I, Krishna Kumar Rangan, hereby submit this as part of the requirements for the degree of:

Master of Science in:
Computer Engineering

It is entitled:
A M-SIMD Intelligent Memory

Approved by:
Dr. Philip A. Wilsey
Dr. Nael B. Abu-Ghazaleh
Dr. Harold W. Carter
A M-SIMD Intelligent Memory

A thesis submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of
Electrical & Computer Engineering and Computer Science
of the College of Engineering

March 20, 2001

by

Krishna Kumar Rangan
Bachelor of Engineering, University of Madras, Madras, India, 1998

Committee Chair: Dr. Philip A. Wilsey
Abstract

The gap between the speed of logic and DRAM access is widening. Existing solutions are stop-gap measures; cache miss rates are becoming higher than the rate at which memory can provide data. Furthermore, trends in applications suggest that future workloads will be more data-intensive. A potential solution to the memory bottleneck problem is the integration of DRAM and logic on the same die. Such solutions are motivated by the following: (i) the bandwidth available within the chip is many orders of magnitude higher than that at the memory bus at a significantly lower access time and with lower power dissipation; and (ii) as typical workloads shift towards data-intensive-multimedia applications, the wide bandwidth can be effectively utilized. However, several issues have to be understood and addressed when logic is implemented together with memory on a single die. In addition, there are difficult challenges in developing architectures and programming models that expose the available bandwidth to end users.

This thesis presents the design of an intelligent memory based on a distributed data-parallel architecture with limited support for control parallelism (called PPIM). It investigates relevant design issues (for example, at the interface between the logic and memory) and the success of such an architecture in supporting data-intensive applications. The design is evaluated as a stand-alone system, and also as a co-processor acting as a memory access filter – applications are partitioned to execute parallel portions on the co-processor, while executing irregular portions on the superscalar processor. A cycle-accurate simulator is developed and used to study the performance of the architecture for some real data-intensive applications. The performance is compared against that of a modern superscalar processor (simulated using the simplescalar tool-set).
To my Grandparents and parents
Acknowledgements

I wish to thank my advisor Dr. Philip A. Wilsey for the opportunity to work under his guidance, for his encouragement and support. I thank Dr. Nael B. Abu-Ghazaleh for his help and guidance throughout this work. I thank Dr. Harold Carter for being on my committee.

Special thanks to my colleagues in Experimental Computing Laboratory and friends in UC for making my graduate life a memorable experience.
# Contents

List of Tables iii

List of Figures iv

1 Introduction 1

1.1 Motivation ......................................................... 4
1.2 Research Goals ................................................... 6
1.3 Organization of Thesis ............................................ 7

2 Background and Related Work 9

2.1 Solutions within Traditional Architectures .................. 10
2.1.1 Processor Side Solutions to the Memory Bottleneck .... 10
2.1.2 DRAM Architecture and High Performance Memories .... 13
2.2 Solutions using Integrated Logic and DRAM ................ 15

3 PPIM Processor 22

3.1 Introduction ....................................................... 22
3.2 PE Organization .................................................. 24
3.3 Controller Organization ........................................... 28
3.3.1 Pipeline ......................................................... 29
3.4 Communication subsystem ...................................... 34
3.5 Address Translator ............................................... 35
3.6 Chip Area .......................................................... 37
3.7 Summary ........................................................... 38

4 Instruction Set Architecture and Programming model 39

4.1 Instruction Set Architecture ...................................... 39
4.2 Programming Model ............................................... 41

5 PPIM Simulation 44

5.1 Introduction ....................................................... 44
5.2 Related Architectural Simulators ................................ 45
5.3 Simulator Overview ............................................... 47
5.4 Simulator Organization ............................................. 51
   5.4.1 Pipeline ................................................. 52
   5.4.2 Verification ............................................. 55
5.5 Summary ......................................................... 56

6 Applications ....................................................... 58

7 Experiments ....................................................... 64
   7.1 Developing Applications for PPIM .......................... 64
   7.2 PPIM as a Processor ....................................... 66
   7.3 PPIM as co-processor ..................................... 70
   7.4 Summary ..................................................... 73

8 Conclusions and Future Work .................................. 75
   8.1 Conclusion .................................................. 76
   8.2 Future work ................................................ 77

Bibliography ......................................................... 80

A ................................................................. 87
   A.1 Experiments on \textit{ppim-sim} ........................... 87
   A.2 Events within the Simulator .............................. 92

A ................................................................. 94
   A.1 PPIM as a Processor ...................................... 94
       A.1.1 Number Of Cycles .................................. 94
   A.2 PPIM as co-processor ................................... 95
       A.2.1 Memory Traffic .................................... 95
       A.2.2 Number Of Cycles .................................. 96
List of Tables

7.1 Number of cycles taken by applications executing on a pure SIMD and two controller configurations of the PPIM processor. .......................... 68
7.2 Number of cycles taken by applications executing on a pure SIMD and four controller configurations of the PPIM processor. .......................... 69

A.1 Events used within *ppim-sim* simulator ................................. 93
A.1 Image Segmentation .......................................................... 94
A.2 Query Handling .............................................................. 94
A.3 Contour Extraction .......................................................... 95
A.4 Fault Simulation ............................................................ 95
A.5 Image Segmentation .......................................................... 95
A.6 Query Handling .............................................................. 96
A.7 Image Segmentation .......................................................... 96
A.8 Query Handling .............................................................. 96
List of Figures

2.1 Conventional DRAM Organization (A DRAM cell is shown in the box) 13
3.1 PPIM Processor ................................. 23
3.2 PE Block Diagram ................................. 25
3.3 Processing elements incorporated in an Unit array. Architectural configuration shown for 1024 byte wide unit array and a pitch match ratio of 1:4. ................................. 26
3.4 Controller Pipeline ................................. 30
3.5 Translation of address generated by superscalar processor to PPIM internal address, illustrated using a 2-D image array mapping. ................................. 36
4.1 Executing iteration statements (within SIMD block) when loop terminating condition depends on a scalar, compile time value. ............ 40
4.2 Existing iteration statements (within SIMD block) when loop terminating condition depends on a vector variable. ..................... 41
4.3 PPIM Programming Model ............................... 42
5.1 PPIM Simulator Overview ............................... 48
5.2 Virtual memory layout in a PPIM executable ............................. 50
5.3 Structure of main loop in ppim-sim simulator ............................. 53
6.1 Mapping of a 2-D image onto the PE mesh ............................. 60
6.2 Sample Network and mapping of faults to PEs ............................. 62
7.1 Cycle count on Superscalar and PPIM processor ............................. 65
7.2 Cycle count on Superscalar and PPIM processor ............................. 66
7.3 Expected Vs. Observed speedup (best and worst cases) ..................... 67
7.4 Reduction in memory traffic with PPIM co-processor ............................. 71
7.5 Cycle counts without and with PPIM co-processor ............................. 73
A.1 Simulator run time for increasing number of PEs ............................. 88
A.2 Simulator run time for increasing number of PEs ............................. 89
A.3 Simulator and simulation process peak memory usage ............................. 90
A.4 Simulator and simulation process peak memory usage ............. 91
A.5 Simulation time in sim-outorder and ppim-sim ..................... 91
Chapter 1

Introduction

VLSI technology continues to develop at a staggering rate presenting two challenges to computer designers: (i) how to capitalize on the additional resources that are available on a chip; and (ii) how to evolve computer architecture models that are well matched to the significantly changed physical parameters of deep sub-micron technology and the expanding needs of applications. One of the chief challenges is overcoming the widening gap between processor and DRAM memory (whereas processor speed has historically grown at 60% a year, DRAM speed has only improved 7% a year). Caches are increasingly used to bridge this speed mismatch. Consequently, multi-level caches and on-chip caches account for 50 - 90% of the transistor counts of modern processors. Even allowing for such large caches, memory will not be able to provide data to the processor at a rate that satisfies cache miss rates [27, 31, 74]. Furthermore, most complexities of modern processors, such as speculative instruc-
tion pre-fetching and out-of-order execution are targeted to hide the memory latency. These aggressive latency tolerance techniques expose the limitations in memory bandwidth [6]: even if an ideal memory access predictor is used to prefetch memory into cache, the raw number of memory requests generated overwhelms the capabilities of the memory system. This trend is also accelerated by the changing nature of workloads; data-intensive multimedia workloads form a major and increasing portion of computer workloads [11, 36]. In order to reflect this change, 12 out of the 26 programs making up the newly released SPEC’2000 benchmark suite have a resident memory size of over 100 MBytes (10 over 150MBytes) [20].

The mismatch in memory latency/bandwidth has been well studied using separate processor and memory side improvements. Alternatively, a tightly knit processor and main memory (called Dynamic RAM or DRAM) organization (such as, on a single chip) can reduce the memory bottleneck. The main objection to a single chip logic-memory processor is the fact that the DRAM processes were not “ready” to take the high-speed logic; in other words, introducing logic with DRAM on a memory chip reduces the yield, increases the cost per Megabyte of memory, impairs DRAM performance and affects the logic speed. Recently, Mitsubishi demonstrated a 32-bit RISC logic core (having 83 instructions) and an embedded DRAM processor [43]. It was fabricated in a Hyper-DRAM (mixed logic and DRAM), 0.5 \( \mu m \) technological process [43]. Much more recent 0.15 \( \mu m \) embedded DRAM [71, 23] technologies are competitive with conventional logic process. Accordingly, it can be identified that
integrating logic and memory on a single chip is a feasible task based on the modern technological characteristics. Moreover, such integrated architectures are attractive in terms of self-testability [58].

Integration of logic and memory on a chip eliminates the overhead caused by memory and cache controllers; the result is a low latency interface coupled with increased bandwidth provided by wider internal memory buses. However, there are significant challenges in developing architectures and programming models that expose the available bandwidth to the application and operate efficiently within an integrated logic and memory process. In addition, there are difficult design decisions at the interface between the logic and DRAM. It is clear that parallel processing is required to capitalize on the effective bandwidth. Vector/SIMD approaches makes effective use of the bandwidth at low overhead (instruction space and control logic overhead) for supporting the parallelism model. Moreover, they provide a good match to multimedia/streaming applications in many situations. However, studies have shown that a limited amount of control parallelism exists in most applications [3]. Hence, the following questions arise - How do we effectively support both data and control parallelism present in the applications taking into account the mixed process constraints? Can we do some or all of the processing in memory? How much memory traffic can we eliminate by processing in memory? These are precisely the questions we have investigated in this thesis. In this work, we present the design of an intelligent memory based on a distributed data-parallel architecture with limited support
for control parallelism. In developing this design, several architectural design issues at the interface between the logic and memory are addressed. A cycle-accurate simulator is developed to study the architecture allowing a comparison of its performance on some data-intensive applications against that of a modern superscalar processor. The simulator is also used to study the success of the architecture as a memory access filter to superscalar processors. The performance gains achieved by processing in memory are presented.

1.1 Motivation

The growing gap in the performance between processing logic and DRAM, and the changing nature of “typical” applications severely challenge conventional computer architecture. Integrating processing and memory on the same chip offer potential relief for this problem. The arguments for integrating logic and memory are compelling in light of current technology trends [33]:

1. the aggregate bandwidth within the chip is a few orders of magnitude higher than that at the I/O pads and system bus (restricted by the row bandwidth at the sense amplifiers rather than the chip interface/system bus width). More specifically, the available bandwidth within the chip is on the order of a few Terabytes per second, compared to a few hundred Megabytes per second on current system buses.
2. the access latency is significantly decreased; much of the latency is due to delays outside of the DRAM chip itself (for the AlphaServer 8200, using an Alpha 21264 microprocessor, DRAM latency accounted for only 18 of the 76 CPU cycle latency of the memory system; address multiplexing, memory controller overhead and inter-chip communication accounted for the other 58 cycles [45]).

