A Distributed Multiple-SIMD Intelligent Memory*

Krishna Kumar Rangan†, Nael B. Abu-Ghazaleh‡ and Philip A. Wilsey†

†Experimental Computing Laboratory
Dept. of ECECS, University of Cincinnati
Cincinnati, OH 45221–0030

‡Computer System Research Laboratory
Dept. of CS, Binghamton University
Binghamton, NY 13902–6000

February 4, 2001

Abstract

A potential solution to the present-day 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, there are difficult challenges in developing architectures and programming models that expose the available bandwidth to end users. This paper presents the design of an intelligent memory based on a distributed data-parallel architecture with limited support for control parallelism (called PPIM). We investigate some of the relevant design issues and the success of such an architecture in supporting data-intensive applications. We also investigate PPIM as a memory access filter; applications are partitioned to execute parallel portions on the PPIM co-processor, while executing other irregular portions on the superscalar processor. A cycle-accurate simulator is developed to study the architecture and performance for some real-world data-intensive applications is compared against that of a modern superscalar processor (simulated using the *simplescalar* tool-set).

Keywords: Processing-in-memory architectures, PPIM, M-SIMD, Data-intensive applications, ppim-sim

1 Introduction

Rapid advances in VLSI technology facilitates having more functional cores on a chip. Higher bandwidth for communication and memory access, low latency interconnections among functional cores and low power consumption ratifies the the trend towards system-on-chip designs. On the other hand, such huge design space presents additional challenges to computer designers. For example, how to efficiently utilize the additional resources on a chip? Specifically, for designers in computer architecture, the challenge is how to evolve the architecture models to utilize the

*This research was partially supported by NSF grant EIA-991099 and an Ohio Board of Regents grant

significantly advanced 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 the speed mismatch. Not surprisingly, caches account for 50 - 90% of the transistor counts of modern processors. Even accommodating such large caches, performance of the processor is limited by (i) memory latency: memory will not be able to provide data to the processor at a rate that satisfies cache miss rates [21, 23, 48]. (ii) memory bandwidth: 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 [6]. Furthermore, the "typical" applications are shifting towards data-intensive, vector-oriented media type workloads [9, 27]. 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) [18].

The mismatch in memory latency/bandwidth has been well studied using separate processor and memory side improvements. Conversely, integrating 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 intuitive 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 [2]. Hence, the following questions arise: (i) How do we effectively support both data and control parallelism present in the applications taking into account the mixed process constraints? (ii) Can we do some or all of the processing in memory? (iii) How much memory traffic can we eliminate by processing in memory? In this paper, we present the design of the PPIM processor, an intelligent memory based on a distributed data-parallel architecture with limited support for control parallelism. PPIM is primarily a memory chip with logic sprinkled inside. A cycle-accurate simulator is developed to study the architecture

allowing an investigation into the above mentioned issues. Moreover, in developing the design we had to address several design issues and tradeoffs at the interface between the logic and memory; these issues are also discussed in this paper.

The remainder of this paper is organized as follows. Section 2 reviews the motivation for an alternative, logic-in-memory approach. Section 3 overviews efforts targeting the memory bottleneck problem in modern computer architecture. Section 4 elaborates on the architecture of processing elements and controllers in the PPIM processor. Section 5 describes the programming model of the PPIM processor. Section 6 describes *ppim-sim*, a cycle-accurate simulator developed to study the PPIM system. The data-intensive applications studied on the PPIM processor are also described in this section. Section 7 compares the performance of the applications to that using a traditional superscalar architecture. Finally, Section 8 presents some concluding remarks.

2 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 [25]:

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 [34]).

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 [20, 45].

4. because of the available data parallelism in multimedia workloads [9, 27], data-parallel extensions to instruction set architectures (ISAs) of modern microprocessors have appeared (for example Intel's MMX [37] and AMD's 3D now [41]). These extensions are successful in speeding up multimedia workloads [39], 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. The density of a DRAM array in a DRAM process is much higher than that of SRAM in logic process. For example, 64 MB memory implemented in a DRAM 0.35 μm process is 51 times denser than an equal process SRAM implementation [14]. But, logic implemented in DRAM process occupies more area and also suffers in terms of speed. This is because, traditionally, 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 [13, 12], Hyper-DRAM technologies [32] 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 [13].

3 Related Research

The von-Neumann memory bottleneck due to the gap between processing rate and off-chip memory access latency/limited bandwidth has been known for years. This section reviews some of the architectural efforts targeted towards reducing the effect of this problem.

3.1 Solutions within Traditional Architectures

The use of high-speed local memory (caches) that keep recently accessed memory locations has been highly successful in hiding this gap in latency because of the locality properties of memory accesses [35]. Sophisticated cache designs have been successful in hiding the memory bottleneck until recently [48]. Latency tolerance techniques such as non-blocking/split transaction caches [26] and prefetching [3] hide some of the latency, but increase the bandwidth [6].

From the memory system side, improvements in DRAM technology such as Fast Page Mode (FPM), Extended Data Out (EDO) and Synchronous DRAM (SDRAM/ESDRAM) increase the memory bandwidth by allowing accesses to the same page (the row currently in the sense-amp 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 but not latency [8]. Often, it is the memory bus that causes the differences in the performance of memory organizations [43]. The current industry standard SDRAM bus runs at 133MHz; it is being challenged by high speed bus technologies such as Rambus' RDRAM and SLDRAM. 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 [43].

