29E+08	6.03E+08	3.34E+08	1.68E+08	1.42E+04	2.92E+03	2.62E+10
	6.55E+04	1.52E+09	2.79E+09	1.55E+09	7.73E+08	1.72E+05	6.66E+03	1.21E+11
	2.62E+05	6.5E+09	1.2E+10	6.64E+09	3.31E+09	8.01E+05	2.71E+04	5.19E+11
Smith Waterman	5.24E+05	8.90E+07	1.61E+08	8.91E+07	3.98E+07	6.25E+05	3.33E+05	7.51E+09
DCT	1.64E+04	1.87E+09	2.46E+09	1.89E+09	6.89E+08	2.75E+04	3.98E+03	1.38E+11
	6.55E+04	3.75E+09	4.87E+09	3.82E+09	1.22E+09	2.78E+05	1.17E+04	2.71E+11
	2.62E+05	1.12E+10	1.45E+10	1.15E+10	3.31E+09	3.37E+05	4.74E+04	8.02E+11
	1.05E+06	4.11E+10	5.31E+10	4.23E+10	1.18E+10	5.88E+05	2.76E+05	2.93E+12
DWT	2.62E+05	1.19E+08	1.68E+08	1.2E+08	4.67E+07	8.53E+05	2.81E+04	9.25E+09
	1.05E+06	4.79E+08	6.88E+08	4.82E+08	1.94E+08	3.37E+06	1.10E+05	3.74E+10
	4.19E+06	1.92E+09	2.87E+09	1.94E+09	7.77E+08	2.61E+07	4.38E+05	1.55E+11
Max Entropy Class.	5.24E+05	2.05E+08	2.81E+08	2.06E+08	8.68E+07	7.05E+04	2.04E+04	1.56E+10
	1.05E+06	4.11E+08	5.66E+08	4.13E+08	1.74E+08	1.29E+05	6.68E+04	3.13E+10
	2.10E+06	8.51E+08	1.17E+09	8.53E+08	3.62E+08	1.62E+05	9.96E+04	6.46E+10
Color Interp.	2.62E+05	3.90E+08	5.81E+08	3.93E+08	1.65E+08	8.75E+05	4.68E+04	3.00E+10
	5.24E+05	7.80E+08	1.16E+09	7.86E+08	3.29E+08	1.75E+06	1.01E+05	5.99E+10
	1.05E+06	1.56E+09	2.33E+09	1.57E+09	6.58E+08	3.49E+06	2.31E+05	1.20E+11
Census	5.24E+05	4.06E+08	5.51E+08	4.16E+08	1.31E+08	2.00E+05	1.33E+05	2.97E+10
	1.05E+06	7.99E+08	1.08E+09	8.18E+08	2.60E+08	3.88E+05	2.58E+05	5.85E+10
	2.10E+06	1.23E+09	1.69E+09	1.25E+09	4.42E+08	5.21E+05	3.90E+05	9.15E+10

- The total energy for the processor based execution was calculated as:

$$\begin{aligned}
E_{proc} = & Count_{Instr} * E_{Instr} + Count_{IL1Access} * E_{IL1Access} + \\
& Count_{DL1Access} * E_{DL1Access} + Count_{L2Access} * E_{L2Access} + Count_{L2Miss} * E_{L2miss}
\end{aligned}$$

Representative values for instruction execution energy (E_{Instr}), L1 I-cache access energy ($E_{IL1Access}$), L1 data cache access energy ($E_{DL1Access}$), L2 cache access energy ($E_{L2Access}$) and L2 miss or off-chip access energy (E_{L2Miss})

Table 9.5: Benchmark mapping results

Benchmark	Vert Cnt in orig. DFG	Vert Cnt in decomp. DFG	Vert Cnt after fusion	Total MLB Energy (J)	Total int. Energy (J)	Latency (s)	# of MLBs	Reconfig. data (Bytes)	Op per MLB
AES	42	62	51	5.20E-09	1.09E-11	1.8E-07	2	4.19E+04	25.50
ATR	92	118	109	1.13E-08	4.79E-11	2.55E-07	5	1.26E+05	21.80
census	49	51	47	5.40E-09	1.09E-11	2.4E-07	2	1.90E+04	23.50
color interp.	44	62	62	6.48E-09	8.11E-11	1.35E-07	6	1.95E+04	10.33
2D DCT	116	150	142	1.35E-08	2.50E-10	3.75E-07	16	1.70E+04	8.88
DWT	50	61	61	6.96E-09	3.44E-11	3.15E-07	4	1.81E+04	15.25
max entropy	115	132	120	1.27E-08	4.71E-11	3.9E-07	6	5.51E+04	20.00
motion estimation	124	133	133	1.45E-08	1.55E-10	2.55E-07	13	1.41E+05	10.23
sha	73	221	175	1.67E-08	2.46E-10	1.95E-07	11	1.63E+05	15.91
Smith Waterman	151	152	118	1.40E-08	4.97E-11	3.75E-07	5	1.45E+05	23.60

at 45nm technology node were obtained from (68). These values are listed in Table 9.3.

- Assuming the processor runs at 1GHz clock period, the total execution time for each benchmark was calculated as: $T_{exec} = SimCycle * 1E-9$.

Table 9.4 lists the energy and performance results for the different benchmarks when executed on the baseline processor for different sizes of the input dataset.

9.4.2 MAHA energy and performance estimates

For the MAHA framework, the energy and performance estimates were obtained as follows:

- For each benchmark, we model the dataset that would be finally returned to the processor if the actual data was processed off-chip outside the processor. Energy and performance for loading this resultant dataset was estimated through *Simplescalar* simulations.
- For the final configuration, we mapped each of the benchmarks to the instrumented Flash array and noted the energy and performance parameters

summarized in Table 9.5. The important energy parameters are i) *Total MLB Energy*, which indicates the total energy consumed in all the MLBs of a single DFG instantiation for evaluating a single input vector; ii) *Total Interconnect Energy*, which indicates the energy expended in the inter-MLB programmable interconnects in a single cycle of a single DFG instantiation. In Table 9.5, the *Latency* is calculated as $(\# \text{ of vert. levels in CDFG}) \times (\text{Cycle time})$ where *Cycle time* is taken as 20ns.

- The total energy consumption and latency for evaluating a given dataset are calculated as $(\# \text{ of vectors}) \times (\text{Energy per vector})$ and $(\# \text{ of vectors}) \times (\text{Latency per vector})$. *Energy per vector* is calculated as $(\text{Total MLB Energy}) + (\text{Total Interconnect Energy}) \times (\# \text{ of vert. levels in CDFG})$. The total number of vectors each DFG has to each operate is given by $(\text{Dataset per DFG}) / (\text{Data per vector})$.
- Before actual execution, it is necessary to reconfigure the MAHA framework. Since our software flow generates the bitstream for functional validation, it is easy to estimate the total size of the reconfiguration data. The reconfiguration data for different benchmarks is provided in Table 9.5. The reconfiguration latency is calculated as $(\text{Time to send reconfig. data}) + (\text{Time to write into Flash memory})$. We assume that the data is being send in a pipelined fashion, such that 10^4 cycles (68) are required to transfer first 16-bits of data (since we assume 16-bit Flash interface) and the remaining data is sent in $\text{Reconfig. data}/16$ cycles. Writing to the flash also happens in a pipelined fashion with the first $800\mu s$ required to write first page, followed by a total of $(\text{Reconfig. Data}) / (\text{Page Size})$. The reconfiguration energy is also calculated in a similar manner: $(\text{Energy to txfer reconfig. data}) + (\text{Energy to write to Flash})$. In (68), 13000pJ is the estimate of the energy required to send data to memory with levels higher than the main memory. With this estimate and assuming the write energy per bit for Flash to be 5fJ (47), the total reconfiguration energy is calculated as $(5fJ + 13000pJ/16) \times \text{Reconfig. Data}$.