3. the power consumption of DRAMs is much lower than SRAM caches [14]. The increasing importance of wireless computing places a premium on low-power high performance architectures [25, 68].

4. because of the available data parallelism in multimedia workloads [11, 36], data-parallel extensions to instruction set architectures (ISAs) of modern microprocessors have appeared (for example Intel’s MMX [48] and AMD’s 3D now [53]). These extensions are successful in speeding up multimedia workloads [50], but are limited by the width of the data path on the microprocessor. Integrating processing with memory capitalizes primarily on data-parallelism, with the ability to support much wider data paths. When DRAM memory is accessed, an entire row corresponding to the given address is read into the sense amplifiers.

5. the current technology trends facilitate system-on-a-chip designs. Traditionally, logic implemented in DRAM process occupies more area and also suffers in terms of speed. This is because, historically, faster logic and high density DRAM processes were incompatible. However, recent merged technological processes
supports efficient implementation of logic and DRAM cells in the same chip. Embedded DRAM [71, 23], Hyper-DRAM technologies [43] offer advantages of wider bandwidth, reduced latency, low power consumption and self-testability without compromising high-performance making it suitable for system-on-a-chip designs. The mixed process supports logic speeds of over 400 MHz and on-chip memory speeds of over 500 MHz [71]. Hence, in the light of current technological process, integrating logic-memory on the same die is a compelling approach to solve the memory bottleneck problem.

1.2 Research Goals

Processor-memory speed mismatch and limited bandwidth available at the pins of the DRAM chip form a more pronounced bottleneck on applications that work on large data-sets. The chief goal of the research study is to alleviate the effects of these bottlenecks on data-intensive applications. In particular, we investigate if a control-parallel, SIMD architecture implemented in memory is conducive to applications that crunch large amounts of data. Since the future workloads are more likely to be data-intensive, it will be of great importance in understanding if intelligent memory processors are effective in improving the application performance, while also reducing the memory traffic. The following sub-goals were identified to show the feasibility of this approach.
• Identify domains that are data-intensive and develop applications for study.

• Understand the constraints when attempting to implement logic and memory on the same chip.

• Design an architecture taking into account the characteristics of data-intensive applications and mixed-mode technological process constraints.

• Develop a simulator to study the architecture for executing such applications.

• Validate the results experimentally by comparing the result of the simulation with that of an existing superscalar processor simulator.

• Analyze the performance gains achieved by implementing a processor in memory.

In the light of the work presented in this thesis, intelligent memories show a promising, alternate direction for architectural designs and has the potential for further research considerations.

1.3 Organization of Thesis

The remainder of this thesis is organized as follows.

• Chapter 2 provides an overview of improvements in the processor and memory over the past years. The chapter also discusses related work that target the memory bottleneck problem using an integrated processor memory approach.
• **Chapter 3** discusses the architecture of PPIM as a stand-alone processor and as a co-processor to superscalar processors (memory organization, Processing Elements (PEs), controllers and the communication system) and the various design issues.

• **Chapter 4** presents the Instruction Set Architecture and the programming model of the processor.

• **Chapter 5** explains the cycle-accurate simulator (called *ppim-sim*) that models the PPIM processor and the issues involved in its design and development.

• **Chapter 6** elaborates on the different data-intensive applications that have been developed to study on the PPIM processor.

• **Chapter 7** presents various experiments performed to evaluate the performance of PPIM processor. A comparative analysis of the simulation results obtained from a superscalar processor simulator are also presented. It also presents experimental evaluation of reduction in memory traffic using PPIM co-processor.

• **Chapter 8** summarizes on the results of PPIM processor. The scalability and the extensibility of the design is discussed. It also discusses a few directions for future research.
Chapter 2

Background and Related Work

A traditional computing system consists of processing logic and an external memory where data and instructions are stored. Data and instructions are brought to the processor using a memory bus. The computed results are then stored back to memory. This separation of logic and memory has many advantages. Firstly, the devices can be fabricated according to their needs. Processor fabrication translates to having fast transistors for fast logic and having many metal layers to accelerate communication and to simplify power dissipation. On the other hand, DRAM memory fabrication is optimized to have small DRAM cells and low current leakage. Secondly, the separation also provides the advantage of scaling the memory chips independent of the number of processors. There are disadvantages as well because of this separation. Memories are external devices and suffer delays due to memory access latency. Furthermore, the bandwidth visible on the system bus is limited. This
von-Neumann memory bottleneck due to the gap between processing rate and off-chip memory access latency/limited bandwidth severely cripple the modern microprocessors. Several efforts have focussed on addressing this problem. More specifically, the approaches are twofold: (i) improvements in traditional architectures, such as advancements in processor and DRAM side. (ii) integrated logic-memory processor solutions. This chapter reviews such efforts targeted towards addressing the memory bottleneck problem.

2.1 Solutions within Traditional Architectures

Numerous memory access latency hiding techniques are in effect on modern superscalar processors. On the other side, research has been going on for improving the performance of memory subsystem, by improving the basic DRAM architecture and the performance of memory bus.

2.1.1 Processor Side Solutions to the Memory Bottleneck

One of the earliest attempts to hide the growing mismatch in processor and memory speed is the use of caches in microprocessors. The technique exploits the spatial and temporal locality in memory accesses of running programs by caching the frequently used instructions/data in an high-speed local memory (cache) [46]. References reach the physical memory only if the data is not found in the cache.
Caches have been well studied for many years and numerous body of literature exists in cache organization, design options and optimizations [8, 18, 46, 63]. Indeed, sophisticated cache designs have been highly successful in hiding the memory bottleneck until recently [74]. The continued trend of logic switching speed improving faster than the memory access time coupled with the nature of streaming applications \(^1\) (that are atypical of present day applications) have severely challenged the cache benefits. Therefore, alternative solutions are sought to hide the mismatch in latency and bandwidth. Non-blocking/split transaction caches allow multiple cache misses to be serviced concurrently [34]. In the time to service the miss, instructions not dependent on the fetch are allowed to execute. The most obvious technique to improve the latency is to increase the size of cache and to add additional levels of off-chip caches. Not surprisingly, caches occupy 50 - 90% of the transistor counts of modern microprocessors.

An alternative solution to reduce miss penalty and latency in caches is to use hardware and compiler controlled pre-fetching methods. Pre-fetching techniques attempt to utilize the memory bandwidth that otherwise would be wasted when memory is not in use. For example, pre-fetching hardware in Alpha AXP 21064 processor fetches the requested block and the next consecutive block [46]. While the requested block is placed in the cache, the anticipated block is placed in an instruction stream buffer. With 16 blocks in the instruction buffer, the hit rate improved to 72%

\(^1\)streaming applications such as audio and video playback typically do not exhibit data reuse (locality)
for a 4-KB direct mapped instruction cache with 16 byte blocks [26]. A similar pre-fetching approach for data accesses is also shown to be effective in reducing the latency [26]. Software pre-fetching employs compiler techniques, such as unrolling the loop and determining the future memory accesses. It has the advantage of not pre-fetching data that may not be used. An analysis of instruction traces showed that prefetching 10% of the loads can eliminate 73 - 99% of latency penalties for some workloads [1]. However, compiler controlled pre-fetching has a price of increased code size resulting from software overhead [29, 39]. Despite its advantage in reducing the latency, pre-fetching techniques may actually affect the performance when it locks up the memory bandwidth, while actual cache misses occur in an application. In fact, the latency tolerance techniques such as prefetching expose the bandwidth “wall” of memory, that arguably is a more fundamental limit to the performance [6].

Multi-threaded architectures switch context among different threads in every cycle. If a thread has a cache miss, the latency will be hidden as other caches are taking turns executing. The latency can be effectively reduced to one cycle if sufficient threads exist [67]. However, expensive support for hardware contexts is needed; moreover, while latency is hidden by executing other threads, bandwidth is not reduced. Bandwidth requirements are also increased by instruction level parallelism as the additional instructions executed per cycle increase the number of references to memory.
Figure 2.1: Conventional DRAM Organization (A DRAM cell is shown in the box)

2.1.2 DRAM Architecture and High Performance Memories

Figure 2.1 shows a conventional DRAM chip organization. Each DRAM cell consists of a single transistor and capacitor as shown in the inner box of figure. When enabled, the transistor connects the capacitor to the bit-line that is precharged on reads. DRAMs multiplex row and the column address to reduce the pins required to read the address. Hence, first the row address is provided to DRAM and the Row Address Strobe (RAS) is asserted; the address enables the transistors in the full row. The charge on the capacitor causes a small swing in the bit-line’s voltage level; the sense amplifiers are sensitive to detect the swing and the data is latched in a SRAM buffer (also called page). Note that huge-bandwidth is available at this SRAM buffer stage. Since reads are destructive, data must be written back to the cells. After the row is read into the sense amplifiers, the memory controller sets the Column Address Strobe (CAS) to zero. The selected word is then read into the output buffers of the
chip. The output buffers of the all the accessed memory chips are then fed into the data lines of the memory bus. The charge on the capacitors drain with time; hence, the memory is refreshed by reading a row at a time and writing back the contents. An integrated logic-memory processor requires more frequent refreshing, because the noise introduced by the high speed logic drains the charge on the capacitors more quickly.

Improvements in DRAM technology such as Fast Page Mode (FPM) [56], Extended Data Out (EDO) [21] and Synchronous DRAM (SDRAM/ESDRAM) [22] increase the memory bandwidth by allowing accesses to the same page (the row currently in the sense-amplifier latches) to proceed quickly; pipelining is used in EDO and SDRAM; SDRAM outputs successive elements synchronously eliminating some of the handshaking; ESDRAM pipelines reads from the cached page with the read of a different row. Performance studies of memory organizations show that the above optimizations improve memory bandwidth by a factor of three as compared to the earlier DRAM architectures, but not the latency component [10]. The factors that effect the DRAM latency are: (i) address decode time, (ii) word line delay, (iii) bit sensing delay and (iv) output driving delay. Out of the 53ns latency of a typical DRAM, these four components took 7ns, 23ns, 16ns and 7ns respectively [66]. All four components can be reduced using optimized circuits [64, 66, 41, 65]. However, the optimizations trades off density for latency and increases the chip area. Circuit techniques such as, partial cell activation can be used to reduce the power consumption as well [64].
The performance of the memory subsystem is a function of the performance of DRAMs as well as the memory bus performance. Often, it is the memory bus that causes the differences in the performance of memory organizations [57]. The current industry standard SDRAM bus runs at 133MHz; it is being challenged by high speed bus technologies such as Rambus’ RDRAM [49] and SLDRAM [61, 62]. These buses use novel impedance matched, low-voltage bus technology that allows them to be clocked an order of magnitude faster than SDRAM buses. As can be expected, these buses perform similar to SDRAM at low loads (limited by DRAM latency), but have much better performance under high loads [57].

2.2 Solutions using Integrated Logic and DRAM

The idea of integrating logic and memory is not new. Early processors such as the Inmos Transputer had DRAM integrated with the processor [69]. Other SIMD machines were built as smart or associative memories. More recently, integrating a single super-scalar processor with DRAM has been investigated by several studies and shown to be competitive with the traditional model with the current technological process [45, 55]. Although the processor logic is significantly slower when built in the DRAM process, high computation to memory access ratio in irregular applications makes up for the difference in speed [45].