3.2 Integrated Logic and DRAM

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). However, there are some difficult problems that have to be solved before the full potential of IMs is realized. For example, it is not evident how to exploit the available bandwidth. It is clear that to take advantage of a substantial portion of the bandwidth, parallel processing must be used. There are several proposals ranging from the use of using static low-level data/stream parallelism as per the vector/SIMD model [34, 40] to high-granularity full processor "tiles" with associated memory and reconfigurable communication paths [30, 47], to using only implicit instruction level parallelism (using the bandwidth for wide fetches of cache line) [42], as well as others [11, 15, 28, 29, 40, 44, 47]. A brief description of some of these projects is presented in the remainder of this section.

Vector IRAM implements a vector processor in memory. 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 [24]. The Imagine project [40] developed multimedia processor in memory architecture. Imagine operates on "stream" data that is efficiently loaded to the logic through a shared stream register file. An interesting feature of imagine is the high-speed I/O that is matched to the streaming

rate using SDRAM buffers. Elliott *et. al.* use SIMD architecture for his integrated processor-memory system, called the Computational RAM [11]. Processing elements are small, single bit ALUs operating on a bit at a time. Processing elements 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. The FlexRAM project presents numerous compute engines interleaved with DRAM macros, a RISC superscalar processor and cache on a chip [22]. 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 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 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 [28]. 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 [40] and Hydra [17] 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 [33]. 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 associative functions the operate on it to active pages. Other proposals for reconfigurable IM architectures for multimedia applications exist [15].

Our 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. Pitch-matching introduces the

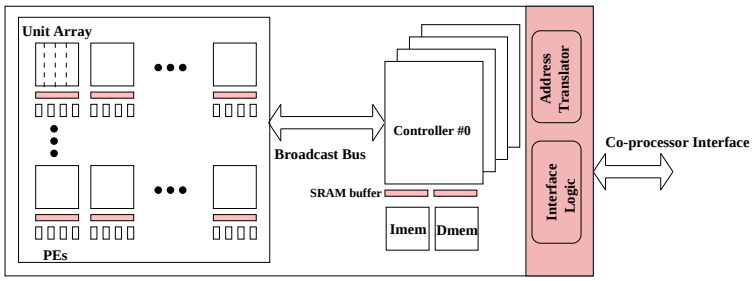


Figure 1: PPIM Processor

difficulty of having multiple PEs share a single DRAM bank – access must be serialized if they require access to different locations. PPIM also supports a limited amount of control parallelism (consistent with that available in data intensive applications [2, 10]). The study presented herein is also valuable in that the modeling is consistent with the latest mixed logic and DRAM fabrication processes. Furthermore, we believe that the solutions used at the interface between the logic and DRAM to hide the mismatch in latency are novel and can be modified for use in other process in memory organizations.

4 Architecture Overview

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 [9, 27]. Moreover, most data-intensive applications demonstrate high degree of data parallelism and limited amount of control parallelism [2, 10]. The benefits of the model must be considered against the cost of supporting it - SIMD/Vector architectures are good candidates for low overhead data parallel architectures but cannot support irregular control patterns. On the other hand, a MIMD configuration is good for such control parallelism but has large implementation overheads. Consistent with our notion of viewing PPIM as a memory chip, the logic overhead to exploit the parallelism should be less.

PPIM is a distributed Multiple-SIMD architecture. The design 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 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

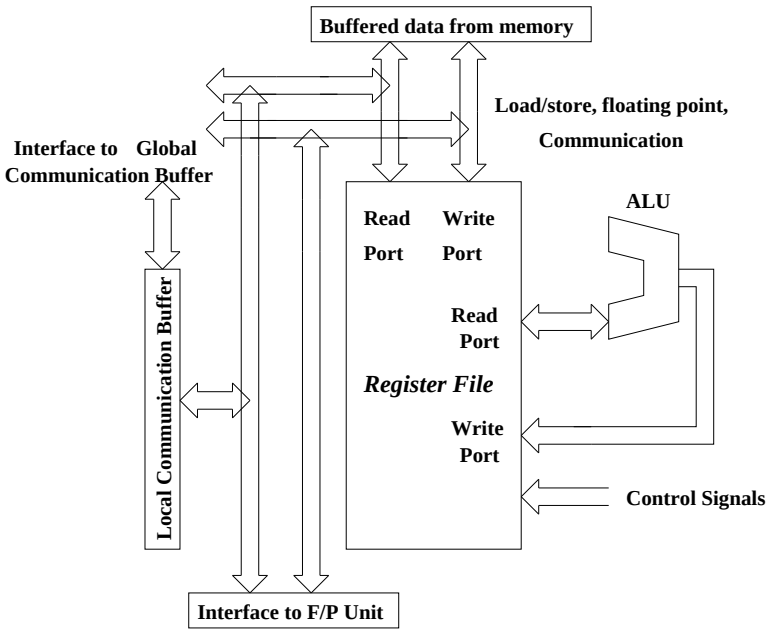


Figure 2: PE Block Diagram

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 section 5).

While the data-intensive/vector-oriented portion of the code can be processed from within the PPIM processor, scalar/irregular portion can be handled in a superscalar processor. Hence, PPIM can be viewed as a co-processor to modern superscalar processor. Based on the application, whole or part of the processing may be done on this processor. 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 1 shows the architectural overview of the PPIM processor.

4.1 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 amount of bandwidth available at the sense amplifiers is in the order of Terabytes per second. The PEs are pitch-matched to different columns of the DRAM array similar to the CRAM model [11]. 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. However, there is 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. More expensive functionality (such as floating point operations) can be shared at a coarser level to amortize the cost over more memory columns.

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. The solution we chose 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 are mapped to different column slices in the unit array.