9.4 Simulation results

Figure 9.9: Breakdown of the total energy consumed in the MAHA framework and comparison with the energy required by the processor.

Figure 9.10: Breakdown of the total execution time in the MAHA framework and comparison with processor based execution.

The breakdown for energy and latency for different benchmarks when mapped to the MAHA framework is shown in Fig. 9.9 and Fig. 9.10. The breakdown is shown for the case when the largest dataset is executed by the MAHA framework. The energy and the delay speed-up over the processor based execution is also shown in Fig. 9.9 and Fig. 9.10. From Fig. 9.9 and Fig. 9.10, we note that, i) interconnect energy is much smaller compared to the other energy components; ii) for few benchmarks (such as AES, Smith Waterman and census applications), reconfiguration energy and latency dominates the total energy and latency; iii) following Amdahl's law, the maximum energy and latency improvement for a few applications (such as AES, ATR, DCT, DWT and color interpolation) which produce the same amount of resultant data as the original dataset is always capped by the sequential portion of the code which is executed in the processor.

Table 9.6: Improvement in energy-efficiency in off-chip MAHA framework

	Times improv in EDP w/ Reconfig.	Times improv. in EDP w/o Reconfig.
AES	187.7	471
ATR	49	67
SHA-1	13	97
Motion Est.	31	32
Smith Waterman	4	181
2D-DCT	2546	2630
DWT	1	1.5
Max. Entropy Class.	3	4
Color Interp.	2	2
Census	7966	9680
Average	35	73

Table 9.6 shows the final energy-efficiency comparison between the processor based execution and off-chip MAHA based execution with and without the re-configuration overhead. From Fig. 9.9 and Fig. 9.10 and Table 9.6, we note that benchmarks for which EDP improvement is currently constrained by the re-configuration overhead, the improvement is drastic if the reconfiguration overhead is neglected. Reconfiguration overhead is typically ignored for reconfigurable system, since it is amortized over the large volume of data processed by the system. If we assume, that the data being processed is present in the Flash blocks which are not instrumented, then the reconfiguration overhead is also nullified for the case of MAHA. Currently, data is assumed to be present only in the MLBs which are instrumented.

Conclusion and future directions

In this work, we have presented a novel hardware reconfigurable framework that utilizes two-dimensional memory array for computation. The idea proposed is to partition an application to multi-input multi-output LUTs and map them to dense 2-D memory arrays at each compute element of the framework. Each of these compute elements evaluates the LUTs over multiple clock cycles in a temporal manner and multiple compute elements combine to spatially map any large application.

We have defined the hardware architecture for the proposed spatio-temporal reconfigurable platform. Details of synchronization across multiple compute elements in the framework using time-multiplexed interconnect have been presented. The hardware modifications to realize a reconfigurable framework on demand from an existing memory organization are also proposed as part of this work.

In addition to the hardware architecture, we have developed a complete software flow which automates the application mapping process to the MBC framework. Details of the software flow are included in this dissertation. The software flow not only allows design space exploration for the MBC framework, but also functional validation of applications mapped to the proposed framework.

We have demonstrated how the memory based computing model can be effectively used to improve the reliability of operation for a general-purpose processor

framework. In addition, we have also investigated hardware-software co-design approaches for improving the reliability of operation for the MBC framework.

Finally, we have presented MAHA, a memory based hardware reconfigurable framework, which instruments an off-chip CMOS-compatible non-volatile Flash memory for on-demand computing. Computing in such close proximity with the data, minimizes the energy requirement for the system and mitigates the Von-Neumann bottleneck.

We believe this work lays the foundation for alternative reconfigurable frameworks which are significantly more energy-efficient, scalable with technology and amenable to easy application mapping. There are, however, a number of design choices and software optimizations which can be exploited to enhance the performance and functionality of the proposed MBC framework. We are listing some of them below:

- *Modifications to the MLB interconnection architecture:* The MLB architecture can be further optimized. Currently we use dummy registers to support feed-forward signals within each MLB. This is primarily because we do not support register selection while writing the result of a LUT or a datapath operation. If selection is allowed for both reading and writing to the temporary registers, then dummy registers can be eliminated.
- *Exploration of inter-MLB architectures:* A larger scope of optimization is present in defining the inter-MLB communication architecture. For MBC based stand-alone reconfigurable computing, we currently rely on an elaborate programmable interconnect framework similar to the FPGA. We have demonstrated that MBC reduces the requirement of PI significantly through localized execution and time-multiplexing of the inter-MLB communication channels. It is therefore possible to define the inter-MLB interconnection framework which is far less demanding in terms of area, delay and power compared to PI conventional FPGAs.
- *Software optimizations:* The largest scope of optimization lies in the software. The spatio-temporal nature of the proposed framework impacts

unique advantages which cannot be leveraged in a purely spatial framework. For example, it is possible to minimize the impact of inter-MLB routing delay by scheduling the dependant operations in the next clock cycle. This retiming increases the latency by one clock cycle but prevents increase in cycle time. Similarly, to minimize the routing congestion, it is possible to schedule the transmission of signals over a time-multiplexed interconnect which is again not possible over a fully spatial framework. All these optimizations can be part of our future investigations.

- *Exploring the effectiveness of MAHA framework for on-chip memories:* We have noted that reconfigurable computing based on off-chip non-volatile memory brings significant improvement to system energy and relieves the bandwidth bottleneck for data-intensive applications. In future, we will investigate the effectiveness of on-chip MAHA in accelerating compute intensive applications.
- *Exploring the effectiveness of MBC framework for emerging non-volatile memories:* A number of promising non-volatile non-CMOS memory technologies are already on the horizon. For some of these technologies, we have already demonstrated that MBC can be an extremely effective computing platform. Future work will explore other emerging memory technologies such as Memristor and PCRAM and investigate their suitability in realizing the proposed MBC framework.

Appendix A

A.1 Description of the hardware architecture

A.1.1 Custom datapath in the MAHA framework

Fig. A.1 shows the custom datapath proposed for the MAHA framework. As we note from Fig. A.1, the custom datapath supports a multiplication, an addition and a shift or rotate or select operation in a given clock cycle. Note that to support all the three operations, the inputs must be shared either between the inputs or the outputs of the multiplication or addition operation. For example, the structure can support the following operations concurrently: i) $A \times B$, ii) $A + B$ and iii) $C \ll / \gg \textit{shiftamt}$ or $A, B \ll / \gg \textit{shiftamt}$ or $C \textit{lrot/rrot rotamt}$ or $A, B \textit{lrot/rrot rotamt}$ or $\textit{sel?}A : B$, where *shiftamt* and *rotamt* denotes the amount by which the operands must be shifted or rotated, *lrot* and *rrot* denote the left and right rotations and $\textit{sel?}A : B$ denotes the selection between two-operands based on the select signal *sel*. Otherwise, the datapath can also support operations such as $((A \times B) + C) \ll / \gg \textit{shiftamt}$ where the two operands A and B are multiplied and added with another operand C. The result of addition is finally shifted by the *shiftamt*. The same holds true for rotation. The custom datapath as illustrated in Fig. A.1 is specified by a single design parameter *Addsize* which denotes the width for the adder. Currently the multiplier size

Figure A.1: Custom datapath inside each MLB consists of an adder, a multiplier and a shift/select/rotate unit. The size of the multiplier is half the size of the add and shift/select/rotate unit.

denoted as *Multsize* is set to half the adder size so that the multiplier output is of size *Addsize*. The bitwidth for the shifter is same as *Addsize*.