With the integration of processing and memory, the full bandwidth of the DRAM becomes available to the processing logic at low latency (limited by the row
width at the sense amplifiers). An analysis using contemporary DRAMs showed that stream of address that miss the L2 cache contains significant amount of locality as measured by the hit rate in the row buffers [10]. However, there are some difficult problems that have to be solved before the full potential of intelligent memories is realized. For example, an important question is - how to exploit the available bandwidth? It is intuitive that to take advantage of a substantial portion of the bandwidth, parallel processing must be used. There are several proposals ranging from using static low-level data/stream parallelism as per the vector/SIMD model [45, 51] to high-granularity full processor “tiles” with associated memory and reconfigurable communication paths [40, 73], to using only implicit instruction level parallelism (using the bandwidth for wide fetches of cache line) [55], as well as others [13, 16, 37, 38, 51, 59, 73]. A brief description of some of these projects is presented in the remainder of this chapter.

**Vector IRAM** implements a vector processor in memory. It uses the available on-chip area to provide more DRAM memory instead of SRAM. With 16 1024-bit wide memory ports, a collective bandwidth of 100 GB/sec is achieved [45]. Since vector instructions access memory with a regular pattern, multiple memory banks can supply data concurrently. Accordingly, a significant throughput is achieved using 8 concurrent vector units. However, the utilized bandwidth is limited to that consumed by these units; adding additional units is expensive because it requires more ports on the vector register set [32].
The Imagine project developed multimedia processor in memory architecture [51]. Imagine operates on “stream” data that is efficiently loaded to the logic through a shared stream register file. Streams are pipelined through SIMD controlled clusters (with one functional unit per cluster). Each cluster is assigned to an image processing kernel and streams are passed from cluster to cluster to achieve their computation. An interesting feature of imagine is the high-speed I/O that is matched to the streaming rate using SDRAM buffers.

The Computational RAM uses SIMD architecture for the integrated processor-memory system [13]. Processing elements are small, single bit ALUs operating on a bit at a time. They are pitch-matched to the memory columns of DRAM array to effectively utilize the memory bandwidth available at the sense amplifier level. The inter-processing element communication network is restrictive due to limitations of the layout. PPIM architectural approach is similar to the Computational RAM project in that the processing element is pitch-matched to a fixed number of a columns in the DRAM array.

The FlexRAM project presents numerous compute engines interleaved with DRAM macros, a RISC superscalar processor and cache on a chip [28]. The RISC processor is used to co-ordinate the compute engines and to take some load off the host processor. The FlexRAM chip defaults to a DRAM chip when an application does not use it. Experiments conducted 4 FlexRAM chip system show that the workstation runs 25-40 times faster. Off-chip communication (among multiple
commodity FlexRAM chips) is required to exploit the control threads present in the application.

**The Terasys PIM array** integrates a SIMD array into the architecture of a high performance computer [15]. In their design, sun spare-2 processor is interfaced to one or more PIM units through a Terasys interface board. The SIMD PIM array could either be used as a processor array or as a conventional memory. Processing elements are single bit ALUs; operations are sent as memory writes through the interface board, and cause the ALUs to perform the operation at the specified row address across all the columns of memory. Unlike CRAM, the interprocessor communication is performed using host computer memory.

The architectures discussed so far target low-level data parallelism. A dual approach uses tiles of complete processors. The advantages of this approach is the naturally tiled (and therefore localized) floor-plan, and the ability to allocate complete memory banks to each processor (rather than having PEs share the same banks as is necessary for low granularity processors). However, a primary disadvantage is the relatively low bandwidth utilized due to the necessarily smaller degree of parallelism and the overhead for a general parallel architecture.

**The PPRAM project** defines a high-level standard for integrating active memories into a system [40]. Processing elements actually are standard RISC processors with megabytes of DRAM, cache SRAM and PPRAM link interface. Their prototype design consists of 4 RISC processors each with 8 MB of DRAM and 24Kb
cache SRAM; DRAM and cache memory are interconnected with 1Kb signal lines achieving a bandwidth of 25 Gb/s per processor element. Communication among processors is through standard PPRAM link and through a shared global register file. While the PPRAM link provides an high bandwidth communication, the global register file provides a low latency communication and synchronization among processors on same chip [40]. Communication among different implementations of PPRAM nodes (processing elements) is made possible using a common communication protocol.

**M32R/D Microprocessor** represents an earliest integrated logic and DRAM processor fabricated and sold in volume. The architectural design is similar to a standard RISC processor with embedded DRAM; however, integrated approach enables the use of a wide 128 bit bus for cache and instruction queue fills [43]. The processor and memory are operated by a 66.6 MHz, while the connection to the peripheral circuits is clocked at 16.67 MHz. The processor is fabricated using a merged logic and DRAM process called the Hyper DRAM.

**The single chip multiprocessor integrated with DRAM** has four processors and performs an average of 52% faster than the SRAM based architecture on floating point applications with large working sets [75]. In contrast, a similar uniprocessor integrated with DRAM gives only 4% improvement over conventional architecture [75]. Each computational core consists of a two way superscalar processor, with each processor having a separate instruction and data L1 cache. They
are connected with 256 bit wide read/replace data bus, whose width is the same as cache line size. The memory design consists of 32 MB memory banks with 128 bits of memory bus width. The whole architecture runs at a speed of 500 MHz. Cache coherency is achieved by using the update coherence protocol.

The Smart Memories reconfigurable architectural design consists of an array of smart memory tiles that could be configured as a processing tile or as a DRAM block [37]. Each tile consists of a reconfigurable memory, 64-bit processor with reconfigurable instruction format and interconnect. The user programs the tile to match the structure of the application. The reconfigurable approach is promising because it allows customizing the processor to the application – a feature that is useful in multimedia computations. It runs the Imagine [51] and Hydra [19] benchmarks (after the architecture is re-configured into these designs) only with modest performance degradation.

Active pages is representative of using both re-configurable logic and DRAM memory for intelligent memory processor [44]. Similar to FlexRAM approach, Active pages partition the computation between processor and active page intelligent memory. The computation, however, is performed by loading the data and the associated functions that operate on it to active pages.

The IBM Execube (with 8 hypercube interconnected processors and DRAM on each chip) is an early example of the above approach [30], while the Smart Memories project is a recent example [37]. The RAW project can also be classified in this
category, although it uses reconfiguration and “compiler place and route” to specialize the hardware to the application [73]. This approach is promising because it allows customizing the processor to the application – a feature that is useful in multimedia computations. Other proposals for reconfigurable intelligent memory architectures for multimedia applications exist [16].
Chapter 3

PPIM Processor

3.1 Introduction

The intuition behind our design is to provide support for the emerging data intensive workloads in a way that is compatible with the parameters of a processor in memory configuration. Typical workloads are shifting towards multimedia/data-intensive applications [11, 36], that demonstrate high degree of data parallelism and limited amount of control parallelism [3, 12]. Vector/SIMD approaches are best candidates for low overhead, fine grained data-parallel architectures. On the other hand, MIMD configurations that are suited for exploiting control parallelism comes with large implementation overheads. To best utilize the bandwidth within the memory chip, moreover, requires a close coupling of logic with the memory. While the simpler SIMD-type processing elements can be pitch-matched to “read” the data directly
from the DRAM memory, greater replication of the logic is necessary to maximally exploit the integration. On the other hand, a MIMD design space trades-off parallelism granularity for programming flexibility. Hence, it becomes critical to evaluate the benefit of the model against the the cost of supporting it.

PPIM is a distributed Multiple-SIMD architecture. It consists of a small number of controllers that broadcast instructions to all the PEs. Every PE can choose (based on the results of local conditions) to receive its control from any of the controllers. Thus, the parallelism model is a balance between overhead and flexibility. PPIM primarily aims to exploit the data parallelism (which is the primary source of parallelism in data-intensive applications - for example, CAD, database and scientific applications) but offers limited support for control parallelism present in the applications. While traditional SIMD architectures can efficiently handle iterative constructs, PPIM processor provides support for other structures that affect the program flow such as if-then-else and case statements using limited amount of controllers. During execution, when code takes multiple execution paths, additional controllers are forked (described in chapter 4).
The interface between the memory and logic is best suited to data-parallelism (especially if multiple PEs are matched to a single DRAM bank). The PPIM architecture consists of multiple SIMD controllers and PEs (4 and 1024 respectively in the current configuration). With controllers residing on each chip, the design can be scaled. Each controller in the PPIM processor has a memory of 16 MBits. In addition, each PE has a memory of 256 KBits. Figure 3.1 shows the architectural overview of the PPIM processor. The Address translator unit is used only when PPIM is used as a co-processor to superscalar processors (discussed in section 3.5).

3.2 PE Organization

While designing a PE system, two primary design issues need to be considered. The issues are matching the DRAM memory organization to the access patterns required by the PPIM parallel model, and bridging the speed mismatch between DRAM access and logic speed. The PEs are pitch-matched to different columns of the DRAM array similar to the CRAM model [13]. By placing the processing elements close to the sense amplifiers, the large bandwidth is exposed to the PEs, and the distance the data has to travel to get to the logic is minimized. Furthermore, the amount of bandwidth available at the sense amplifiers is in the order of terabytes per second. However, there is a cost/benefit tradeoff: by placing the PEs close to the sense amplifiers, more PEs are needed to “cover” the memory. Thus, only the most common/least expensive functionality must be supported at the lowest level.
The PEs could be listening to any of the multiple controllers; they may address different memory rows in the same cycle. In such cases, access to memory has to be serialized resulting in performance degradation. One solution to this problem is to use multiple memory banks. However, the overhead of having a bank per PE is expensive since it involves extra addressing logic per bank. Another solution is to have multiple ports per bank for concurrent reads and writes – this is also expensive. Third solution is to implement software multi-porting i.e., access to different memory banks are serviced concurrently. If the access are destined to the same memory bank, they are serialized through a centralized (at the controller) interlock mechanism. Thus, a memory bank (also called an unit array) is shared between a small number of PEs. Each PE has its share in the memory, as each PE is mapped to different column
Figure 3.3: Processing elements incorporated in an *Unit array*. Architectural configuration shown for 1024 byte wide unit array and a pitch match ratio of 1:4.

slices in the unit array. This is illustrated in figure 3.3. In the current configuration of PPIM, row width of the unit array is 1024 bytes and 4 PEs share an unit array (pitch match ratio of 1:4).

A small SRAM buffer is used in each PE to minimize the effect of serializing access to different memory rows, as well as to hide the mismatch between logic speed and DRAM memory access speed. The number of bits in the SRAM buffer is the same as the number of bits in a single row of the unit array. When an address is given to the unit array, an entire row is fetched into the SRAM buffer. Hence, this buffer can be viewed as a single line cache for further memory accesses. With a pitch match ratio of 1:4, each PE has its share of 256 bytes in the SRAM buffer (figure 3.3). Memory *store* operations, however, do not fetch the data into the SRAM buffer.

Each PE consists of an ALU that performs integer arithmetic and logical operation, 16 32-bit dual read and write ported SRAM register file that acts as an-
other level of cache for hiding memory latency, an one byte flag register, and logic to select the controller being followed. It was observed that data-intensive applications typically manipulate array data in nested loops with no intervening access to other variables. Accordingly, a small register set with 16 registers was sufficient for applications studied on the processor. If this register set is accessed often, number of accesses to memory is significantly reduced resulting in better performance. Furthermore, deploying a large SRAM register set would reduce the memory yield. Hence, there is a tradeoff between speed and area in the case of SRAM and DRAM. Since the registers are 32-bits wide, compilers can pack byte or half-word data into a single register to improve code efficiency. Register 14 is designated as the stack pointer and register 15 is designated as the communication register. Register 15 is read and written to by the communication unit (communication among PEs is discussed in section 3.3). The Flag register holds the activity bit of the PEs and the two bit controller code the PE listens to for instructions. PEs execute the broadcasted instructions only if the activity bit is set. Since the register set is small, it can be multi-ported on a per-processor basis. Arithmetic and logical operations use a R/W port of the register file, and an other R/W port is shared by load/store and communication operations (Figure 3.2). Load/store instructions that take multiple cycles to complete are placed in a global load/store queue so that memory access can be overlapped with execution of other instructions. Instructions dependent on the result of a pending load operation in the queue stalls the pipeline. The interlock mechanism is centralized at the controllers
(no support is needed at PE-side).