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. This buffer serves as a single line cache for further memory accesses. Each PE consists of an ALU that performs integer arithmetic and logical operation; 16 32-bit dual read and write ported SRAM register file (which act as another level of cache for hiding memory latency); an one byte flag register, and logic to select the controller being followed. Register *15* is designated as the *communication register*. It is read and written to by the communication unit (communication among PEs is discussed in section 4.2).

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. If this register set is accessed often, number of accesses to memory is significantly reduced resulting in better performance. Hence, there is a tradeoff between speed and area in the case of SRAM and DRAM. Arithmetic and logical operations use a R/W port of the register file, and an other R/W port

is shared by communication/floating point operations (Figure 2). Load/store instructions take multiple cycles to complete if the computed row address does not match the row in SRAM buffer; they are then 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).

4.2 Controller Organization

Controllers are responsible for fetching, decoding, executing scalar instructions 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 and floating point operations. The controller with id 0 acts as the main controller. It broadcasts control signals to PEs in pure SIMD mode; during control parallelism, it forks other controllers and initiates their execution. It then waits for them to do a join after their execution is complete. Controllers have separate instruction and data memories. SRAM buffers for instruction and data act as single line cache for the controllers.

Each controller implements a simple four stage pipeline with in-order issue and out-of-order completion of instructions. The fetch stage reads instructions from the instruction memory. The decode stage decodes both PE and scalar instructions. Scalar instructions are executed in the controllers and parallel instructions are broadcast to the PEs. Each controller maintains four queues - a local load/store queue that holds scalar load/store instructions, a local floating point queue, a parallel load/store queue, and a parallel floating point/communication queue. Each queue has four entries. Load/store, floating point, and communication instructions (for PEs) that take multiple cycles are pushed into the queues. 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. The execution unit of the controller performs logical operations, integer addition, and subtraction. Data is forwarded both from execution and write-back stages of the pipeline to the decode stage, to reduce the pipeline stalls due to dependencies.

The primary goals of the communication subsystem design are: (a) it should be simple, occupying less 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

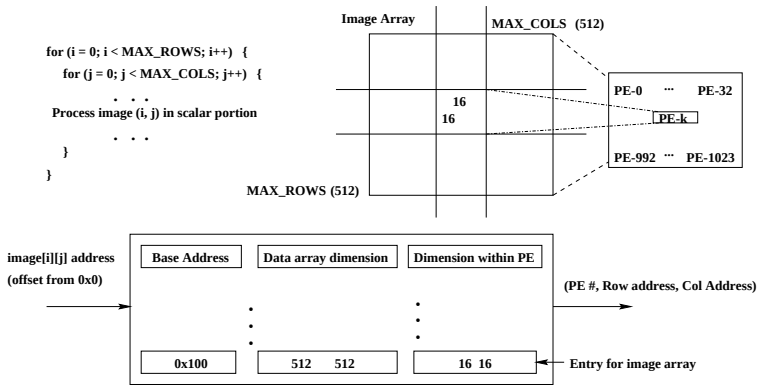


Figure 3: Translation of address generated by superscalar processor to PPIM internal address, illustrated using a 2-D image array mapping.

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 south-ward communication for the row and a east-ward communication for the column is indicated by a 0 in the leading bit.

4.3 Address Translator

For processing from within the M-SIMD programming model, data elements are partitioned and distributed in PE memories. On the other hand, memory addresses generated by the superscalar processor are based on an implicit assumption of contiguous data storage. Hence, it is necessary to perform the translation of such addresses to address recognized by 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 to intelligent memory. Accordingly, we chose to optimize the address mapping using a translator hardware integrated with PPIM (Figure 1). Since PPIM processor may co-exist with pure DRAM memory in the system, separate instructions are used to distinguish between accesses to intelligent and DRAM memory. Annotated load/store instructions such as `lw.t` and `sw.t` go through the translator.

The translator maps the address generated by the superscalar processor to PPIM internal address. 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. Translator has eight entries in the current configuration. The most significant three bits of the memory access instruction are used to index into the translator table to pick the appropriate array entry.

4.4 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, we chose to use range of values instead of single ones for the calculations.

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 [19]. Present day microprocessors typically have a density of 270000 - 310000 transistors per square mm in 0.18 μm logic technology [7]. In addition, each 1Mb embedded memory core requires 0.8 square mm of area in logic-based mixed 0.18 μm technological process [12]. Transistor densities are important because, they include both the area used by the cells and the interconnect. 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 μm technological process. However, additional area will be required for communication and floating point units. As the VLSI technology improves, PPIM design can be scaled to have more number of processing elements.

5 Programming Model

The PPIM instruction set was defined as an extension of the MIPS instruction set (with SIMD extensions for the PEs, broadcast and reduce instructions as well as multiple controller fork-join instructions). SIMD instructions are prefixed with *p.*. Data manipulation instructions are similar across the two instruction sets (controller and PE instructions); they differ primarily in control flow and activity management instructions. Control instructions such as *jump* and *branch on condition*

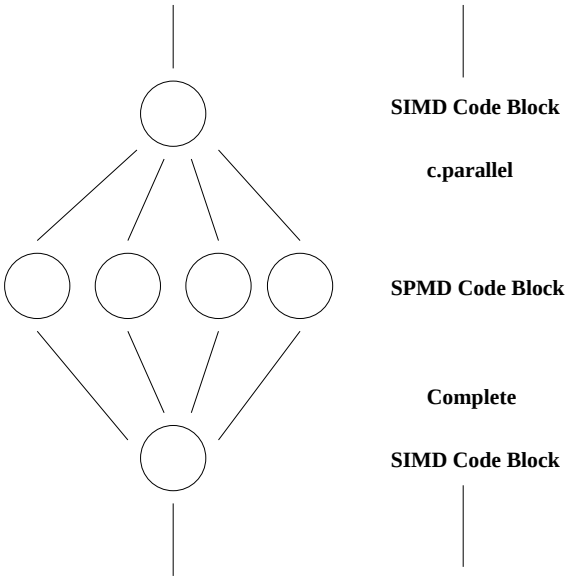


Figure 4: PPIM Programming Model

alter the program flow on the controller side. At the PE side, control flow instructions are replaced by activity management instructions which 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 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.