Note that the input operand for shift/rotate/select unit is divided into the upper and lower bits to allow the operands A and B to be concatenated before the resultant is shifted or rotated. This allows decomposition of operands of large size using shift/rotate units of smaller size. For multiplication A and B must be of size $Addsize/2$, while for addition they can be of size $Addsize/2$ or *Addsize*. Operand C is always of size $Addsize/2$. Many applications such as Secure hash Algorithm (SHA) require shift/rotate operations of constant value. As shown in Fig. A.1, this value must be provided from the *schedule table*. The custom datapath operations occur in parallel to *function table* access in the negative half of the clock cycle and each of the multiplier, adder and shift/rotate/select unit output is written into the temporary registers. Select signals for the multiplexers which control the flow of data inside the custom datapath and select the shift/rotate/select signals are read from the schedule table. Each MLB in the

MAHA framework with the memory array as the *function table* and the custom datapath thus advocates the same multi-cycle temporal computing model that was proposed for MBC based generic reconfigurable framework.

A.2 Description of individual steps in software flow

A.2.1 Decomposition

Following decomposition routines are currently part of the tool.

A.2.1.1 Decompose_bitswC:

For additions and subtractions, this routine decomposes bit-sliceable with carry operations to satisfy the maximum LUT input and output constraint if custom adder is absent. If custom adder is present inside MLB, then decomposition is performed to form sub-operations which fit into the adder size. For comparison operations, the decomposition is always with respect to the maximum LUT input and output values. Fig. A.2(a) shows the decomposition of a 32-bit addition operation into 4 8-b additions, for a 8-b custom datapath adder.

A.2.1.2 Decompose_bits:

Bit-sliceable operations without carry are decomposed into parallel sub-operations which satisfy the maximum LUT input and output constraint.

A.2.1.3 Decompose_mult:

This decomposes 2-input multiplication into sub-operations which consists of multiplications of smaller bitwidth and partial product addition. The decomposition

A.2 Description of individual steps in software flow

Figure A.2: a) A 32-bit addition operation is decomposed into 4 8-bit addition operation for 8-bit custom datapath. b) A 16-b shift operation decomposed to fit 8-bit shifter and c) decomposition of load operation with 16-b address distributed to MLBs with 4KB memory.

is performed in keeping with the multiplier size if a custom multiplier is present inside a MLB or according to the LUT input and output bounds.

A.2.1.4 Decompose_sel:

If a custom selection logic is present inside a MLB, a N-input selection operation is decomposed into multiple 2-1 selection. Note the decomposition is performed along the number of operands as well as along the size of the input operands.

A.2.1.5 Decompose_shift/rotate:

Shift/rotate operations are also decomposed into shift and rotate operations of smaller bitwidth according to the size of the custom shift/rotate logic present inside each MLB. The decomposition of the shift and rotate operations are performed following the policy outlined in (184). The idea is to divide the operand to be shifted or rotated into overlapping segments with each segment equal to

the size of the shift/rotate unit. Final shifted/rotated value is selected from the output of these individual shift/rotate units. Fig. A.2(b) shows the decomposition of a 16-bit shift operation using a 8-bit shifter.

A.2.1.6 Decompose_load/store:

As pointed out in (108), the fundamental obstacle to FPGA-based computing today is the FPGAs lack of a common, scalable memory architecture. When developing applications for FPGAs, designers are often directly responsible for crafting the application-specific infrastructure logic that manages and transports data to and from the processing kernels. It is in this respect that an effort was made in (108) to abstract the memory resources present inside the FPGA to mimic a virtual memory system similar to the processor. The *loads* and *stores* operation as defined by our ISA is also meant to serve the same purpose. The proposed decomposition of these operations into individual address spaces of the memories distributed among multiple MLBs is however quite different from the approach proposed in (108). In (108), the distributed memory inside the FPGA is divided into groups and each group is served by a network-on-chip router which is responsible for matching the address for that group and storing/forwarding data from that group. Abstracting the explicit data distribution and access operations in the above manner is beneficial when the memory under consideration is large. However, the total dataset for many applications (such as discrete cosine transform (DCT), advanced encryption standard (AES) etc.) can be suitably divided into smaller non-overlapping sets which can be operated on in parallel by multiple instantiations of these applications. For these smaller datasets, the proposed decomposition of *load* and *store* would be more beneficial.

The idea is to first allocate memory in one or more MLBs depending on the address size used for load/store and the amount of memory present inside each MLB. For *load* operation, data from these distributed memories will be read using the lower address bits, while the higher address bits will be used to multiplex the data coming from these distributed memories. For *store*, the address is decoded and only a particular MLB is selected for storing the input

data. This decomposition requires only a multiplexer for *load* and a decoder for the *store* operation. The user is still abstracted from explicitly instantiating the memory segments and accessing/storing the data from/to the correct one. Fig. A.2(c) shows the decomposition of a load operation with 16-bit address. If each MLB incorporates 4KB of memory, the load operation is distributed to 16 such MLBs.

A.2.1.7 Decompose_loadPR/storePR:

loadPR and *storePR* vertices are only allowed in the case where each MLB incorporates a large temporary storage referred to as the page-register. Decomposition of the loadPR and storePR operations are performed in the same manner as load and store, except the decomposition is constrained by the page-register size.

A.2.2 Fusion

Individual fusion routines are described below:

Figure A.3: Formation of cycle among partitions in conventional partitioning approaches for FPGA. A similar cycle in the proposed framework would necessitate multiple execution of the same partition.

A.2.2.1 Fusion of random LUT based operations

As mentioned earlier, conventional hypergraph based partitioning techniques (112) can lead to the formation of cycles, which may result in multiple evaluation of the same partition in the MBC framework. In the example shown in Fig. A.3(a), the original netlist is technology mapped to 4-input LUTs. In FPGA, partition P1 can be evaluated twice depending on the relative latency of the inputs $I_7 \cdots I_{12}$. Given enough time a spatial framework such as FPGA will evaluate the output correctly. However, MBC framework imposes the constraint that the original netlist be partitioned in a topological manner and the partitions are evaluated topologically. In light of the above observation, we have developed a heuristic for partitioning a target application into multi-input multi-output partitions. The vertices inside each partition are then fused to form a single vertex to be mapped as a LUT operation.

Pseudocode for the proposed fusion (also referred to as *partitioning* since vertices are first grouped into partitions and thereafter fused) heuristic is presented in Algorithm 4. The input to our partitioning algorithm is a Directed Acyclic Graph (DAG) representation of the technology mapped circuit netlist. Let us denote this graph by $G(V, E)$, where V is the set of combinational nodes and E represents the dependencies between the nodes. The partitioning algorithm begins by topologically sorting the vertices in G , which is accomplished through the following tasks.

- Assign to each vertex $v \in V$ a level $l(v)$ where $l(v) = \max(l(v') | v' \in D(v)) + 1$, where $D(v)$ denotes the set of drivers for vertex v . The primary inputs are assigned to level 0.
- Sort the vertices in the descending order of their levels.

Let V' denote the array for the topologically sorted vertices. If there are no pre-existing partitions, a new partition is created with the Maximum Fanout Free Cone (MFFC) (185) with vertex $v \in V'$ serving as the starting node. If there are pre-existing partitions, for each vertex $v \in V'$, the algorithm finds the Maximum Fanout Free Subgraph (MFFS) (185) of the node set $N_i \cup v$, where N_i denotes

A.2 Description of individual steps in software flow

Algorithm 4 *Partitioning*

```

1: Input: Circuit as DAG  $(G(V, E))$ , partition size  $(M \times N)$ 
2: Output: Set of partitions  $(P'')$ 
3: Levelize the vertices  $v_i \in V$ 
4: Sort vertices topologically and store them in  $V'$ 
5: for  $i = 0$  to  $|V'| - 1$  do
6:   if  $v_i \in V'$  is unmarked then
7:     if  $|P|$  then
8:       for  $j = 0$  to  $|P| - 1$  do
9:          $temp_j = N_j \cup v_i$ 
10:        if  $MFFS(temp_j)$  is  $\leq$  Max_LUT_input & number of outputs  $\leq$  Utblkx & does not form cycle
            with other partitions then
11:           $C = C \cup MFFS_j$ 
12:        end if
13:      end for
14:      if  $C \neq \emptyset$  then
15:        Find  $C_j \ni I(C_j) = \min(I(C_i) | C_i \in C)$ 
16:        For partition  $P_k$  corresponding to  $C_j$ 
17:        Update nodeset of  $P_k = C_j$ 
18:        Mark  $v_j | v_j \in C_j$ 
19:      else
20:        Form new partition  $p \ni N(p) = MFFC(v_i)$ 
21:         $P = P \cup p$ 
22:        Mark  $v_j | v_j \in MFFC(v_i)$ 
23:      end if
24:    else
25:      Form new partition  $p \ni N(p) = MFFC(v_i)$ 
26:       $P = p$ 
27:      Mark  $v_j | v_j \in MFFC(v_i)$ 
28:    end if
29:  end if
30: end for

```

the set of nodes already present in partition P_i . Let us denote it as $MFFS_i$. P_i is considered as a possible candidate to include vertex v if $MFFS_i$ satisfies the following conditions:

- i. M -feasible
- ii. the number of outputs is less than N
- iii. $MFFS_i$ does not lead to formation of cycle with other partitions.