As shown in the figure 3.1, unit arrays and PEs are arranged in a $N \times N$ mesh
(N being a power of two). Every PE has both and east and west neighbors with a
wrapped-around mesh connection. PEs can communicate directly with its immediate
neighbors. As mentioned earlier, controller broadcasts instructions to the PE mesh
using a broadcast tree. The broadcast tree is also used by the PEs when a single
PE wants to send data to the other active ones. With the PE number performing
the send encoded in the instruction itself, the send process is optimized with this
approach. In addition, reduction operations such as global AND and global OR are
supported using a broadcast-reduction tree.

### 3.3 Controller Organization

Controllers are responsible for fetching, decoding, executing scalar instruc-
tions and controlling the PEs. Each controller has 32 32-bit registers. The register
file has two Read and Write ports; one used for integer operations and the other for
load/store operations. Controllers are numbered from 0 to number of controllers -
1. The controller with id 0 acts as the main controller. The executable program’s
.text, .rdata, .data sections are loaded in controller 0, which begins the execution of
the application program. The PPIM executable also contains sections namely .text1,
.text2, .text3 and .text4. These sections hold the control parallel code and are loaded
into other controller memories. During control parallelism, controller zero forks con-
trollers and initiates their execution. It then waits for them to do a join after their execution is complete.

Each controller has a 32 bit address space and separate instruction and data memories. Together they form an address space of 4GB. However, the physical memory available to any single controller is limited by the chip size and transistor density. Moreover, PEs are not assigned a lot of local memories. Therefore, PPIM programs may access data and load instructions from off-chip memory core. Virtual address translation is performed using Instruction and Data Translation Buffers (TLBs) in each controller. In the current configuration of the processor, number of TLB entries is 8. Similar to the PE memory system, SRAM buffers serve as single line cache for both instruction and data memories. The width of the buffer is same as the width of the row in instruction and data memories, which is 1024 bytes.

3.3.1 Pipeline

With pipelining, multiple instructions are overlapped in execution to obtain more speedup. Consistent with the notion of viewing PPIM as a memory chip, logic overhead to exploit the parallelism should be as minimal as possible. Hence, we sought to implement a simple pipeline in the current configuration of the PPIM processor. Deeply pipelining or implementing a complex pipeline consumes a lot of gates. The decision to implement a simple pipeline also stems from the current technological limitation than any architectural barrier. Though logic based technological processes
Figure 3.4: Controller Pipeline

(for example IBM’s Cu-11 [9] and SA-27E [54]) does not alter the logic processes [23], the DRAM cell size in such mixed-technology process is greater than the cell size in the commodity DRAM process. Therefore, for the same amount of DRAM memory the area of Embedded DRAM chips will be greater. This increase in area has a negative impact on the yield and the cost of the chip. It can be observed that ongoing trends in mixed-mode technology are promising; additional complexity such as, utilizing multiple functional units and support for dynamic branch prediction can be added to the controller pipeline as the mixed-mode VLSI technological process matures.

Each controller implements a simple four stage pipeline with in-order issue and out-of-order completion of instructions. The pipeline uses separate instruction and data memories that are implemented using separate, single-line caches. The fetch stage reads instructions from the instruction memory. If the virtual address entry is cached in the instruction TLB, then fetch proceeds normally. Otherwise, the fetch stage stalls till TLB miss is served. In most cases, TLB miss is served either by
bringing in an off-chip memory core (page fault) or by replacing an entry in TLB cache; the latter occurs if the accessed page is found in on-chip memory but the TLB does not have an entry corresponding to that page. After the TLB miss is served, the fetch is restarted. The fetched instructions are passed down the pipeline to both controller and PE decode stages (shown in figure 3.4) whereby they are decoded. In the second half of the cycle, decode stage issues control signals to read the instruction operands into the pipeline registers from the appropriate register file. The controller execution unit performs integer addition, subtraction and logical operations. Other complex arithmetic operations such as multiplication and division are implemented in software. It can be seen that the outcome of the scalar conditional control instruction (such as a branch instruction) is known only at the execution stage of the pipeline. If a branch or a jump is made to a different address, the pipeline is flushed and the execution begins from the target address. PPIM processor has a branch delay of two cycles. Compilers can be made aware of the delay slots whereby useful operation can be performed by rescheduling instructions before the branch without violating dependencies. Each controller maintains three queues -

- Local load/store queue, where load/instructions meant for the controller are placed
- Parallel load/store queue, where parallel load/store instructions are placed
- Parallel communication queue, where communication instructions are placed
Each queue has four entries. Most instructions execute in a single cycle. Load/store, and communication instructions (for PEs) that take multiple cycles are pushed into the queues, and their execution is overlapped with the execution of other instructions in the pipeline. While the execute stage of the controller pipeline executes scalar instructions, PE stage of the pipeline broadcasts control signals to the PEs for execution of parallel instructions. Write-back stage of the pipeline always completes in a single cycle and does not cause pipeline stalls. In the first half cycle of the write-back stage, the results of the instruction are written to the appropriate register file.

**Dependency checking**

An instruction occurring in a pipeline may use the result of an earlier instruction still in execution in the pipeline. For example, an Add instruction reads its register contents in the decode stage. If an input register argument to the Add instruction is a result of a previous Shift instruction still in execution stage, then pipeline stalls. Such *Read-Before-Write* data hazards and the resultant stalls are avoided by forwarding data from the execution and write-back stage of the pipeline to the decode/execute stage pipeline registers. In PPIM, specifically, data is forwarded from write-back stage to execution/decode stage and from execution stage to decode stage as need be. As mentioned earlier, communication and memory instructions that takes multiple cycles to execute are pushed into the queues in the execution
stage. When an instruction is received from fetch stage, it is decoded and checked for flow dependency in the queues. Pipeline stalls if it has dependencies over the instructions still pending in the queue. This simple dependency checking guarantees that Write-Before-Write data hazards cannot occur. Moreover, stalling on the first occurring instruction dependency during issue does not require complex dependency tracking hardware. It can be observed that due to the nature of the PPIM pipeline, wherein the register contents are always read during the decode stage and written during the write-back stage, Write-Before-Read data hazards cannot occur.

Parallel load/store Operations

During control parallel portions of the program, active controllers may generate different memory addresses at the same time. Consequently, PEs in an unit array listening to different controllers may want to access different memory locations. Hence, some kind of logic is necessary to select and execute a single load/store instruction in an unit array. In PPIM processor, simple arbitration logic picks up the load/store instruction at the head of a controller’s queue and executes it. All the PEs following the selected controller completes the memory access. Likewise, all the controllers are served in a circular manner. Despite the simplicity of this scheme, it can be claimed that access is maintained fair to all the participating controllers.
3.4 Communication subsystem

The primary goals of the communication subsystem design are: (a) it should be simple, occupying small area; (b) it should utilize typical DRAM properties effectively without requiring excessive overhead. A communication operation is analogous to a parallel memory copy. Hence, an instruction of this type gets stored in the parallel load/store queue after decoding. Only active PEs take part in communication. Communication is supported as follows: (i) PEs copy the data that has to be transferred to the *communication register*. (ii) The communication instruction specifies the row shift amount and the column shift amount. The highest order bit in the row and the column shift amount indicates the direction of communication. A southward communication for the row and a east-ward communication for the column is indicated by a 0 in the leading bit.

Based on design goals mentioned above, following architectural choices were identified: (i) a centralized communication unit supporting regular communication operations. The communication unit consists of a buffers equal to total number of PE rows. The width of each buffer is the product of size of an integer word and number of PEs in a row. There is also a global communication buffer of the same width. The buffer, implemented as a shift register, transfers data by shifting it to a different PE. The communication instruction specifies the vertical and horizontal movements; if the vertical movement is zero and horizontal movement is non-zero, a horizontal shift in all rows is done in parallel. Otherwise, the contents of a single
column is picked across all the rows and copied to global buffer, shifted vertically and copied back. After all rows are operated on by the unit, data in buffer is written to the communication register of the PEs. (ii) In the second design approach, the communication buffer in every PE is connected to its four near-neighbors to form an interconnection network. The localized communication logic in every PE exchanges data according to vertical and horizontal movement specified by the instruction. In the former approach, the bit-lines have to be pulled out for a vertical read into the global communication buffer. The merged logic-DRAM process has additional metal layers to support it. However, if PPIM is fabricated in a metal poor DRAM process, it is not suitable. The simulator models the latter communication architecture.

3.5 Address Translator

The address translator is an optimization for improving the performance of PPIM when operating as a co-processor. For processing from within the M-SIMD programming model, it is necessary to partition the data in ways that match the internal architecture. On the other hand, the superscalar compiler is unaware of the mapping needed for PPIM and generates memory addresses on the assumption of a normal, contiguous data store. Hence, it is necessary to perform the translation of such addresses to the addresses as mapped within PPIM. Software translation can be performed by the compiler with the knowledge of data distribution in the PEs. However, it adds significant overhead (due to additional instructions) for every access
Figure 3.5: Translation of address generated by superscalar processor to PPIM internal address, illustrated using a 2-D image array mapping.

to intelligent memory. Accordingly, we chose to optimize the address mapping using a translator hardware integrated with PPIM (Figure 3.1).

For data arrays partitioned across PEs, an entry in the translator contains the size of an array, base address in the PEs and partitioned array size present in the PEs. For example, consider a 512 * 512 image partitioned as 16 * 16 sub-images and stored in PE memories beginning at location 0x100. An entry in the translator hardware is loaded with these dimensions within the scalar portion of the code. The translation process is illustrated in figure 3.5. Translator has eight entries in the current configuration (additional arrays would require software translation). Superscalar compiler generates only the offset address for all the arrays stored in PPIM memory; the translator entry holding the base address of the accessed array is used to construct the final address. The most significant three bits of the memory access instruction are then used to index into the translator table to pick the appropriate array entry.
3.6 Chip Area

In the PPIM architectural design, each controller has a memory of 16 MBits. In addition, each of the 1024 PEs has a memory of 256 KBits. It is difficult to precisely quantify the chip area of the PPIM processor without performing layout and routing. However, such an experimental evaluation of the design is premature and expensive. On the other hand, an approximate value of area can be estimated using the transistor counts of architectural designs that utilize logical elements present in the PPIM processor. Hence, range of values are used instead of single ones.

Every controller in the PPIM processor closely resembles the MIPS R2000 or SPARC logic core; MIPS R2000 was implemented using 110000 transistors, and the earlier SPARC CPU core used 75000 transistors. Processing element logic, however, is much simpler and can be implemented using 15000 - 20000 transistors [24]. In addition, transistor densities are important because, they include both the area used by the cells and the interconnect. Present day microprocessors typically have a density of 270000 - 310000 transistors per square mm in 0.18 \( \mu m \) logic technology [7]. Since there is no penalty for logic in logic-based mixed processes, the same density can be established. Furthermore, each 1Mb embedded memory core requires 0.8 square mm of area in logic-based mixed 0.18 \( \mu m \) technological process [23]. Based on these numbers, area of logic and memory portions of both controllers and PEs can be estimated to be 302 square mm in the logic-based mixed 0.18 \( \mu m \) technological process. However, additional area will be required for the communication units. As
the VLSI technology improves, PPIM design can be scaled to have more number of processing elements.