Within the SIMD code block, different execution paths are supported by the 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 contain the controller number to follow. For example, those PEs that successfully test the *if* portion of the conditional statement may begin listening to controller one (say). Rest of the PEs test the condition for the *else_if* part; successful PEs may begin listening to controller two and so on. The code for the default case is supported by controller zero. *ctrl* instruction writes the controller id to the flag register, which is read by the steering logic to select the controller instruction to follow. After the PEs select the controller they want to listen to, controller zero executes the *c.parallel* instruction to begin execution of participating controllers. Processor reverts to the SIMD mode after all the controllers execute the *complete* instruction. The programming model for the PPIM processor is of Single Program

Multiple Data (SPMD) type, as shows in figure 4. Specifically, PEs execute same portion of code on different data. However, each PE make take its own path in the program.

6 Experiment Environment

ppim-sim is a cycle accurate simulator for the PPIM processor. 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.

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. Execution stage of the pipeline is simulated by call to two routines: *ctrl_execute* which executes scalar instructions, followed by *pe_execute* which executes parallel instructions, for all of the active controllers. As mentioned earlier, it is a simple pipeline with in-order issue and out-of-order completion with respect to load/store, floating point and communication instructions. Most integer instructions execute in a single cycle. Load/store, communication and floating point instructions that take multiple cycles to execute are pushed into corresponding queues. The decode stage of the pipeline checks for input and output dependencies against instructions pending in the 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. All the stages of the pipeline can schedule events to occur in the future time; the simulator is both cycle-driven and event-driven. An event handler walks through the event queue and processes events meant for the current cycle time. A more detailed description of the structure and organization of the simulator is available in [38]. We note that the simulator was extensively tested for functionality and correctness. Superscalar processor is simulated using *sim-outorder* of the Simplescalar tool-set [5].

Application Development: For the PPIM processor, applications are developed in the PPIM assembly language by modifying a sequential version compiled into MIPS assembly. 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 GNU assembler, which has been re-targeted for simplescalar architecture and extended for assembling PPIM programs. The scalar

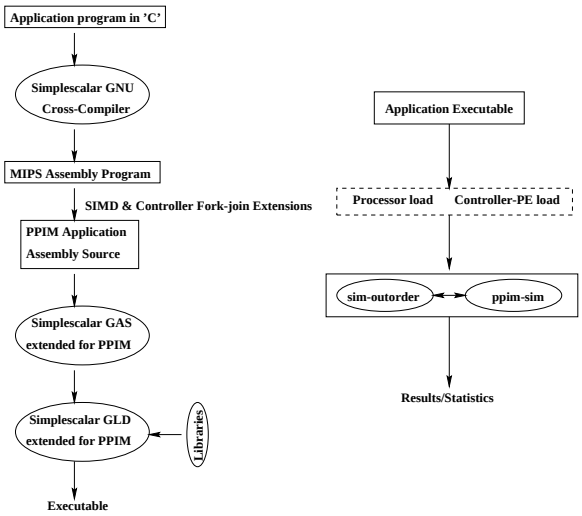


Figure 5: PPIM Simulator Overview

portion of the code is compiled using the simplescalar compiler. Figure 5 shows the overview of the simulator. The simulators use use GNU BFD library routines to read the executable, to get the program entry address, to read the size of each section and section contents. The section contents are appropriately loaded into processor, controller and PE memories. *ppim-sim* initializes the PE memories with the contents from section .psdata. Data that is not common to PEs are read from a separate file and the corresponding PE memories are initialized.