$MFFS_i$ is thereafter stored in an array C . Once all the existing partitions are investigated, the most suitable candidate is obtained as P_j where $I(P_j) =$

Figure A.4: Fusion of multiple bit-sliceable operations into a single bit-sliceable operation.

$\min(I(C_i) | C_i \in C)$, $I(C_i)$ being the number of inputs to the MFFS stored as element C_i . This ensures that the vertices with the most overlapping input cone are grouped together.

A.2.2.2 Fusion of bit-sliceable operations

Bit-sliceable operations that cannot be fused together by the above MFFC/MFFS based fusion routine due to LUT input/output constraint, are fused together and then bit-sliced to satisfy the LUT input/output constraint. Since the minimum size of each operand in the framework is defined by the *granularity*, the number of bit-sliceable operations fused (N) through this routine must satisfy the constraint $N \times \text{granularity} \leq \text{LUT}_{\text{input}}$. The fusion of bit-sliceable operations is currently implemented as a greedy heuristic which first levelizes all vertices in the CDFG. Then in an order of decreasing vertex levels, input cones for vertices with type *bits* are traversed for vertices of similar types which contribute to the immediate fanin of the vertex under consideration. If the number of inputs to these two vertices satisfies the above constraint, they are fused together to form a single vertex. Vertex of type *bits* which form the immediate fanin of this fused vertex are again considered for fusion and the heuristic proceeds in this manner. Unlike

MFFC/MFFS based algorithms which are optimal, this greedy heuristic fails to identify the opportunity of fusing vertices when there are reconvergent fanouts. The fused vertices are thereafter bits-sliced in accordance to the maximum LUT output constraint. The heuristic is illustrated in Fig. A.4(a), while the example is shown in Fig. A.4(b).’

A.2.2.3 Fusion of custom datapath operations

Any custom datapath as described in Fig. A.1 present inside the MLB lends itself to more opportunities for fusion of individual operations mapped to the custom datapath. For the datapath illustrated in Fig. A.1, following vertex types can be obtained after fusion:

- *multadds*: stands for the function $(a * b + c) \ll shiftamt$.
- *multaddsel*: stands for the function $sel?(a * b + c) : c$.
- *adds*: stands for the function $(a + b) \ll shiftamt$.
- *addsel*: stands for the function $sel?(a + b) : a$.
- *adds*: stands for the function $(b + c) \ll shiftamt$.
- *multadd*: stands for the function $a * b + c$.
- *mults*: stands for the function $(a * b) \ll shiftamt$.
- *multsel*: stands for the function $sel?(a * b) : c$.

As evident from the above vertex types, constraint for fusion of the datapath operations is that they must share one or more input operands. Note that operations involving addition are equally applicable for subtraction and shift operation includes both left and right shift.

Bibliography

- [1] M. SIPPER E. SANCHEZ. **Configurable chips meld software and hardware.** *IEEE Computers*, **33**(1), 2000. 1
- [2] R.TESSIER W.BURLESON. **Reconfigurable Computing for Digital Signal Processing: A Survey.** *Journal of VLSI Signal Processing Systems*, **28**(1/2), 2001. 1, 48
- [3] N.TREDENNICK B.SHIMAMOTO. **Go Reconfigure.** *IEEE Spectrum*, **40**(12), 2003. 1
- [4] S.BROWN J.ROSE. **FPGA and CPLD architectures: a tutorial.** *IEEE Design and Test of Computers*, **13**(2), 1996. 2
- [5] [ONLINE]. **FPGAs Provide Reconfigurable DSP Solutions.** www.altera.com/literature/wp/wp_dsp_fpga.pdf. 2, 22
- [6] [ONLINE]. **DSP Co-Processing in FPGAs: Embedding High-Performance, Low-Cost DSP Functions.** www.xilinx.com/support/documentation/white_papers/wp212.pdf. 2, 22
- [7] F.LI Y.LIN AND L.HE. **Field Programmability of Supply Voltages for FPGA Power Reduction.** *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **26**(4), 2007. 2, 20
- [8] S.SRINIVASAN A.GAYASEN N.VIJAYKRISHNAN M.KANDEMIR Y.XIE AND M.J.IRWIN. **Improving soft-error tolerance of FPGA configuration bits.** In *ICCAD*, 2004. 2, 21, 24, 119

- [9] H.SINGH M.LEE G.LU F.J.KURDAHI N.BAGHERZADEH AND E.M.CHAVES FILHO. **MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications.** *IEEE Transactions on Computers*, **49**(5), 2000. 3, 19, 30, 45, 67, 75, 113
- [10] J.BACKUS. **Can programming style be liberated from the von-neumann style? A functional style and its algebra of programs.** *ACM Communications*, **21**(8), 1978. 4, 47, 187
- [11] K.COMPTON AND S.HAUCK. **Reconfigurable computing: a survey of systems and software.** *ACM Computing Surveys*, **34**(2), 2002. 9, 11
- [12] S.DAS A.P.CHANDRAKASAN, A.RAHMAN AND R.REIF. **Wiring requirement and three-dimensional integration technology for field programmable gate arrays.** *IEEE Transactions on Very Large Scale Integration Systems*, **11**(1), 2003. 9, 21, 42
- [13] I.KUON AND J.ROSE. **Area and delay trade-offs in the circuit and architecture design of FPGAs.** In *Intl. Symp. on FPGAs*, 2008. 9
- [14] [ONLINE]. **The Landscape of Parallel Computing Research: A View From Berkeley.** http://view.eecs.berkeley.edu/wiki/Main_Page. 10, 30
- [15] F.POLLACK. **New microarchitecture challenges in the coming generations of CMOS process technologies.** In *Intl. Symp. on Microarchitecture*, 1999. 11, 26
- [16] E.S.CHUNG P.A.MILDER J.C.HOE AND K.MAI. **Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?** In *Intl. Symp. on Microarchitecture*, 2010. 11, 31, 46, 188, 192, 193, 199
- [17] R.HARTENSTEIN. **A decade of reconfigurable computing: a visionary retrospective.** In *DATE*, 2001. 12, 18, 20, 67
- [18] K.EGURO AND S.HAUCK. **Resource allocation for coarse-grain FPGA development.** *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **24**(10), 2005. 12