3.7 Summary

This chapter presented the architecture of the PPIM processor and the issues involved in design. Though PPIM processor capitalizes on fine grained data-parallelism as its primary source of parallelism, it supports independent control threads to a limited extent. Both processing elements and controllers support only the commonly used functionality. More expensive functionality (such as floating point operations) can be studied at a coarser level of sharing to amortize the cost over more memory columns. An estimate of the area of the PPIM processor using the current technological trends was also presented. The experiments conducted to study the performance of the architecture are presented in chapter 7.
Chapter 4

Instruction Set Architecture and Programming model

This chapter discusses the Instruction set architecture and the programming model of the PPIM processor. It describes the primary differences between SIMD and the scalar instruction sets and how multiple controllers can be effectively used in the program.

4.1 Instruction Set Architecture

The PPIM instruction set consists of MIPS based scalar instructions that operate on the controllers, SIMD instructions for the PEs, and multiple controller fork-join instructions. Data manipulation instructions are similar across both controller and PE instruction sets (SIMD instructions are prefixed with \( p \)). These in-
Figure 4.1: Executing iteration statements (within SIMD block) when loop terminating condition depends on a scalar, compile time value.

Instructions operate only on registers, while the memory references are performed only by load/store instructions (the instructions are not shown for conciseness). Such RISC type instruction set implementations require simpler control logic. The instruction sets, however, differ in control flow and activity management instructions.

Control instructions such as jump alter the program flow unconditionally. Other control instructions such as branch, alter the program flow based on the condition tested on scalar variables. Conversely, the outcome of a test on a vector variable may be different across PEs. Hence, at the PE side control flow instructions are replaced by activity management instructions that determine whether a PE participates in the broadcast instruction or not. Activity management instructions set/reset the activity bit based on the local condition tested on the PE side. If the activity bit is reset, the broadcasted instruction is ignored by the PE. Unconditional instructions, such as movuc <register> and neguc <register> are executed by the PEs irrespective of the state of activity bit. Unconditional move and negate instructions are
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD block: (start_row/end_row - differ across PEs)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>for (index = start_row; index &lt; end_row; index++) {</td>
<td></td>
</tr>
<tr>
<td>&lt;parallel loop code&gt;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2: Existing iteration statements (within SIMD block) when loop terminating condition depends on a vector variable.

necessary to restore the activity status of the PEs on completion of a branch and to activate all PEs at the beginning of a SIMD routine respectively.

Figure 4.1 and Figure 4.2 exemplifies evaluation of termination condition for iteration statements such as for and while in the SIMD block. As shown in figure 4.1, if a compile time constant determines the termination of the loop, the loop can be evaluated in the scalar unit. Conversely, loop conditions involving vector variables are evaluated in the PEs (figure 4.2). Finally, global reduction operators are used to evaluate if all the PE’s have finished executing the loop.

4.2 Programming Model

The program begins by executing scalar instructions. It may call methods that operate on the PE array anytime during execution. Within the SIMD code block,
control parallel portions (different execution threads) are supported using multiple controllers. PEs decide on the controller they want to listen to by executing the instruction `ctrl <register>`, where the two least significant bits of the register argument contain the controller number to follow. For example, consider an if-elseif-else statement in the SIMD code block. Those PEs that successfully test the if portion of the conditional statement may choose to listen to controller one (the actual execution of controller one begins afterwards, however). Rest of the PEs test the condition for the elseif part; successful PEs may choose to listen to controller two and so on. The code for the default case is supported by controller zero. The `ctrl` instruction writes the controller id to the flag register, which is read by the steering logic to select the controller instruction to follow. As mentioned above, it does not initiate the execution of controllers. After the PEs select the controller they want to
listen to, controller zero executes the \texttt{c.parallel} instruction to begin execution of participating controllers. Multiple instructions can be active in the PE array after this time. Since the controllers may finish execution at different times, processor reverts to the SIMD mode after all the controllers execute the \texttt{complete} instruction. The programming model for the PPIM processor is of Single Program Multiple Data (SPMD) type, and is illustrated in Figure 4.3. Specifically, PEs execute same portion of code on different data. However, each PE make take its own path in the program.
Chapter 5

PPIM Simulation

This chapter discusses \textit{ppim-sim}, a cycle-accurate simulator that models the PPIM architecture. It discusses the challenges involved in its design and development. Simulation using \textit{ppim-sim} is a three step procedure - Firstly, applications are developed in the PPIM assembly language by extending a sequential version compiled into MIPS assembly. Secondly, the assembly code is compiled using GNU assembler extended for PPIM architecture. Thirdly, the simulator reads the PPIM binaries and \textit{run} them.

5.1 Introduction

PPIM is a large scale system designed for chips with hundreds of millions of transistors. Analytical evaluation of such a system is infeasible without simplifying assumptions that would make it unrealistic. Experimental evaluation of the design
is premature and expensive (a silicon run in a mixed process is currently priced at
around half a million dollars). Alternatively, PPIM can be modeled in software; the
architecture can be *simulated* to analyze the performance of typical applications that
run on them. In general, simulation is the most widely used method for architecture
evaluation because it allows inexpensive and flexible evaluation of design tradeoffs.
For example, the cache organization in a microprocessor can be studied with dif-
ferent block sizes and numbers, associativity and replacement strategy. Therefore,
simulation was elected to evaluate PPIM.

5.2 Related Architectural Simulators

Architectural simulators provide a cost-effective method to test new ideas
and analyze trade-offs in the design process. A large number of architectural simula-
tors have been developed in the course of computer architecture research. For space
considerations, only the most notable and widely used efforts are reviewed in this
section. It should be noted that these simulators have undergone several releases and
are highly optimized.

**SPIM:** *SPIM* is a simulator that runs MIPS R2000/R3000 programs [35].
The MIPS assembler implements a virtual machine that hides the restricted address
modes, delayed loads and delayed branches present in the MIPS architecture. In
addition, it provides a number of pseudoinstructions that appear as real assembly
language instructions. *SPIM* can simulate both the virtual machine and the actual
hardware [35]. The simulator is also equipped with a debugger that allows inserting breakpoints and single stepping through the assembly code. Furthermore, it provides a small subset of operating system services for reading and writing onto the terminal through system calls. *SPIM*, however, does not simulate memory access latencies and caches.

**Simplescalar:** The *Simplescalar* tool-set [5] performs simulation of modern microprocessors that implement the Simplescalar architecture (that is based on the MIPS architecture). The tool-set includes five execution-driven processor models that range from fast functional simulators to a detailed out-of-order issue model that implements speculation, wrong path execution and non-blocking caches [5]. The models tradeoff accuracy of simulation to execution time. In addition, the Simplescalar tool-set includes GNU tools re-targeted for Simplescalar architecture. This allows the study of realistic benchmarks – any C program can be cross compiled and simulated. The simulators read binaries built for *Simplescalar* architecture and executes them. The tool-set includes a symbolic debugger and a tool to trace out-of-order pipeline. *ppim-sim* is based on the *Simplescalar* simulator organization.

**SimOS:** *SimOS* is a complete simulation environment for uniprocessor and multiprocessor computer systems [52]. The research tool-set provides simulators of microprocessors, disk drives, memory buses, Ethernet and other components in the computing system. The tool-set provides models of MIPS R4000 & R10000 and Digital Alpha processor families. It simulates the hardware in enough detail that the
commercial operating systems that run on these processors, can be booted through the simulation environment [60]. The operating system runs the application workloads similar to actual machines. Similar to *SimpleScalar*, the tool-set includes CPU models with varying speed-detail tradeoff. Moreover, the simulation level of details can be changed at runtime. This is beneficial when simulating complex workloads because, the less-interesting portions can be run with greater speed while the desired portions of the workload can be executed with greater detail.

**Trimaran:** *Trimaran* is a compiler research infrastructure for Explicitly Parallel Instruction Computing architectures [70]. The EPIC architecture is most similar to our design (although the scale and granularity of parallelism are different). Machine configurations that vary in the number and type of functional units and register files are specified using a machine description component included in the tool-set. Moreover, compiler back-end tools can be used to implement different code optimizations, instruction scheduling and register allocation techniques. A cycle-level simulator is used to study the architecture and the compilation framework; it generates statistics such as resource utilization and execution time, that can be used for further compiler optimizations.

### 5.3 Simulator Overview

*ppim-sim* was developed to study and analyze different configurations of the PPIM processor. The simulator was carefully tuned for performance without
compromising flexibility. More importantly, the simulator is parameterized to enable study of different architectural designs with considerable ease. Configuration options such as number of controllers and processing elements in the processor, amount of memory allocated to each PE, number of columns in the unit array, pitch-match ratio and width of load/store queue are parameterized in the simulator.

`ppim-sim` is both cycle-driven and event-driven (simulator events are discussed in section A.2). Cycle-driven simulation is efficient for synchronous behavior that can be scheduled statically. However, for asynchronous “sparse” events, event-driven simulation is necessary because static scheduling is difficult. Conversely, using event-driven simulation throughout would be inefficient as the overhead of event scheduling and list management is incurred on every event (an overhead that cycle-driven simulation avoids). The simulator is cycle-accurate, providing accurate ma-
chine state at every cycle boundary.

"ppim-sim" is both extensible and portable. Similar to Simplescalar, ppim-sim uses 64-bit instruction encoding that facilities adding new instructions and annotating existing instructions. Moreover, the ported GNU tools makes it easier both to extend the simulator and to develop new simulators at different levels of detail. The simulator can execute both big-endian and little-endian executables. However, for the simulator to work correctly the target endianness must match the host endianness.

Figure 5.1 shows the overview of the simulation tools. Applications are partitioned to execute the parallel portions using ppim-sim, while executing scalar portions using sim-outorder. Application models are developed in the PPIM assembly language by modifying a sequential version compiled into MIPS assembly. GNU assembler re-targeted for Simplescalar architecture is extended with new sections. Control parallel portions are identified within the assembly language using separate sections, given by .text<controller_id> in the object code. Data common to PEs reside in section .psdata. Programs written in PPIM assembly language are compiled using a Simplescalar GNU assembler extended for assembling PPIM programs. The simulator takes binaries compiled for PPIM.

The simulator uses GNU BFD library routines to read the executable, to get the program entry address, to read the size of each section and section contents. In addition, the simulator loads the section contents appropriately into controller and PE memories. Figure 5.2 shows the layout of the PPIM executable. As seen in the
Figure 5.2: Virtual memory layout in a PPIM executable

In the figure, the control parallel portions appear following the data sections in the assembly code. Before simulation begins, they are relocated to a parameterizable start address in the multiple controllers memory. Instruction memory of controller zero is loaded with .text and .data segment. PE memories are loaded with contents from section .psdata. Moreover, data that is not common to PEs are read from a separate file and the corresponding PE memories are initialized. Control parallel sections from the executable are loaded into corresponding controller memories.
5.4 Simulator Organization

PPIM architecture has two types of processors: processing elements and controllers. The instruction set for these two processor types are similar, but has different opcodes. Every scalar and parallel instruction definition consists of:

1. Instruction type: for a scalar instruction, the instruction type can be S_ALU_REGISTER, S_ALU_IMMEDIATE, S_MEMORY, S_CNTRL or S_SPECIAL. Load/store and PE communication instructions are identified based on the type field. The type of the instruction also indicates its instruction format.

2. Input and output registers used by the instruction: precisely, the bit-mask that extracts the registers used by the instruction are specified here. This register list is used for easy dependency checking. For example, during decode input registers used by an arithmetic instruction are checked with the output register used by a pending load instruction to identify input dependency.

3. Number of cycles spent in the execution stage of the pipeline. Integer arithmetic and logical operations execute in a single cycle. These numbers are, however, parameterized within the simulator.

4. An expression that implements the instruction. The expression can be a function call or a macro. Arithmetic and logic instructions manipulate only the
register contents and are implemented as macros. However, load/store, communication and control instructions are implemented as function calls. For example, executing `jump` instruction calls the function `jump_handler` that flushes the pipeline, cancels an on-going instruction fetch operation and sets the program counter to the `jump` address.