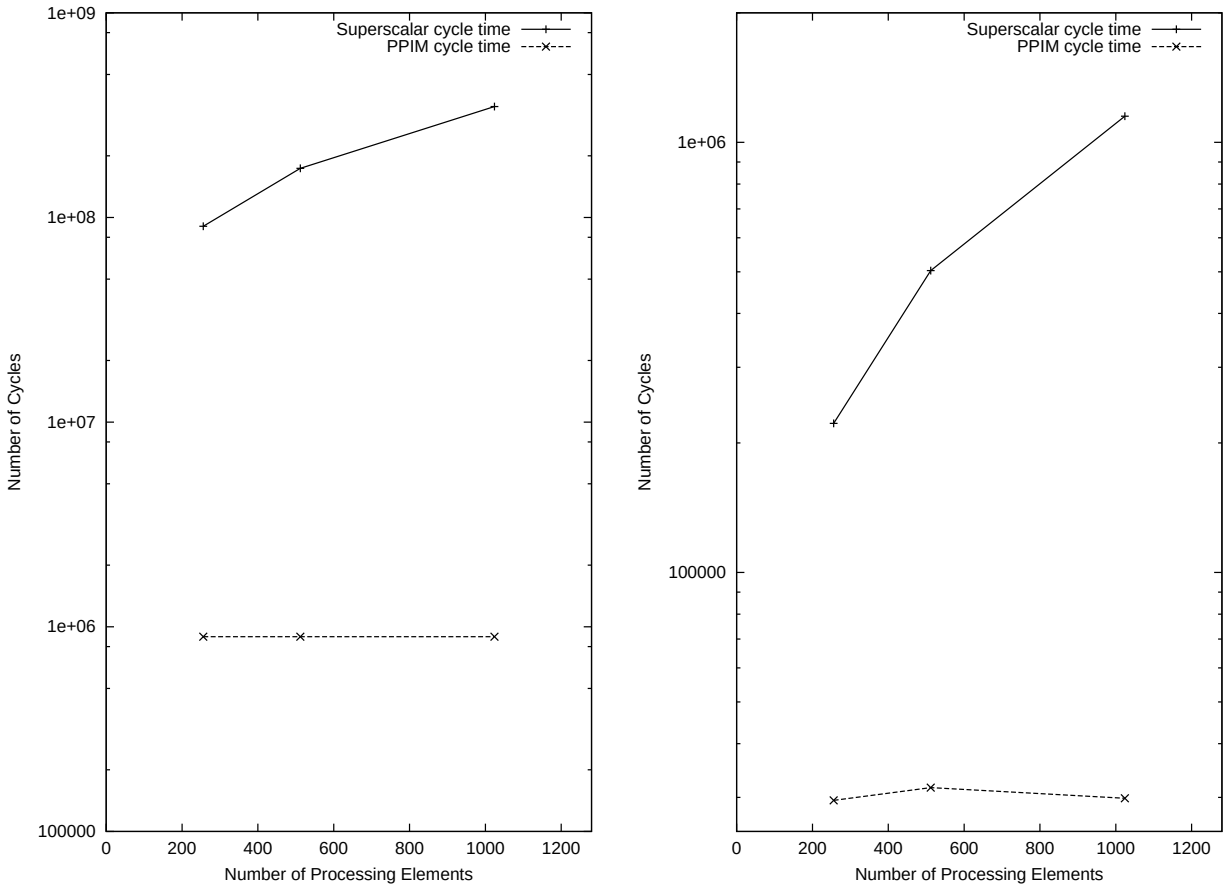
Applications: Applications in VLSI, image processing, multimedia and database domains are known to be data-intensive that severely challenge the capabilities of modern microprocessors. Some applications modeled for our study are:

Query handling: The query processor builds the query tree for the submitted queries. An execution plan is then laid out for the optimized query tree. With a number of processing elements and limited controllers, parallel execution of the query can be done. The query is executed in two phases. 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]. 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. 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.

Image Segmentation: This application divides an image into its constituent parts to identify features of interest. Image segmentation is implemented 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. Each PE operates on sub-images of the original image. To reduce communication, overlapping sub-images are loaded into PE memories. 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 [16]. 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 [16] and used to construct a binary image in which objects of interest are identified.

Contour Extraction: 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; then (ii) neighboring pixels are searched in the anti-clockwise direction to locate the next point on the contour [46, 36]. 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 [36]. 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. Both Image Segmentation and Contour Extraction used `PPM` file format for input image files.

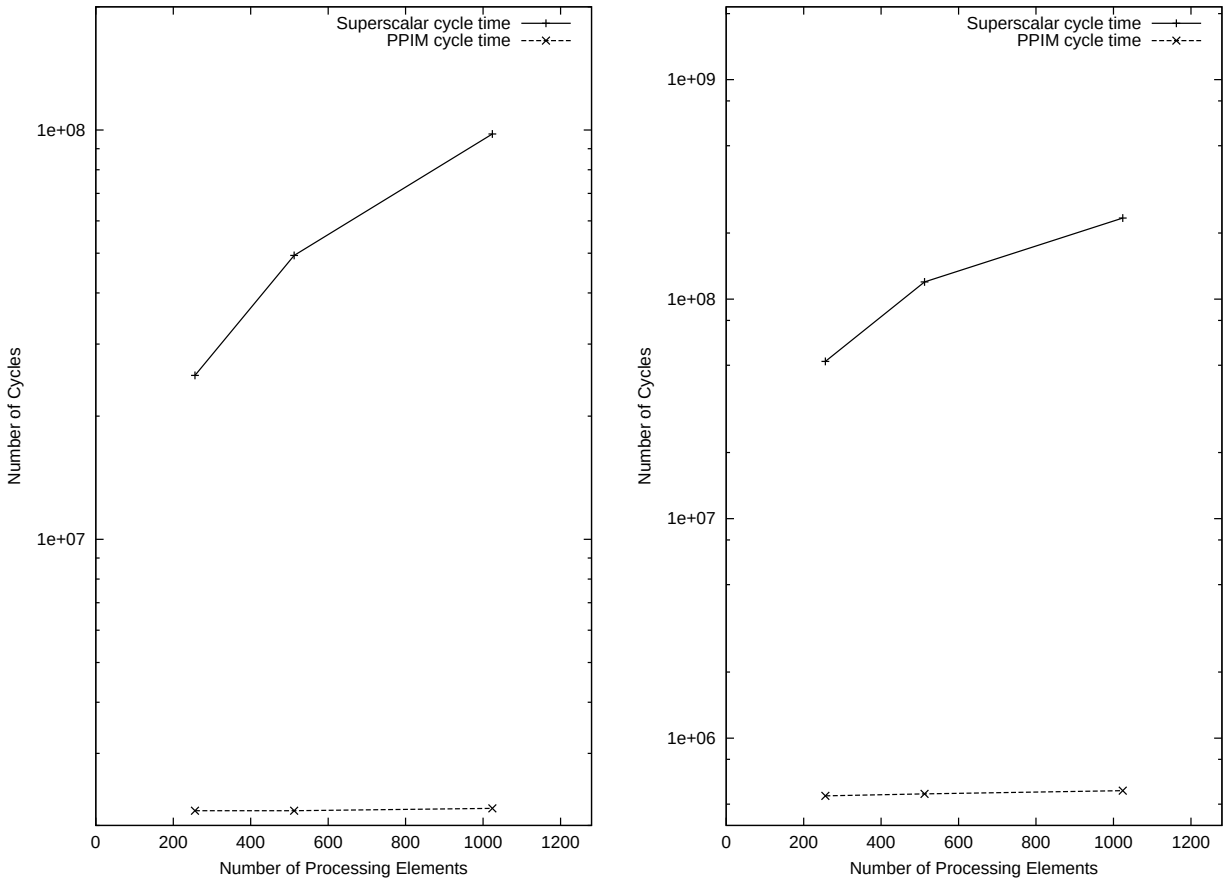
Fault simulation: is an important step in testing of logic networks. It is used to determine the faults identified by the given set of test vectors. The level ordered circuit is placed in every PE memory. The amount of PE memory needed is directly proportional to the number of nodes in the simulated 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. Simulation of the