- [19] V. BETZ J.ROSE AND A.MARQUARDT. *Architecture and CAD for Deep-Submicron FPGA*. Springer, 1999. 13
- [20] A.DEHON. **DPGA Utilization and Application**. In *Intl. Symp. on FPGAs*, 1996. 16, 44, 46, 75
- [21] S.HAUCK T.W.FRY M.M.HOSLER AND J.P.KAO. **The Chimaera reconfigurable functional unit**. *IEEE Transactions on Very Large Scale Integration Systems*, **12**(2), 2004. 16, 186, 194
- [22] C.EBELING D.C.CRONQUIST AND P.FRANKLIN. **RaPiD - Reconfigurable Pipelined Datapath**. In *Intl. Workshop on Field-Programmable Logic and Applications*, 1996. 17
- [23] E.MIRSKY AND A.DEHON. **MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources**. In *FPGAs for Custom Computing Machines*, 1996. 19
- [24] S.C.GOLDSTEIN H.SCHMIT M.MOE M.BUDIU S.CADAMBI R.R.TAYLOR AND R.LAUFER. **PipeRench: a coprocessor for streaming multimedia acceleration**. In *Intl. Symp. on Computer Architecture*, 1999. 19, 30, 45, 186
- [25] C.BRUNELLI F. GARZIA AND J.NURMI. **A coarse-grain reconfigurable architecture for multimedia applications featuring subword computation capabilities**. *Journal of Real-Time Image Processing*, **3**(1), 2006. 18
- [26] [ONLINE]. **The Kress ALU Array**. <http://xputers.informatik.uni-kl.de/faq-pages/kressalu.html>. 18
- [27] R.RAZDAN AND M.D.SMITH. **A High-Performance Microarchitecture with Hardware-Programmable Functional Units**. In *Intl. Symp. on Microarchitecture*, 1994. 18
- [28] T.MIYAMORI AND K.OLUKOTUN. **REMARC: Reconfigurable Multimedia Array Coprocessor**. In *Intl. Symp. on FPGAs*, 1998. 18
- [29] R.D.WITTIG AND P.CHOW. **OneChip: an FPGA processor with reconfigurable logic**. In *FPGAs for Custom Computing Machines*, 1996. 18

- [30] J.BABB ET AL. **The RAW benchmark suite: computation structures for general purpose computing.** In *FPGAs for Custom Computing Machines*, 1997. 18
- [31] M.GOKHALE ET AL. **SPLASH: A Reconfigurable Linear Logic Array.** In *Intl. Conference on Parallel Processing*, 1990. 18
- [32] J.R.HAUSER AND J. WAWRZYNEK. **Garp: a MIPS processor with a reconfigurable coprocessor.** In *FPGAs for Custom Computing Machines*, 1997. 18
- [33] D.CHEN AND J.RABAEY. **Reconfigurable Multi-Processor IC for Rapid Prototyping of Algorithm-Specific High-Speed Datapaths.** *IEEE Journal of Solid-State Circuits*, **27**(12), 1992. 18
- [34] [ONLINE]. **Tensilica's Xtensa customizable processors.** <http://www.tensilica.com/products/xtensa-customizable.htm>. 19
- [35] [ONLINE]. **Stretch: Software Configurable Processors.** <http://www.stretchinc.com/>. 19
- [36] F.BARAT R.LAUWEREINS AND G.DECONINCK. **Reconfigurable Instruction Set Processors from a Hardware/Software Perspective.** *IEEE Transactions on Software Engineering*, **28**(9), 2002. 20, 212
- [37] T.MUDGE. **Power: A First-Class Architectural Design Constraint.** *IEEE Computers*, **34**(4), 2001. 20
- [38] I.KUON AND J.ROSE. **Measuring the Gap Between FPGAs and ASICs.** *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **26**(2), 2007. 20, 23, 67
- [39] L.SHANG A.S.KAVIANI AND K.BATHALA. **Dynamic power consumption in VirtexTM-II FPGA family.** In *Intl. Symp. on FPGAs*, 2002. 20
- [40] T.TUAN AND B.LAI. **Leakage power analysis of a 90nm FPGA.** In *Custom Integrated Circuits Conf.*, 2003. 20
- [41] F.LI Y.LIN L.HE AND J.CONG. **Low-power FPGA using pre-defined dual-Vdd/dual-Vt fabrics.** In *Intl. Symp. on FPGAs*, 2004. 21

- [42] J.H.ANDERSON AND F.N.NAJM. **Low-power programmable routing circuitry for FPGAs.** In *ICCAD*, 2004. 21
- [43] A.GAYASEN ET AL. **Reducing leakage energy in FPGAs using region-constrained placement.** In *Intl. Symp. on FPGAs*, 2004. 21
- [44] Y.LIN F.LI AND L.HE. **Routing track duplication with fine-grained power-gating for FPGA interconnect power reduction.** In *ASP-DAC*, 2005. 21
- [45] [ONLINE]. **Predictive Technology Model.** <http://ptm.asu.edu/>. 21, 109, 114, 130, 134, 156
- [46] [ONLINE]. **Improving FPGA Performance and Area Using an Adaptive Logic Module.** www.altera.com/literature/cp/cp-01004.pdf. 21, 89
- [47] [ONLINE]. **ITRS 2007 : Interconnect.** <http://www.itrs.net/links/2007itrs/home2007.htm>. 21, 26, 31, 200, 217
- [48] G.LEMIEUX AND D.LEWIS. **Circuit design of routing switches.** In *Intl. Symp. on FPGAs*, 2002. 21, 70
- [49] S.PAUL F.CAI X.ZHANG AND S.BHUNIA. **Reliability-Driven ECC Allocation for Multiple Bit Error Resilience in Processor Cache.** *IEEE Transactions on Computers*, **60**(1), 2011. 21, 28, 172, 176, 177, 178, 179, 185
- [50] Y.LIN L.HE AND M.HUTTON. **Stochastic Physical Synthesis Considering Prerouting Interconnect Uncertainty and Process Variation for FPGAs.** *IEEE Transactions on Very Large Scale Integration Systems*, **16**(2), 2008. 22
- [51] L.CHENG P.WONG F.LI Y.LIN AND L.HE. **Device and Architecture Cooptimization for FPGA Power Reduction.** In *DAC*, 2005. 22
- [52] J.ROSE R.J.FRANCIS D.LEWIS P.CHOW. **Architecture of field programmable gate arrays: The effect of logic functionality on area efficiency.** *IEEE Journal of Solid-State Circuits*, **25**(5), 1990. 23

- [53] E.AHMED AND J.ROSE. **The effect of LUT and cluster size on deep sub-micron FPGA performance and density.** *IEEE Transactions on Very Large Scale Integration Systems*, **12**(3), 2004. 23, 57, 89
- [54] A.SHARMA K.COMPTON C.EBELING AND S.HAUCK. **Exploration of Pipelined FPGA Interconnect Structures.** In *Intl. Symp. on FPGAs*, 2004. 23
- [55] [ONLINE]. **Achronix Semiconductor Corp.** <http://www.achronix.com/>. 23
- [56] V.BETZ AND J.ROSE. **FPGA routing architecture: segmentation and buffering to optimize speed and density.** In *Intl. Symp. on FPGAs*, 1999. 23
- [57] [ONLINE]. **Error Detection and Recovery Using CRC in Altera FPGA Devices.** <http://www.altera.com/literature/an/an357.pdf>. 23
- [58] S.P.PARK D.LEE AND K.ROY. **Soft-Error-Resilient FPGAs Using Built-In 2-D Hamming Product Code.** *IEEE Transactions on Very Large Scale Integration Systems*, 2011. 23
- [59] D.CHEN J.CONG AND Y.FAN. **Low-power high-level synthesis for FPGA architectures.** In *ISLPED*, 2003. 23
- [60] V.K.PRASANNA. **Energy-Efficient Computations on FPGAs.** *Journal of Supercomputing*, **32**(2), 2005. 23
- [61] Y.HU Y.LIN L.HE AND T.TUAN. **Physical Synthesis for FPGA Interconnect Power Reduction by Dual-Vdd Budgeting and Retiming.** *ACM Trans. on Design Automation of Electronic Systems*, **13**(2), 2008. 24
- [62] [ONLINE]. **Stratix IV FPGA ALM Logic Structure's 8-Input Fracturable LUT.** <http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/overview/architecture/stxiv-alm-logic-structure.html>. 24
- [63] [ONLINE]. **VPR and T-VPack 5.0.2 Full CAD Flow for Heterogeneous FPGAs.** <http://www.eecg.utoronto.ca/vpr/>. 24, 77, 89, 165
- [64] M.A.ABDUL-AZIZ AND M.B.TAHOORI. **Soft Error Reliability Aware Placement and Routing for FPGAs.** In *ITC*, 2010. 24