Simulator reads an instruction at a time; based on its opcode, it executes appropriate code that models the instruction definition. System calls are handled in the simulator using a proxy system call handler that performs the system call on behalf of the target program. More specifically, the simulator reads the system call inputs from the target programs memory, performs the system call on the host machine and updates the register and memory of the target program based on the result of the call. The simulator also takes into account the refresh cycles, that are necessary to periodically refresh stored values before they are drained by leakage currents.

5.4.1 Pipeline

As mentioned in Chapter 3, each controller implements a simple pipeline with in-order issue and out-of-order completion with respect to load/store and communication instructions. The simulator models each stage of the pipeline as a software routine. Concurrent execution of all controllers is simulated by executing a stage of the pipeline for all active controllers before beginning the next stage. Figure 5.3 shows
for (;;) {
  for (all active controllers)
    process_event_queue(); /* process events scheduled for this cycle */
  for (all active controllers)
    fetch(); /* do a fetch from memory/buffer */
  for (all active controllers)
    decode(); /* decode instruction and identify dependencies */
  for (all active controllers)
    ctrl_execute(); /* execute scalar instruction */
  for (all active controllers)
    pe_execute(); /* execute SIMD instruction */
  for (all active controllers)
    ctrl_writeback(); /* writeback stage for controller */
  for (all active controllers)
    pe_writeback(); /* writeback stage for PEs */
  cycle_time++;
}

Figure 5.3: Structure of main loop in \textit{ppim-sim} simulator

the simulator loop that gets executed in every cycle. Simulation ends when the target program executes an \textit{exit} system call.

The fetch unit models the machine’s instruction fetch stage. The fetch stage goes through the instruction TLB. If the page for the instruction is not found in on-chip memory, pipeline stalls till the page miss is serviced. The entire row corresponding to the PC address is then read into the single line buffer that acts as a cache for future memory accesses. On the other hand, if the page is found in the memory but the generated row address does not match the row in the buffer, the \textbf{fetch} stage generates an event with timestamp, current cycle time + 1 to stall the pipeline and an event with timestamp, current cycle time + 1 + memory access time to un-stall the pipeline.

The routine \textbf{decode} models the decode functionality for both scalar and parallel instructions. An instruction from the fetch stage is decoded and checked for flow and anti-dependencies by walking through the controller queues. If a dependency
exists, the routine posts an event to restart the decode operation at a time equal to the completion of the first instruction in the queue + 1.

As shown in figure 5.3, the execution stage of the pipeline is simulated using two methods. For all the active controllers, the method `ctrl_execute` executes scalar instructions and the method `pe_execute` executes parallel instructions. Most instructions spend only a single cycle in the execution stage. However, the scalar and parallel load/store instructions that take multiple cycles to execute are placed in the corresponding queues. Communication instructions are placed in a separate queue as well. In the PE array, arithmetic and logic instructions broadcasted by all the controllers are executed concurrently. However, only a load/store operation from a controller can be active at a time. Scalar load/store instructions go through the data TLB. Contrary to the fetch stage, an exception such as page fault or TLB miss does not cause a pipeline stall. Pipeline stalls only if the result of the pending load operation is needed down the instruction stream.

The final write-back stage of the pipeline is simulated in the methods `ctrl_writeback` and `pe_writeback`. This stage always finishes in a cycle and does not create any pipeline stalls. The results are assumed to be written to the register file and also forwarded to avoid stalls due to data dependencies. As seen, all the stages of the pipeline can schedule events to occur in the future time. The method `process_event_queue` walks down the list of events on each controller and processes them. Primarily, it either stalls or un-stalls the pipeline based on the posted events.
Queue Management

Each controller has three queues, namely local load/store queue (LSQ), parallel load/store queue and parallel communication queue. These queues are manipulated in the corresponding queue’s operation handler function. For example, the scalar load_handler method pushes the load instructions whose dependencies are satisfied to the scalar load/store queue. In addition, it generates a LOAD_COMPLETE event with the timestamp of the completion time of the pushed instruction. However, if the insertion was unsuccessful due to LSQ being full, it generates a STALL_PIPELINE_LS event to stall the pipeline till a place is created in the queue. Conversely, since each controller has its own parallel load/store queue, the actual completion of a parallel load/store operation is based on the time taken in multiple controllers. It should be noted that one of the write ports of the register file is accessed by load/store as well as communication operations. Hence, accesses to this port are serialized. It is accomplished by maintaining the time at which the port will be available for writing. If a load/store instruction finishes at the same time as a communication operation, access is given to the instruction that first arrived in program order.

5.4.2 Verification

Rigorous testing was done to verify the correctness of the simulation process and the accuracy of the statistics generated. In fact, simulation components were tested after a checkpointed stage in the design was reached. The simulator’s controller
portion was tested using the assembly code and binary generated by GNU compiler re-targeted for Simplescalar. Correctness of the single controller (scalar) portion of ppim-sim was verified by matching the instruction trace generated by the program running on Simplescalar and ppim-sim. In addition, behavior of the controller pipeline was verified by processing the pipeline trace collected from the simulator. Moreover, appropriate system calls were added to the simulator to print the execution output of programs.

After the back-end of the compiler was modified to support SIMD semantics, thorough testing was done to check the correctness of each of the components of the simulation system. Execution outputs gathered from the processing element memories were used to verify the functionality and correctness of the SIMD portion of the simulator. In addition, many small data-intensive applications were run on the ppim-sim simulator to verify the simulation statistics. Finally, the output for each of the studied applications were also verified against their sequential counterparts.

5.5 Summary

This chapter presented ppim-sim that models the PPIM architecture in software. Experiments were conducted to evaluate the simulator using a number of data-intensive application models for varying PPIM configurations. It was observed from the experiments that ppim-sim not only simulates large models in tractable amounts of time, but also is memory efficient. The experiments conducted together
with the results obtained are presented in section A.1 of Appendix 8.2. In addition, the parameterized design of *ppim-sim* coupled with robust and effective interfaces makes it a research tool to study different processing element and controller architectures implemented in memory.
Chapter 6

Applications

Applications in VLSI, image processing, multimedia and database domains are known to be data-intensive. Hence, applications were chosen from these domains for development and study on the PPIM processor. This chapter discusses such applications and how they were mapped onto the M-SIMD programming model.

Query handling: For the user submitted queries to a database system, the query processor initially builds the query trees. Optimal execution of the query is determined; an execution plan is then laid out using the optimized query tree. With a number of processing elements and limited controllers in PPIM processor, parallel execution of the query can be done. Two phases are identified in the execution of the query – In the first phase, the portions of the query that are not dependent are used to select the tuple sets from different relations. Multiple controllers operating on different relations do the selection operation concurrently [4]. For example, consider the query that picks the employee records, for employees working in the “Production” department, on a project at location “Clifton” and whose salaries not falling in the
average range:

```sql
SELECT emp.number, emp.name FROM employee as e, department as d, project as p
WHERE p.location = 'clifton' and d.name = 'production'
    and (e.deptid = d.id) and (p.deptid = d.id)
    and (e.projid = p.projid) and (e.sal >= lowsal && e.sal <= hisal)
```

Since a database relation is large to be held in a PE memory, it is distributed to a group of PE memories. After completion of the first phase, every PE holds employee or project or department records matching the selection criteria from its subset of the relation records. In the second phase, join operation is performed. Join operation combines related tuples from different relations into single tuples. A Cartesian product of different relations is performed and only the tuples satisfying the join condition are extracted. To accomplish this, a single controller works on PE memories containing the relations to be joined. The relation with fewer selected tuples is chosen as the inner relation and are broadcast to the PEs holding the larger outer relation. The join operations result in the interested records. In the above example, the selected department_id is used to eliminate some tuples in employee and project relations. Then a join between project and employee relations gives the result. Relational database operations such as `update`, `delete` and `add` can also be efficiently handled using multiple controllers.

**Image Segmentation:** This application divides an image into its constituent parts to identify features of interest. Image segmentation is implemented
Figure 6.1: Mapping of a 2-D image onto the PE mesh

in the PPIM processor, by logically viewing the 1024 PEs as a grid of 32 * 32, and superimposing the $E^*E$ image on the grid. This is illustrated in Figure 6.1. Each PE operates on sub-images of the original image. To reduce communication, overlapping sub-images are loaded into PE memories (Figure 6.1).

For every pixel in the sub-image, gradient and Laplacian values are found. Gradient values are used to detect the presence of an edge in the image and Laplacian values are used to determine whether a pixel lies in the dark or light side of an image [17]. Computation of gradient of an image is essentially obtaining first order, partial derivatives of an image; Sobel operator mask is used to compute the gradient values for every pixel location in the image. The resultant gradient image has the same size as the original image. Computation of the Laplacian of an image is based on obtaining second order, partial derivatives of an image; a spatial mask with the coefficient associated with the center pixel be positive and the outer pixels be negative
is used to compute the Laplacian values for every pixel location in the image. Based on the gradient and Laplacian values a three level image is generated [17]. Multiple controllers support different execution threads based on the values of the three level image, to construct the binary image; the objects of interest are identified in the binary image.

**Contour Extraction:** It is the manipulation of an image such that only contours remain. The input image is partitioned similar to image segmentation as described above and stored in PE memories. The algorithm works in two phases: (i) Each PE locates the first point on the contour in its sub-image with a linear left-to-right scan. In the implemented contour extraction, identifying the first point on the contour involves locating the user specified color of the contour in the sub-image. For example, landscape contours are identified as brown, while the oceanic contours are identified as blue. PEs terminate the first phase, when a point on the contour is identified in their sub-image or if none is found. Consequently, PEs complete the first phase at different times. (ii) The neighboring pixels are then searched in the anti-clockwise direction to locate the next point on the contour [72, 47]. Every pixel not lying on the image boundary has 8 neighbors to it; searching in the anti-clockwise direction walks along the neighboring pixels and selects the rightmost available pixel that belongs to the contour set [47]. Based on the search direction that revealed a contour, different execution threads are possible. After the phase two is complete, PEs resume scanning and searching to find other contours in their sub-images. When
the PEs have finished scanning its entire sub-image all the contours in the sub-image are traced.

**Fault simulation:** is an important step in testing of logic networks. It is used to determine the faults identified by the given set of test vectors. The level ordered circuit is placed in every PE memory. The amount of PE memory needed is directly proportional to the number of nodes in the simulated network. For larger networks, a sub-network with expected faults is simulated in detail and the rest of the network is logically abstracted to a smaller one. On the other hand, simulation of the entire larger circuit in detail can also be accomplished by partitioning the circuit and proceeding in phases.

In the implemented parallel fault simulation, one PE simulates a fault-free
network and every other PE simulates a faulty network \([42]\). If the number of faults in the network is more than the number of PEs, then fault list can be divided and simulation accomplished in more than one pass \([42]\). Each PE uses the Parallel Pattern Single Fault Propagation (PPSFP) algorithm for fault simulation \([2]\). The input vectors are placed in the PE memories. Logic values of the nodes of the network during simulation are stored in corresponding PE memory as well. Same input vector is simulated across all the PEs. After the simulation, the outputs of the fault-free network are sent to PEs simulating the faulty network. The participating PEs then compare their local output with the fault-free output. In case of a output match in a PE, the input vector did not successfully determine the fault simulated by that PE. The circuit is then simulated for the remaining test vectors. All the faults identified by the set of test vectors is determined.
Chapter 7

Experiments

This chapter presents the experiments conducted to determine: (i) performance of data-intensive applications on the PPIM processor as compared to the performance on modern superscalar processor; (ii) quantify the usefulness of multiple controllers; (iii) extent in memory traffic reduction when using PPIM as a co-processor; (iv) performance of applications executing only on superscalar processor as compared to applications also using PPIM co-processor. This chapter also presents discussions on the results obtained from the experiments.