(a) Image Segmentation **(b) Query Handling**

Figure 6: Cycle count on Superscalar and PPIM processor

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 [31]. If the number of faults in the network is more than the number of number of PEs, then fault list can be divided and simulation accomplished in more than one pass [31]. Each PE uses the Parallel Pattern Single Fault Propagation (PPSFP) algorithm for fault simulation [1]. Logic values of the nodes of the network during simulation are stored in corresponding PE memory. 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.



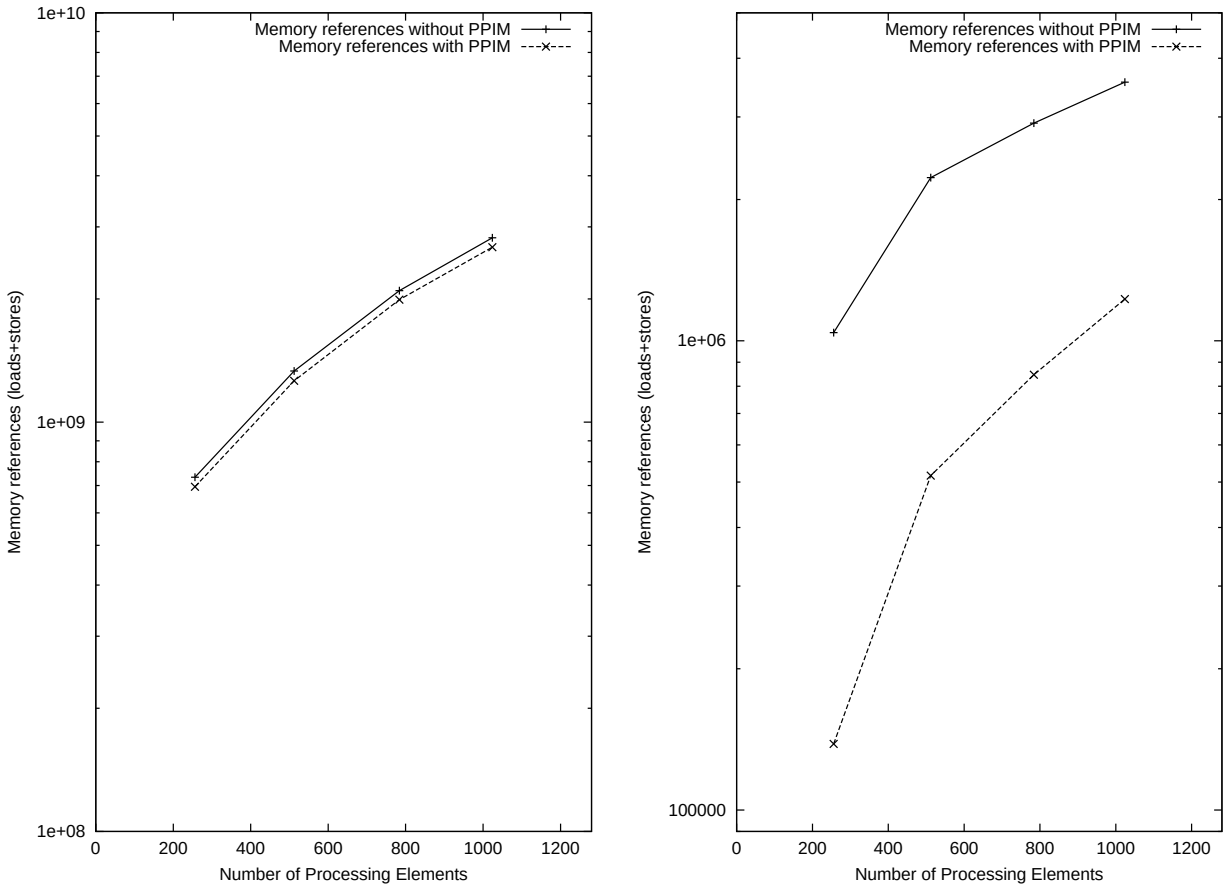
(a) Contour Extraction **(b) Fault Simulation**

Figure 7: Cycle count on Superscalar and PPIM processor

7 Experiments

Experiments were 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.

Application models studied in the PPIM processor were discussed in section 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. (ii) The algorithm is then parallelized suitable for implementation on the PPIM processor with SPMD programming model. The parallel version was tested before it was developed in PPIM assembly language. (iii) The high-level C program is compiled to MIPS assembly using the GNU compiler tools for *Simplescalar* tool-set and extended with SIMD and control parallel semantics and compiled to get the PPIM executable (figure 5).