- [65] S.GOLSHAN AND E.BOZORGZADEH. **Single-Event-Upset (SEU) Awareness in FPGA Routing**. In *DAC*, 2007. 24
- [66] O.SHACHAM ET AL. **Rethinking digital design: why design must change**. In *Intl. Symp. on Microarchitecture*, 2010. 25
- [67] G.M.AMDAHL. **Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities**. In *AFIPS*, 1967. 26
- [68] P.KOGGE ET AL. **ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems**. <http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf>. 26, 27, 28, 137, 188, 189, 190, 193, 216, 217
- [69] [ONLINE]. **Future processing : Extreme scale**. http://www.hpcuserforum.com/EU/downloads/DARPAExtremescaleHarrod_ACS_Brief_01709.pdf. 28, 29
- [70] S.MUKHOPADHYAY H.MAHMOODI AND K.ROY. **Modeling of Failure Probability and Statistical Design of SRAM Array for Yield Enhancement in Nanoscaled CMOS**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **24**(12), 2005. 28, 35, 122, 155, 156, 158, 167, 172
- [71] J.L.GUSTAFSON. **Reevaluating Amdahl's Law**. *ACM Communications*, **31**(5), 1988. 29
- [72] S.YEHIA N.CLARK S.A.MAHLKE AND K.FLAUTNER. **Exploring the Design Space of LUTbased Transparent Accelerators**. In *CASES*, 2005. 30, 192
- [73] A.AGARWAL ET AL. **A 320mV-to-1.2V on-die fine-grained reconfigurable fabric for DSP/media accelerators in 32nm CMOS**. In *Intl. Solid-State Circuits Conference*, 2010. 30
- [74] [ONLINE]. **The Berkeley Intelligent RAM (IRAM) Project**. <http://iram.cs.berkeley.edu/>. 31, 37, 192
- [75] T.W.ANDRE ET AL. **A 4-Mb 0.18- μ m 1T1MTJ toggle MRAM with balanced three input sensing scheme and locally mirrored unidirectional write drivers**. In *Intl. Solid-State Circuits Conference*, 2004. 31, 127, 186

- [76] W.Y.CHO ET AL. **A 0.18- μ m 3.0-V 64-Mb nonvolatile phase-transition random access memory (PRAM).** *IEEE Journal of Solid-State Circuits*, **40**, 2005. 32
- [77] S.T.HSU T.LI AND N.AWAYA. **Resistance Random Access Memory Switching Mechanism.** *Applied Physics*, 2007. 32, 186
- [78] T.RUECKES ET AL. **Carbon nanotube-Based Nonvolatile Random Access Memory for Molecular Computing.** *Science*, 2000. 32
- [79] W.WU ET AL. **One-kilobit cross-bar molecular memory circuits at 30-nm half-pitch fabricated by nanoimprint lithography.** *Applied Physics*, 2005. 32
- [80] S.PAUL AND S.BHUNIA. **Computing with Nanoscale Memory: Model and Architecture.** In *Intl. Symp on Nanoscale Architecture*, 2011. 32, 33, 34, 127
- [81] M.R.STAN ET AL. **Molecular Electronics: From Devices and Interconnect to Circuits and Architecture.** *Proceedings of the IEEE*, **91**, 2003. 33
- [82] P.M.KOGGE T.SUNAGA H.MIYATAKA K.KITAMURA E.RETTER. **Combined DRAM and logic chip for massively parallel systems.** In *Conf. on Advanced Research in VLSI*, 1995. 37, 193
- [83] [ONLINE]. **Computational RAM.** <http://www.eecg.toronto.edu/~dunc/cram/>. 37, 192, 193
- [84] A. DEHON. **Reconfigurable Architectures for General-Purpose Computing.** Technical report, MIT, 1996. 38
- [85] T.SASAO S.NAGAYAMA AND J.T.BUTLER. **Numerical Function Generators Using LUT Cascades.** *IEEE Transactions on Computers*, **56**(6), 2007. 38
- [86] J.CONG AND S.XU. **Technology Mapping for FPGAs with Embedded Memory Blocks.** In *Intl. Symp. on FPGAs*, 1998. 38, 39, 42, 43, 126
- [87] S.J.E.WILTON. **SMAP: Heterogeneous Technology Mapping for Area Reduction in FPGAs with Embedded Memory Arrays.** In *Intl. Symp. on FPGAs*, 1998. vii, 38, 39, 42, 43, 126

- [88] [ONLINE]. **Lookup table**. http://en.wikipedia.org/wiki/Lookup_table. 39
- [89] S.PAUL AND S.BHUNIA. **Dynamic Transfer of Computation to Processor Cache for Yield and Reliability Improvement**. *IEEE Transactions on Very Large Scale Integration Systems*, 2011. 39, 48, 141, 143, 146, 147, 153
- [90] S. PAUL AND S. BHUNIA. **Memory based computing: reshaping the fine-grained logic in a reconfigurable framework**. In *Intl. Symposium on Field Programmable Gate Arrays*, 2011. 39, 75
- [91] S.PAUL S.CHATTERJEE S.MUKHOPADHYAY AND S.BHUNIA. **A circuit-software co-design approach for improving EDP in reconfigurable frameworks**. In *ICCAD*, 2009. 39, 119, 122, 123, 124, 125
- [92] S.PAUL S.CHATTERJEE S.MUKHOPADHYAY AND S.BHUNIA. **Nanoscale Reconfigurable Computing Using Non-Volatile 2-D STTRAM Array**. In *Intl. Conf. on Nanotechnology*, 2009. 39, 119, 127
- [93] J.CONG AND S.XU. **Performance-Driven Technology Mapping for Heterogeneous FPGAs**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **19**(11), 2000. 42, 43, 127
- [94] S.TRIMBERGER. **Scheduling Designs into a Time-Multiplexed FPGA**. In *Intl. Symp. on FPGAs*, 1998. 44
- [95] D.JONES AND D.M.LEWIS. **Time-multiplexed FPGA architecture for logic evaluation**. In *Custom Integrated Circuits Conference*, 1995. 44
- [96] G.WU J.LIN AND Y.CHANG. **Generic ILP-Based Approaches for Time-Multiplexed FPGA Partitioning**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **20**(10), 2001. 44
- [97] [ONLINE]. **Mosaic Developing power-efficient coarse-grained reconfigurable architectures and tools**. <http://www.cs.washington.edu/research/projects/lis/www/mosaic/>. 45, 67, 75
- [98] B.V.ESSSEN R.PANDA A.WOOD C.EBELING AND S.HAUCK. **Energy-Efficient Specialization of Functional Units in a Coarse-Grained Reconfigurable Array**. In *Intl. Symp. on FPGAs*, 2011. 45, 67, 76