7.1 Developing Applications for PPIM

Application models studied in the PPIM processor were discussed in chapter 6. They were developed in many phases: (i) Implementation was carried out in the high-level language; testing and debugging were done using the host machine tools.
The high-level C program is then cross-compiled using the GNU compiler tools of the Simplescalar tool-set. The resultant executable is simulated using sim-outorder of the simplescalar simulation tool-set. (ii) The algorithm is parallelized suitable for implementation on the PPIM processor with SPMD programming model. Simulation of the parallel SPMD program was carried out to test the correctness of parallelization methodology. It was time efficient to test the correctness before it was developed in PPIM assembly language. The verified code is cross-compiled to generate MIPS assembly. (iii) The assembly program is extended with SIMD and control parallel semantics and compiled to get the PPIM executable (figure 5.1). PPIM binaries
are executed using \textit{ppim-sim}.

### 7.2 PPIM as a Processor

\textit{ppim-sim} models the architectural configuration of PPIM mentioned in chapter 3. The simulated superscalar processor has a separate instruction and data L1 cache, unified L2 cache and four integer ALUs. The cache configurations given as Number of sets/block size/ associativity are: Instruction L1 cache - 512/32/1, Data L1 cache - 128/32/4 and L2 cache - 1024/64/4. Both L1 and L2 caches used LRU cache replacement strategy. The graphs in figure 7.1 and figure 7.2 show the
execution cycles of the applications on the modern superscalar processor as compared to the cycles on the PPIM processor. The Y-axis of the graphs shows the number of cycles in log scale. For the experiments, the problem size was scaled with the PEs, keeping the work done per PE a constant. Studies with varying number of PEs show that PPIM performs better than superscalar processor, with the speedup increasing as the utilization of the PEs increase. The difference in cycle times when PEs are scaled could largely be attributed to the change in communication patterns for larger datasets. It should also be noted that when number of PEs are scaled, the difference in cycle times is very small as compared to total execution cycle time of
<table>
<thead>
<tr>
<th>Application Name</th>
<th>Pure SIMD</th>
<th>PPIM 2-controllers</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault Simulation</td>
<td>633,923</td>
<td>603,495</td>
<td>5%</td>
</tr>
<tr>
<td>Image Segmentation</td>
<td>1,019,609</td>
<td>979,845</td>
<td>4%</td>
</tr>
<tr>
<td>Query Handling</td>
<td>34,339</td>
<td>32,210</td>
<td>6%</td>
</tr>
<tr>
<td>Contour Extraction</td>
<td>2,376,693</td>
<td>2,288,756</td>
<td>4%</td>
</tr>
</tbody>
</table>

Table 7.1: Number of cycles taken by applications executing on a pure SIMD and two controller configurations of the PPIM processor.

The graphs in figure 7.3 show the theoretical and observed speedup for the Image Segmentation (best case) and the Query Handling (worst case) applications. Although, the observed speedup is linear and increases with the utilization of the PEs, it does not match with the expected speedup. This is because all the PEs are not doing useful work during all the stages of execution. More specific reasons include: (i) iteratively activating and de-activating PEs within a loop involves execution overheads that are not accounted for theoretical model; (ii) applications contain irregular portions that cannot be parallelized and are best executed sequentially (parallelism overheads negate benefits gained by parallelization); (iii) granularity within the application is not sufficient to utilize all the available PEs; (iv) synchronization operations that are necessary to guarantee correct execution of the application results in additional overheads.

The difference in ratio of speedup for the best and worst case applications (shown in figure 7.3) could be described as follows: Image segmentation is a reg-
<table>
<thead>
<tr>
<th>Application Name</th>
<th>Pure SIMD</th>
<th>PPIM 4-controllers</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault Simulation</td>
<td>633,923</td>
<td>545,366</td>
<td>14%</td>
</tr>
<tr>
<td>Image Segmentation</td>
<td>1,019,609</td>
<td>894,092</td>
<td>12%</td>
</tr>
<tr>
<td>Query Handling</td>
<td>34,339</td>
<td>29,534</td>
<td>14%</td>
</tr>
<tr>
<td>Contour Extraction</td>
<td>2,376,693</td>
<td>2,173,514</td>
<td>9%</td>
</tr>
</tbody>
</table>

Table 7.2: Number of cycles taken by applications executing on a pure SIMD and four controller configurations of the PPIM processor.

ular application well suited to the M-SIMD type programming model. Moreover, assigning overlapping sub-images to the PEs significantly reduces the communication overheads. On the other hand, join phase within the Query Handling application is communication bound and relatively irregular. The irregularity limits the overall speedup gained by this application.

Table 7.1 and Table 7.2 show the number of cycles taken by applications executing on 2-controller and 4-controller configurations of PPIM processor as compared to its pure SIMD implementation. The simulations for the results shown in tables used 256 PEs. It was observed that utilization of multiple controllers remained almost the same for varying number of participating PEs. This is because, control parallelism is inherent in the application and does not change with number of PEs used. Limited amount of control parallelism was observed in the studied applications - Contour Extraction and Query Handling had four control independent portions, and Fault Simulation and Image Segmentation had three control independent portions. For Fault Simulation and Image Segmentation, adding a
third controller supports all the control-independent portions in the applications; it can be observed that, this results in a significant performance improvement using 4 controllers (14% in the best case). The overhead in terms of silicon area needed to obtain this speedup is comparatively small (75,000 transistors for each controller core). Presently, only a single dominant control parallel portion in the application is exploited. It is done by placing each control independent fragment in separate section (given by .text<controller_id>) within the assembly file; when multiple controllers are activated, execution begins from the instruction at the beginning of the section and continues till the end. It is possible to exploit all of the control parallel portions present in the applications by beginning execution of multiple controllers at a different address. However, if the control parallel portions are trivially small, the cost of initializing, activating and de-activating the controllers would counter any benefit obtained from exploiting this parallelism.

7.3 PPIM as co-processor

The experiments discussed so far execute entire applications on PPIM processor. For code sequences that are not data-intensive and have little inherent parallelism, a superscalar processor remains the most effective model for execution. This is also true for operations that are not supported within PPIM (e.g., floating point operations). We investigate the use of PPIM as a co-processor, where regions of the code that are data-intensive are mapped to PPIM and others that are executed
Figure 7.4: Reduction in memory traffic with PPIM co-processor

on a primary superscalar core. The benefit of this configuration is two-folds: (i) data-intensive regions can exploit the parallelism available within PPIM; and (ii) the effective working set for the superscalar processor is made smaller, reducing the traffic on the system bus and alleviating the memory bottleneck. *ppim-sim* was integrated with the superscalar tool-set to operate as a co-processor. We extended Image Segmentation and Query Handling applications discussed in Chapter 6 to operate in this fashion. If the original input image in Image Segmentation is contaminated with noise, discontinuities appear in the final edge detected image. In such cases, the image has to be corrected before further processing. Hough transform for line detection can be used to extract broken lines in the image. After a point-to-curve
transformation of every foreground pixel in the image, the co-linear points are identified as the curves that intersect at a common point [17]. This floating point intensive application is implemented by quantizing the hough parameter space into discrete bins, and accumulating values in the bins. **Query Handling** application is extended to support set aggregate operations on query that gets executed on the superscalar processor. For the experiments, PPIM co-processor shared the virtual address space of the processor. Figure 7.4 shows the number of references to memory while running these applications on a superscalar processor with and without co-processing support. The Y-axis of the graphs shows the number of references in log scale. As illustrated by the graphs, co-processing using PPIM significantly reduces the traffic on the memory bus - as much as 85% for the **Image Segmentation** and 65% - 86% for **Query Handling**. The effective working set is increased in performing aggregate operations on larger datasets for **Query Handling**; this contributes to the variation in memory traffic for the partitioned code sequence executing on the superscalar processor.

The graphs in figure 7.5 gives the execution cycles of applications on superscalar processor as compared to the cycles taken both on PPIM co-processor and superscalar processor. **Query Handling** runs twice as faster using the PPIM co-processor for executing data-intensive portions, while **Image Segmentation** runs 1.5 times faster. The speedup, however, is controlled by the fraction of the code sequence that gets executed on the PPIM co-processor. Executing pure data-intensive applications produces orders of magnitude in performance improvement (Figure 7.1
Figure 7.5: Cycle counts without and with PPIM co-processor

and 7.2), with the improvement deteriorating as the scalar code sequence within the
application increases (Figure 7.5). With a multi-tasking operating system, the superscalar processor can context-switch to a different execution thread while waiting for the results from the co-processor execution.

7.4 Summary

The results obtained from the simulation data can be summarized as follows:

- For the studied data-intensive applications, PPIM processor yields orders of magnitude improvement in performance as compared to the superscalar proces-
sor.

- Using singlescalar, multiple controllers results in considerable speedup as compared to pure SIMD implementation of PPIM.

- Data-intensive applications using PPIM co-processor provides dual advantage of faster execution and significantly reduced traffic on the memory bus.
Chapter 8

Conclusions and Future Work

Memory sub-system performance has not kept pace with the performance of processing logic; limited bandwidth available on the system bus and access latency continue to form a bottleneck in modern high-performance superscalar processors. Furthermore, the “typical” applications are shifting towards data-intensive and multimedia type workloads. Integrating logic and memory on the same chip offers potential relief for this problem and is also a viable solution in the current technological processes. These observations formed the motivation for developing a parallel processor-in-memory. PPIM architectural design, consisting of four simple, single-scalar controllers and 1024 Processing elements integrated with memory on a chip was presented. ppim-sim, that implements the PPIM architectural model in software and capable of simulating PPIM binary programs was discussed. Some applications from VLSI, Image processing/Multimedia and database domains were
modeled for study on the PPIM processor. Experiments conducted to quantity the performance of PPIM processor together with the results obtained were presented.

8.1 Conclusion

The empirical results indicate the effectiveness of PPIM in executing data-intensive applications. Specifically, PPIM performs better than superscalar processor, with the speedup increasing as the utilization of the PEs increase. Utilizing four simple, singlescalar controllers supports control parallel sequences present in the applications and results in significant performance improvement (14% in the best case). The experimental results also show that co-processing using PPIM significantly reduces the traffic on the memory bus (85% in the best case) and improves the execution time of the applications. The following can be concluded from the experiments conducted:

- PPIM processor is conducive in supporting data-intensive applications.

- For irregular, floating point intensive applications superscalar processor remains the best model of execution.

- The performance improvement obtained by co-processing using PPIM is dependent on the fraction of the application that gets executed on the co-processor.

- Four controllers within the PPIM processor was sufficient to exploit all the control independent portions for the studied applications.
• The experiments showed that the speedup increases with increasing number of PEs; hence, more speedup can be expected using larger configurations of PEs in the PPIM processor.

The experimental evidence presented in this thesis does show the effectiveness of processing in memory. More specifically, data-intensive applications that have control and data parallelism achieves superior performance with this approach.

### 8.2 Future work

The von-Neumann memory bottleneck due to the gap between processing rate and off-chip memory access latency/limited bandwidth severely affect the modern microprocessors. Contemporary processors employ increased on-chip caches, sophisticated hardware and software techniques to bridge this gap. This thesis presents an alternative architectural design whereby M-SIMD logic is placed together with the DRAM memory on the same chip; by doing so, the logic is exposed to wider memory bandwidth at a significantly less access latency. It was shown that this approach is promising in improving the performance of data-intensive applications and in reducing the traffic on the memory bus. Some directions of future research in this area are:

• Explore the feasibility of one or more floating point unit on the PPIM processor.

It was seen that different domains have applications that involve floating point
arithmetic operations. In-place floating point unit(s) may help further reduce the memory traffic and improve speedup of execution. In addition, PPIM can support wider variety of applications.