(a) Image Segmentation **(b) Query Handling**

Figure 8: Reduction in memory traffic with PPIM co-processor

The serial implementation was compiled using GNU compiler re-targeted for *Simplescalar* architecture. The resultant executable is simulated using *sim-outorder* of the *simplescalar* simulation tool-set. PPIM binaries are executed using *ppim-sim*.

The applications were simulated using *ppim-sim*, that models the architectural configuration mentioned in section 4. 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 6 and figure 7 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. 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 the application.

Table 1 shows 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 table 1 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 over the 2-controller configuration. As table 1 illustrates, single-scalar multiple controllers are beneficial in exploiting the control parallelism present in the applications. Presently, only a single dominant control parallel portion is exploited 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 it would not be beneficial; the cost of initializing the controllers, activating and de-activating would dominate.

The experiments discussed so far execute entire applications on PPIM processor. On the other hand, it can be seen that floating point and irregular applications (that do not map well to M-SIMD programming model) would have to be run on the superscalar processor. In such cases, the application is partitioned into portions that can be run on the superscalar and intelligent memory. To study the memory traffic reduction using PPIM co-processing while running irregular applications, we extended `Image Segmentation` and `Query Handling` applications discussed in Section 6. 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

Application Name	Pure SIMD	PPIM 2-controllers	Utilization (over SIMD)	PPIM 4-controllers	Utilization (over 2-controllers)
Fault Simulation	633923	603495	5%	545366	10%
Image Segmentation	1019609	979845	4%	894092	9%
Query Handling	34339	32210	6%	29534	8%
Contour Extraction	2376693	2288756	4%	2173514	5%

Table 1: Number of cycles taken by applications executing on a single, two and four controller configurations of the PPIM processor.

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 collinear points are identified as the curves that intersect at a common point [16]. 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 get executed on the superscalar processor. PPIM co-processor shared the virtual address space of the processor for the studied applications. Figure 8 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 query handling). Hence, a PPIM based computing system could be used to flexibly support general purpose applications, while providing increased speedup and reduced traffic on the system bus for the data-intensive ones.

8 Concluding Remarks

In the light of the current technological process, integrating logic and memory on a chip is a viable solution to solve the memory bottleneck problem. We presented PPIM architectural design consisting of four simple, single-scalar controllers and 1024 Processing elements integrated with memory on a chip. We discussed *ppim-sim*, that implements the PPIM architectural model in software and capable of simulating PPIM binary programs. We also presented the experiments conducted to quantity the performance improvement of some data-intensive applications as compared to modern superscalar processors. As illustrated by the experiments, PPIM architectural approach is conducive to applications that operate on large amounts of data. Utilizing simple,

multiple controllers is effective for exploiting the control parallelism to provide increased speedup (upto 13% for the studied applications). For general purpose irregular applications, co-processing using PPIM significantly reduces the memory traffic for the data-intensive portions. In addition, in-place floating point units on the PPIM processor would help further reduce the memory traffic. We are currently investigating feasible floating point unit designs for the PPIM processor.

References

[1] Abramovici, M., Friedman, A. D., and Breuer, M. A. *Digital Systems Testing and Testable Design*. Benjamin/Cummings, 1993.

[2] Allen, J. D., Garg, V., and Schimmel, D. E. Analysis of Control Parallelism in SIMD Instruction Streams. In *Proceedings of 5th Symposium on Parallel and Distributed Processing* (Dec. 1993), pp. 383–390.

[3] Baer, J., and Chen, T. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91* (Nov. 1991), pp. 176–186.

[4] Bitton, D., Boral, H. B., DeWitt, J., and Wilkinson, W. K. Parallel algorithms for the execution of relational database operations. In *ACM Transactions on Database Systems* (Sept. 1983), vol. 8, pp. 324–353.

[5] Burger, D., and Austin, T. The Simplescalar Architectural Research Toolset, Version 2.0, June 1997.

[6] Burger, D., Goodman, J. R., and Kagi, A. Memory Bandwidth Limitations of Future Microprocessors. In *23rd International Symposium on Computer Architecture* (May 1996).

[7] The Tech Report: CPU Heat Dissipation Table, 2000. Available from: `http://www.tech-report.com/cpu/`.

[8] Cuppu, V., Jacob, B., Davis, B., and Mudge, T. A Performance Comparison on contemporary DRAM architectures. In *26rd International Symposium on Computer Architecture* (May 1999), pp. 222–233.

[9] Diefendorff, K., and Dubey, P. How Multimedia Workloads will Change Processor Design. *IEEE Computer* (Sept. 1997), 43–45.

[10] Dietz, H. G., and Cohen, W. E. A Massively Parallel MIMD implemented by SIMD Hardware. *Technical Report No. TR-EE 92-4, School of Engineering, Purdue University* (Jan. 1992).

[11] Elliott, D. G., Stumm, M., Snelgrove, W. M., Cojocaru, C., and Mckenzie, R. Computational RAM: Implementing Processors in Memory. In *IEEE design and Test of Computers* (1999), pp. 32–40.

[12] IBM ASIC Standard Cell/Gate Array Products. (http://www.chips.ibm.com/products/asics/products/edram/).

[13] Technology and Manufacturing - Embedded DRAM. (http://www.tsmc.com/technology/dram.html).

[14] Fromm, R., Perissakis, S., Cardwell, N., Kozytakis, C., McGaughy, B., Patterson, D., Anderson, T., and Yelick, K. The Energy Efficiency of IRAM architectures. In *24th International Symposium on Computer Architecture* (June 1997).