- [99] M.BUSHNELL AND V.AGARWAL. *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Springer, 2000. 46, 161
- [100] S.CHE J.LI J.W.SHEAFFER K.SKADRON AND J.LACH. **Accelerating Compute-Intensive Applications with GPUs and FPGAs**. In *Symp. on Application Specific Processor*, 2008. 48, 192
- [101] J.DIDO ET AL. **A flexible floating point format for optimizing data-paths and operators in FPGA based DSPs**. In *Intl. Symp. on FPGAs*, 2002. 48
- [102] [ONLINE]. **CACTI 5.1**. <http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html>. 53, 62, 63, 65, 92, 94, 107
- [103] J.H.TSENG AND K.ASANOVIC. **Banked Multiported Register Files for High-Frequency Superscalar Microprocessor**. In *Intl. Symp. on Computer Architecture*, 2003. 55, 56
- [104] J.H.TSENG AND K.ASANOVIC. **A speculative control scheme for an energy-efficient banked register file**. *IEEE Transactions on Computers*, **54**(6), 2005. 55
- [105] R.SANGIREDDY H.KIM AND A.K.SOMANI. **Low-Power High-Performance Reconfigurable Computing Cache Architectures**. *IEEE Transactions on Computers*, **53**(10), 2004. 65
- [106] K.T.PARK ET AL. **A 45nm 4Gb 3-Dimensional Double-Stacked Multi-Level NAND Flash Memory with Shared Bitline Structure**. In *Intl. Solid-State Circuits Conference*, 2008. xi, 65, 68, 200, 207, 208
- [107] J.L.HENNESSY AND D.A.PATTERSON. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007. 74, 188
- [108] E.CHUNG J.HOE AND K.MAI. **CoRAM: An In-Fabric Memory Abstraction for FPGA-based Computing**. In *Intl. Symposium on Field Programmable Gate Arrays*, 2011. 75, 227
- [109] S. FRIEDMAN A. CAROLL B.VAN ESSEN B. YLVISAKER C. EBELLING AND S. HAUCK. **SPR: An Architecture-Adaptive CGRA Mapping Tool**. In *Intl. Symposium on Field Programmable Gate Arrays*, 2009. 75, 192

- [110] D. GRANT C. WANG AND G.G.F. LEMIEUX. **A CAD Framework for MAL-IBU: An FPGA with Time-multiplexed Coarse-Grained Elements.** In *Intl. Symposium on Field Programmable Gate Arrays*, 2011. 75
- [111] K. EGURO AND S. HAUCK. **Resource allocation for coarse-grain fpga development.** *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, **24**(10), 2005. 76
- [112] G.KARYPIS. **Multilevel hypergraph partitioning: applications in vlsi domain.** *IEEE Transactions on VLSI*, **7**(1), 1999. 76, 229
- [113] B. VAN ESSEN A. WOOD A. CAROLL S. FRIEDMAN R. PANDA B. YLVISAKER C. EBELLING AND S. HAUCK. **Static versus scheduled interconnect in coarse-grained reconfigurable arrays.** In *Intl. conf on Field Programmable Logic and Applications*, 2009. 76
- [114] [ONLINE]. **Graphviz - graph visualization software.** www.graphviz.org. 78
- [115] F.D.DINECHIN AND A.TISSERAND. **Multipartite Table Methods.** *IEEE Transactions on Computers*, **54**(3), 2005. 82
- [116] K.K.PARHI. *VLSI Digital Signal Processing Systems: Design and Implementation.* J.Wiley & Sons, 1999. 84
- [117] [ONLINE]. **iFAR – intelligent FPGA Architecture Repository.** <http://www.eecg.utoronto.ca/vpr/architectures/>. 90, 107, 130
- [118] M.HANSEN H.YALCIN AND J.HAYES. **Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering.** *IEEE Design and Test of Computers*, 1999.102
- [119] B.S.AMRUTUR AND M.A.HOROWITZ. **Speed and power scaling of SRAMs.** *IEEE Journal on Solid-State Circuits*, **27**(2), 2002. 107
- [120] [ONLINE]. **Power Modeling for FPGAs.** <http://www.ece.ubc.ca/~stevev/powermodel.html>. 108
- [121] [ONLINE]. **Stratix Family of FPGAs.** altera.com. 112

- [122] T.GOOD AND M.BENAISSA. **692-nW Advanced Encryption Standard (AES) on a 0.13 μ m CMOS.** *IEEE Transactions on Very Large Scale Integration Systems*, 2009. 113
- [123] [ONLINE]. **Secure Hash Algorithm - 1.** <http://en.wikipedia.org/wiki/SHA-1>. 113
- [124] [ONLINE]. <http://vigir.missouri.edu/gdesouza/Research/ColorCCD/ColorInterpolation.pdf>. <http://vigir.missouri.edu/~gdesouza/Research/ColorCCD/ColorInterpolation.pdf>. 113
- [125] U. MEYER-BAESE. *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, 2007. 113
- [126] K.ANDRA C.CHAKRABARTI AND T.ACHARYA. **A VLSI Architecture for Lifting-Based Forward and Inverse Wavelet Transform.** *IEEE Transactions on Signal Processing*, 2002. 113
- [127] [ONLINE]. **Smith-Waterman Algorithm.** http://en.wikipedia.org/wiki/Smith-Waterman_algorithm. 113
- [128] K.NIGAM J.LAFFERTY AND A.MCCALLUM. **Using max entropy for text classification.** In *Intl. Joint Conf. on Artificial Intelligence*, 1999. 113
- [129] J.H.ANDERSON AND F.NAJM. **Active Leakage Power Optimization for FPGAs.** *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **25**(3), 2006. 119
- [130] L.CHANG ET AL. **Stable SRAM cell design for the 32nm node and beyond.** In *Symp. on VLSI technology*, 2005. 122, 166
- [131] J.SINGH J.MATHEW S.P.MOHANTY AND D.K.PRADHAN. **Single Ended Static Random Access Memory for Low-Vdd High-Speed Embedded Systems.** In *VLSI Design*, 2009. 123
- [132] J.HUTCHBY AND M.GARNER. **Assessment of the Potential and Maturity of Selected Emerging Research Memory Technologies.** Technical report, ITRS, 2010. 131

- [133] K.PAGIAMTZIS AND A.SHEIKHOLESAMI. **Content-Addressable Memory (CAM) Circuits and Architectures.** *IEEE Journal on Solid-State Circuits*, 2006. 131, 132
- [134] A.AGARWAL ET AL. **A Dual-Supply 4GHz 13fJ/bit/search 65x128b CAM in 65nm CMOS.** In *ESSCIRC*, 2006. 131
- [135] A.KAVIANI AND S.BROWN. **Hybrid FPGA Architecture.** In *Intl. Symp. on FPGAs*, 1996. 131, 133
- [136] F.HEILE AND A.LEAVER. **Hybrid Product Term and LUT based Architectures Using Embedded Memory Blocks.** In *Intl. Symp. on FPGAs*, 1999. 131
- [137] S.BORKAR. **Microarchitecture and Design Challenges for Gigascale Integration.** In *Intl. Symposium on Microarchitecture*, 2004. 137, 155
- [138] A.AGARWAL ET AL. **A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies.** *IEEE Transactions on Very Large Scale Integration Systems*, **13**(1), 2005. 137, 138
- [139] R.MCGOWEN ET AL. **Power and temperature control on a 90-nm Itanium family processor.** *IEEE Journal of Solid State Circuits*, **41**(1), 2005. 138, 147
- [140] D.BROOKS ET AL. **Dynamic thermal management for high-performance microprocessors.** In *Internal Symposium on High Performance Computer Architecture*, 2001. 138
- [141] D.ERNST ET AL. **Razor: A Low-power Pipeline Based on Circuit-Level Timing Speculation.** In *Intl. Symposium on Microarchitecture*, 2003. 138
- [142] C.T.HUANG ET AL. **Built-in redundancy analysis for memory yield improvement.** *IEEE Transactions on Reliability*, **52**(4), 2003. 138
- [143] D.BROOKS AND M.MARTONOSI. **Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance.** In *Intl. Symposium on High Performance Computer Architecture*, 1999. 139, 145
- [144] S.PALACHARLA N.P.JOUPPI AND J.E.SMITH. **Complexity-Efective Superscalar Processors.** In *Intl. Symposium on Computer Architecture*, 1997. 145