- Communication sub-system demands more focus; various design alternatives can be analyzed and their suitability of implementation in a mixed-technological process can be compared. Furthermore, an inter-controller communication protocol (using shared memory or otherwise) will add more flexibility to the code sequences that can potentially get executed in the multiple controllers.

- Using PPIM approach, power consumption for processing is reduced because: (i) applications utilizing PPIM complete sooner; (ii) by doing some of the processing in memory, the need to drive the high-capacitive IO pads and buses is reduced. However, the power consumption of the intelligent DRAM itself may be greater because of logic additions to the chip. It will be interesting to study and quantify the power consumption of the system utilizing PPIM.

- Addition of debugging support in ppim-sim wherein (i) instructions at different stages of the pipeline can be observed; (ii) execution can be controlled by inserting breakpoints and single stepping of assembly code. This enables verifying new instruction implementations and debugging assembly programs easier.

- Enable developing applications for PPIM in high level language. In particular, the GNU gcc compiler re-targeted for SimpleScalar architecture can be extended
to compile high-level PPIM programs. This will reduce the time needed to
develop applications. Consequently, wider set of applications can be developed
and other existing M-SIMD applications can be studied on PPIM processor.

- PEs and controllers within the PPIM processor can be implemented in Magic
to get precise transistor counts. It can be utilized to accurately determine the
area of the PPIM processor.
Bibliography


Appendix A

A.1 Experiments on \textit{ppim-sim}

Experiments were conducted to evaluate the performance of \textit{ppim-sim} (ex-
ecution time and peak memory consumption) on the selected applications and for
different configurations. The performance of the simulator is also compared to the
performance of \textit{Simplescalar} for the same applications. All experiments were run on
a workstation consisting of two Pentium II, 166 MHz processors sharing 128 MB of
memory. \textit{ppim-sim}, however, used only a single processor.

Figure A.1 and Figure A.2 present the simulation times for the applications
for varying number of PEs. It should be noted that run-time of the simulator is the
sum of simulator initialization time, load time and the simulation time. Initialization
time primarily consists of initializing simulator internal data structures that is based
on the simulated PPIM configuration. Load time is the time taken to initialize PE
memories with data before starting the simulation. Hence, the sum of load and ini-
Figure A.1: Simulator run time for increasing number of PEs

 Initialization times gives the time elapsed before simulation begins (or the simulation overhead). As can be seen in the figures, the initialization time in all cases is small compared to the simulation time. Moreover, the graphs show that simulation time increases linearly for application models with larger data-sets. Hence, ppim-sim is scalable and can be used for simulating larger PE configurations in tractable amounts of time. The primary reason for the scalability lies in the design of the simulator itself - it was designed to reduce overhead that affect performance for larger PE configurations. More precisely, nested loops were carefully avoided; dependency checking for executed instructions is made simple by maintaining dependency designators in the instruction itself; and repetition in executed code (when updating controllers and PE
states) as the simulation progresses is reduced by clearly delineating shared portion of information as compared to the portion of information held in individual logical elements.

Figure A.2 and Figure A.4 show the peak memory consumption of the simulator and peak memory consumption during the simulation run for the application models. The latter is an indication of memory requirement for simulating the model. In *ppim-sim*, simulator memory is primarily used up by the controllers coupled with their simulated configuration, the memory unit arrays and corresponding buffers, the processing elements and event queues. As illustrated by the graphs, when number of PEs are scaled simulator memories do not increase significantly. This means that
larger models can be efficiently simulated with the same amount of memory.

The graphs in Figure A.5 show the time taken to simulate some application models using sim-outorder and ppim-sim. sim-outorder is part of the Simplescalar tool-set: it simulates a modern microprocessor with two-level memory system, branch prediction and speculative execution. Although sim-outorder and ppim-sim are simulators of different architectures, this comparison is useful since both simulators are cycle accurate and the times are reported for the execution of the same application. As shown by the graphs, ppim-sim takes lesser time for simulation than sim-outorder. This could primarily be attributed to the careful design of ppim-sim that makes it suitable for simulating large, data-intensive program models coupled with less simu-
Figure A.4: Simulator and simulation process peak memory usage

Figure A.5: Simulation time in sim-outorder and ppim-sim
lator overhead.

A.2 Events within the Simulator
<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLETE_COMPLETE</td>
<td>On execution of complete instruction by a controller</td>
</tr>
<tr>
<td>DECODE_RESTART</td>
<td>Restart decode due to instruction dependencies</td>
</tr>
<tr>
<td>DRAM_REFRESH_COMPLETE</td>
<td>On completion of DRAM refresh cycle</td>
</tr>
<tr>
<td>DRAM_REFRESH_START</td>
<td>Begin refresh cycle of DRAM</td>
</tr>
<tr>
<td>FETCH_COMPLETE</td>
<td>A fetch completion event</td>
</tr>
<tr>
<td>FREE_EXEC</td>
<td>Free the execution unit and un-stall pipeline</td>
</tr>
<tr>
<td>LOAD_COMPLETE</td>
<td>On a scalar load completion</td>
</tr>
<tr>
<td>PES_COMM_COMPLETE</td>
<td>On completion of a PE communication instruction</td>
</tr>
<tr>
<td>PE_DECIDE_RESTART</td>
<td>Restart decode due to instruction dependencies</td>
</tr>
<tr>
<td>PE_FREE_EXEC</td>
<td>Free the parallel execution unit and un-stall pipeline</td>
</tr>
<tr>
<td>PE_LOAD_COMPLETE</td>
<td>A parallel load completion event</td>
</tr>
<tr>
<td>PE_STORE_COMPLETE</td>
<td>A parallel store completion event</td>
</tr>
<tr>
<td>PE_STALL_PIPELINE_EXEC</td>
<td>Stall pipeline (parallel execution unit busy)</td>
</tr>
<tr>
<td>PE_STALL_PIPELINE_FLOAT</td>
<td>Stall pipeline (parallel F/P queue is full)</td>
</tr>
<tr>
<td>PE_STALL_PIPELINE_LS</td>
<td>Stall pipeline (parallel load/store queue full)</td>
</tr>
<tr>
<td>PE_STALL_PIPELINE_FLOAT</td>
<td>Stall pipeline (parallel F/P queue is full)</td>
</tr>
<tr>
<td>PE_STALL_PIPELINE_FLOAT</td>
<td>Stall pipeline (parallel F/P queue is full)</td>
</tr>
<tr>
<td>SS_DRAM_REFRESH_START</td>
<td>Start the refresh of controller’s memory</td>
</tr>
<tr>
<td>SS_DRAM_REFRESH_COMPLETE</td>
<td>On completion of DRAM refresh</td>
</tr>
<tr>
<td>START_CONTROLLER</td>
<td>start a specified controller</td>
</tr>
<tr>
<td>STALL_COMPLETE</td>
<td>On completion of a pipeline stall</td>
</tr>
<tr>
<td>STALL_PIPELINE</td>
<td>Stall pipeline (due to dependency)</td>
</tr>
<tr>
<td>STALL_PIPELINE_LS</td>
<td>Stall pipeline (load/store queue is full)</td>
</tr>
<tr>
<td>STALL_PIPELINE_EXEC</td>
<td>Stall pipeline (execution unit is busy)</td>
</tr>
<tr>
<td>STALL_PIPELINE_FLOAT</td>
<td>Stall pipeline (F/P queue is full)</td>
</tr>
<tr>
<td>STALL_PIPELINE_WAIT</td>
<td>Stall pipeline (on a wait instruction)</td>
</tr>
<tr>
<td>STORE_COMPLETE</td>
<td>On a scalar store completion</td>
</tr>
<tr>
<td>WAIT_COMPLETE</td>
<td>On completion of wait instruction</td>
</tr>
</tbody>
</table>

Table A.1: Events used within *ppim-sim* simulator
Appendix A

A.1 PPIM as a Processor

A.1.1 Number Of Cycles

<table>
<thead>
<tr>
<th>Number of processing elements/Data Size</th>
<th>Superscalar cycles</th>
<th>PPIM Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>90,593,425</td>
<td>894,092</td>
</tr>
<tr>
<td>512</td>
<td>173,824,230</td>
<td>894,092</td>
</tr>
<tr>
<td>1024</td>
<td>348,476,691</td>
<td>894,092</td>
</tr>
</tbody>
</table>

Table A.1: Image Segmentation

<table>
<thead>
<tr>
<th>Number of processing elements/Data Size</th>
<th>Superscalar cycles</th>
<th>PPIM Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>222,139</td>
<td>29,534</td>
</tr>
<tr>
<td>512</td>
<td>503,237</td>
<td>31,604</td>
</tr>
<tr>
<td>1024</td>
<td>1,150,266</td>
<td>29,866</td>
</tr>
</tbody>
</table>

Table A.2: Query Handling
<table>
<thead>
<tr>
<th>Number of processing elements/Data Size</th>
<th>Superscalar cycles</th>
<th>PPIM Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>25,155,810</td>
<td>2,173,514</td>
</tr>
<tr>
<td>512</td>
<td>49,407,820</td>
<td>2,173,514</td>
</tr>
<tr>
<td>1024</td>
<td>97,823,240</td>
<td>2,201,008</td>
</tr>
</tbody>
</table>

Table A.3: Contour Extraction

<table>
<thead>
<tr>
<th>Number of processing elements/Data Size</th>
<th>Superscalar cycles</th>
<th>PPIM Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>52,066,991</td>
<td>545,366</td>
</tr>
<tr>
<td>512</td>
<td>119,754,078</td>
<td>557,193</td>
</tr>
<tr>
<td>1024</td>
<td>234,301,454</td>
<td>576,139</td>
</tr>
</tbody>
</table>

Table A.4: Fault Simulation

### A.2 PPIM as co-processor

#### A.2.1 Memory Traffic

<table>
<thead>
<tr>
<th>Number of processing elements/Data Size</th>
<th>Without PPIM</th>
<th>With PPIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>849,107,922</td>
<td>126,279,660</td>
</tr>
<tr>
<td>512</td>
<td>1,637,046,573</td>
<td>252,552,651</td>
</tr>
<tr>
<td>784</td>
<td>2,434,926,390</td>
<td>378,825,642</td>
</tr>
<tr>
<td>1024</td>
<td>3,241,077,778</td>
<td>509,718,705</td>
</tr>
</tbody>
</table>

Table A.5: Image Segmentation
<table>
<thead>
<tr>
<th>Number of processing elements/Data Size</th>
<th>Without PPIM</th>
<th>With PPIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>1,040,860</td>
<td>138,259</td>
</tr>
<tr>
<td>512</td>
<td>2,226,861</td>
<td>515,834</td>
</tr>
<tr>
<td>784</td>
<td>2,909,039</td>
<td>846,584</td>
</tr>
<tr>
<td>1024</td>
<td>3,558,385</td>
<td>1,227,334</td>
</tr>
</tbody>
</table>

Table A.6: Query Handling

### A.2.2 Number Of Cycles

<table>
<thead>
<tr>
<th>Number of processing elements/Data Size</th>
<th>Without PPIM</th>
<th>With PPIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>296,863,722</td>
<td>207,164,389</td>
</tr>
<tr>
<td>512</td>
<td>587,033,225</td>
<td>414,103,087</td>
</tr>
<tr>
<td>1024</td>
<td>1,175,563,072</td>
<td>827,980,473</td>
</tr>
</tbody>
</table>

Table A.7: Image Segmentation

<table>
<thead>
<tr>
<th>Number of processing elements/Data Size</th>
<th>Without PPIM</th>
<th>With PPIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>405,749</td>
<td>213,144</td>
</tr>
<tr>
<td>512</td>
<td>1,199,000</td>
<td>727,367</td>
</tr>
<tr>
<td>1024</td>
<td>2,736,129</td>
<td>1,615,729</td>
</tr>
</tbody>
</table>

Table A.8: Query Handling