- [145] [ONLINE]. **Simplescalar Toolset 3.0**. <http://www.simplescalar.com/>. 146
- [146] H.LI S.BHUNIA Y.CHEN K.ROY AND T.N.VIJAYKUMAR. **DCG: deterministic clock-gating for low-power microprocessor design**. *IEEE Transactions on Very Large Scale Integration Systems*, **12**(3), 2004. 146
- [147] D.BROOKS V.TIWARI AND M.MARTONOSI. **Wattch: A Framework for Architectural-Level Power Analysis and Optimizations**. In *Intl. Symposium on Computer Architecture*, 2000. 150, 152, 179
- [148] [ONLINE]. **Digital Open Source Hardware**. <http://opencores.org/>. 151
- [149] S.PAUL S.MUKHOPADHYAY AND S.BHUNIA. **A Variation-Aware Preferential Design Approach for Memory Based Reconfigurable Computing**. *IEEE Transactions on Very Large Scale Integration Systems*, 2011. 155, 156, 158, 161, 167
- [150] D.M.KWAI ET AL. **Detection of SRAM Cell Stability by Lowering Array Supply Voltage**. In *Asian Test Symposium*, 2000. 156, 161
- [151] A.PAVLOV ET AL. **Weak Cell Detection in Deep-Submicron SRAMs: A Programmable Detection Technique**. *Journal of Solid State Circuits*, **41**(10), 2006. 156, 161
- [152] N.BANERJEE G.KARAKONSTANTIS AND K.ROY. **Process variation tolerant low power DCT architecture**. In *Design Automation and Test in Europe*, 2007. 161
- [153] N.N.MOJUMDER S.MUKHOPADHYAY J.J.KIM C.T.CHUANG AND K.ROY. **Design and Analysis of a Self-Repairing SRAM with On-Chip Monitor and Compensation Circuitry**. In *VLSI Test Symposium*, 2008. 161
- [154] J.J.KIM K.KIM AND C.T.CHUANG. **Asymmetrical SRAM Cells with Enhanced Read and Write Margins**. In *International Symposium on VLSI Technology Systems and Applications*, 2007. 166
- [155] J.J.KIM S.MUKHOPADHYAY, R.RAO AND C.T.CHUANG. **Capacitive Coupling Based Transient Negative Bit-line Voltage (Tran-NBL) Scheme for Improving Write-ability of SRAM Design in Nanometer Technologies**. In *International Symposium on Circuits and Systems*, 2008. 169

- [156] K.ZHANG ET AL. **A 3-GHz 70Mb SRAM in 65nm CMOS Technology with Integrated Column-Based Dynamic Power Supply.** In *International Solid-State Circuits Conference*, 2005. 169
- [157] J.MAIZ S.HARELAND K.ZHANG AND P.ARMSTRONG. **Characterization of Multi-bit Soft Error events in advanced SRAMs.** In *Intl. Electron Devices Meeting*, 2003. 172
- [158] K.OSADA K.YAMAGUCHI AND Y.SAITOH. **SRAM immunity to cosmic-ray-induced multierrors based on analysis of an induced parasitic bipolar effect.** *IEEE Journal of Solid-State Circuits*, **39**(5), 2004. 172, 180
- [159] C.W.SLAYMAN. **Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations.** *IEEE Trans. on Device and Materials Reliability*, **5**(3), 2005. 172
- [160] D.M.KWAI ET AL. **Detection of SRAM Cell Stability by Lowering Array Supply Voltage.** In *Asian Test Symposium*, 2000. 172
- [161] A.PAVLOV ET AL. **Weak Cell Detection in Deep-Submicron SRAMs: A Programmable Detection Technique.** *IEEE Journal of Solid-State Circuits*, **41**(10), 2006. 172
- [162] S.MUKHOPADHYAY K.KIM H.MAHMOODI AND K.ROY. **Design of a Process Variation Tolerant Self-Repairing SRAM for Yield Enhancement in Nanoscaled CMOS.** *IEEE Journal of Solid-State Circuits*, **42**(6), 2007. 172
- [163] S.RUSU H.MULJONO AND B.CHERKAUER. **Itanium 2 Processor 6M: Higher Frequency and Larger L3 Cache.** In *Intl. Symposium on Microarchitecture*, 2004. 176
- [164] J.L.SHIN B.PETRICK M.SINGH AND A.S.LEON. **Design and Implementation of an Embedded 512-KB Level-2 Cache Subsystem.** *IEEE Journal of Solid State Circuits*, **40**(9), 2005. 176
- [165] J.KIM N.HARDAVELLAS K.MAI B.FALSAFI AND J.C.HOE. **Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding.** In *Intl. Symposium on Microarchitecture*, 2007. 176, 180, 181

- [166] Z.CHISTI A.R.ALAMELDEEN C.WILKERSON W.WU AND S.LU. **Improving Cache Lifetime Reliability at Ultra-low voltages**. In *Intl. Symposium on Microarchitecture*, 2009. 179
- [167] W.J.DALLY ET AL. **Efficient Embedded Computing**. *IEEE Computers*, 41(7), 2008. 189
- [168] P.M.KOGGE. **From Petaflops to Exaflops**. In *Intl. Supercomputing Conference*, 2008. 189, 190
- [169] [ONLINE]. **GPU Technology from NVIDIA**. <http://www.nvidia.com/page/home.html>. 191
- [170] [ONLINE]. **CUDA - NVIDIA's parallel computing architecture**. http://www.nvidia.com/object/cuda_home_new.html. 191, 212
- [171] W.J.DALLY. **The end of denial architecture and the rise of throughput computing**. In *DAC*, 2009. 191
- [172] [ONLINE]. **Thermal dissipated power in GPUs**. <http://www.geeks3d.com/20090618/graphics-cards-thermal-design-power-tdp-database/>. 191
- [173] R.LYSECKY G. STITT AND F. VAHID. **Warp Processors**. *IEEE Transactions on Design Automation of Electronic Systems*, 2006. 192
- [174] B.BETKAoui D.B.THOMAS AND W.LUK. **Comparing Performance and Energy Efficiency of FPGAs and GPUs for High Productivity Computing**. In *FPT*, 2010. 192
- [175] T.L.STERLING AND H.P.ZIMA. **Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing**. In *Intl. Supercomputing Conference*, 2002. 193
- [176] S.PERISSAKIS Y.JOO J.AHN A.DEHON AND J.WAWRZYNEK. **Embedded DRAM for a Reconfigurable Array**. In *Intl. Symp on VLSI Circuits*, 1999. 193
- [177] D.H.WOO AND H.S.LEE. **Extending Amdahls Law for Energy-Efficient Computing in the Many-Core Era**. *IEEE Computers*, 2008. 193

- [178] M.D.HILL AND M.R.MARTY. **Amdahl's Law in the Multicore Era.** *IEEE Computers*, 2008. 193
- [179] K.KANDA ET AL. **A 120mm² 16Gb 4 Multi Level NAND Flash Memory with 43nm CMOS technology.** In *Intl. Solid-State Circuits Conference*, 2008. 200, 211
- [180] V.MOHAN S.GURUMURTHI AND M.R.STAN. **FlashPower: A detailed power model for NAND flash memory.** In *DATE*, 2010. 202, 208, 209
- [181] [ONLINE]. **Micron 1Gb NAND Flash Data Sheet.** http://download.micron.com/pdf/datasheets/flash/nand/1gb_nand_m48a.pdf. 202, 205, 209
- [182] A.N.UDIPI ET AL. **Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores.** In *Intl. Symp. on Computer Architecture*, 2010. 204
- [183] L.BAUER M.SHAFIQUE S.KRAMER AND J.HENKEL. **RISPP: Rotating Instruction Set Processing Platform.** In *DAC*, 2007. 212
- [184] [ONLINE]. **Implementing Barrel Shifters Using Multipliers.** http://www.xilinx.com/support/documentation/application_notes/xapp195.pdf. 226
- [185] J.CONG AND Y.DING. **On area/depth trade-off in lut-based FPGA technology mapping.** In *DAC*, 1993. 229