[15] Goldstein, S., Schmit, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R., and Laufer, R. PipeRench: A coprocessor for streaming multimedia acceleration. In *International Symposium on Computer Architecture (ISCA 99)* (May 1999), pp. 28–39.

[16] Gonzalez, R. C. *Digital Image Processing.* Addison-Wesley Publishing Company, 1993.

[17] Hammond, L. The Stanford Hydra Chip. In *In Proceedings of Hot Chips 11* (Aug. 1999), pp. 23 – 31.

[18] Henning, J. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer* (July 2000), 28–35.

[19] Processor Hall of Fame: Technical Specifications, 2000. Available from: `http://www.intel.com/intel/museum/25anniv/hof/tspecs.htm`.

[20] Intel mobile petium III processor, 2000. `http://www.intel.com/mobile/pentiumIII/ist.htm`.

[21] Jouppi, N. P., and Rangarathan, P. The relative importance of memory latency, bandwidth and branch limits to performance. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, International Symposium on Computer Architecture (ISCA 97)* (June 1997).

[22] Kang, Y., Huang, W., Yoo, S., Keen, D., Ge, Z., Lam, V., Pattnaik, P., and Torrellas, J. Flexram: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design* (Oct. 1999).

[23] Kogge, P., Sterling, T., and Gao, G. Processing in memory: Chips to petaflops. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, International Symposium on Computer Architecture (ISCA 97)* (June 1997).

[24] Kozyrskis, C. Vector IRAM: ISA and micro-architecture. In *IEEE Computer Elements Workshop* (June 1998). (Slides available on www at: http://iram.cs.berkeley.edu/slides/IRAM.VAIL98.pdf).

[25] Kozyrskis, C., and Patterson, D. A new direction in computer architecture research. *IEEE Computer* (Nov. 1998).

[26] KROFT, D. Lockup free instruction fetch/prefetch cache organization. In *International Symposium on Computer Architecture (ISCA 86)* (June 1986), pp. 49–54.

[27] LEE, R., AND SMITH, M. Media processing: A new design target. *IEEE Micro* (Aug. 1996), 6–9.

[28] MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W., AND HOROWITZ, M. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)* (June 2000), pp. 161–171.

[29] MARGOLUS, N. An embedded dram architecture for large-scale spatial-lattice computations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)* (June 2000), pp. 149–160.

[30] MURAKAMI, K., INOUE, K., AND MIYAJIMA, H. PPRAM: (parallel processing RAM): A merged DRAM/logic system-LSI architecture. In *1997 International Conference on Solid State Devices and Materials (SSDM'97)* (Sept. 1997).

[31] NARAYANAN, V., AND PITCHUMANI, V. A massively parallel algorithm for fault simulation on the connection machine. In *26th ACM/IEEE Design Automation Conference* (1989), pp. 734–737.

[32] NUNOMURA, Y., SHIMIZU, T., AND TOMISAWA, O. M32R/D - Integrating DRAM and Microprocessor. *IEEE Micro* (1997), 40–47.

[33] OSKIN, M., CHONG, F. T., AND SHERWOOD, T. Active Pages: A Computation Model for Intelligent Memory. In *International Symposium on Computer Architecture* (1998).

[34] PATTERSON, D., ANDERSON, T., CARDWELL, N., FROMM, R., KEETON, K., KOZYRAKIS, C., THOMAS, R., AND YELICK, K. A case for Intelligent RAM. *IEEE Micro* (March/April 1997), 34–44.

[35] PATTERSON, D. A., AND HENNESSEY, J. L. *Computer Architecture: A Quantitative Approach*, 2^{nd} ed. Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1996.

[36] PAVLIDIS, T. *Graphics and Image Processing.* Computer Science Press, 1982.

[37] PELEG, A., AND WESER, U. MMX technology extension to the intel architecture. *IEEE Micro* (Aug. 1996), 51–59.

[38] RANGAN, K. K., PISOLKAR, N., ABU-GHAZALEH, N. B., AND WILSEY, P. A. Ppim-sim: An efficient simulator for a parallel processor in memory. In *Proceedings of 34th Annual Simulation Symposium [forthcoming]* (Apr. 2001).

[39] RANGANATHAN, P., ADVE, S., AND JOUPPI, N. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)* (May 1999), pp. 124–135.

[40] Rixner, S., Dally, W., Kapasi, U., Khailany, B., Lopez-Langunas, A., Mattson, P., and Owens, J. Bandwidth efficient architecture for media processing. In *Proceedings for the 31st International Symposium on Microarchitecture* (1998).

[41] S. Oberman et al. AMD 3DNow! technology and the K6-2 microprocessor. In *Proceedings of Hot Chips 10* (1998).

[42] Salsbury, A., Pong, F., and Nowatzyk, A. Missing the memory wall: The case for processor/memory integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* (May 1996), pp. 90–101.

[43] Schwartzman, N. RAMBUS DRAM performance, 2000. `http://www.usa.samsungsemi.com/new/rdram-whitepaper.htm`.

[44] Silicon-access: Terabit routers and smart memory technology, 2000. `http://www.siliconaccess.com/whitepaper.pdf`.

[45] The technology behind the crusoe processor, 2000. Transmeta's Mobile Processor Whitepaper; available from: `http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf`.

[46] Tuomenoksa, D. L., III, G. B. A., Siegel, H. J., and Mitchell, O. R. A parallel algorithm for contour extraction: Advantages and architectural implications. In *IEEE Computer Society Symposium on Computer Vision and Pattern Recognition* (June 1983), pp. 336–374.

[47] Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., and Agarwal, A. Baring it all to software: RAW machines. *IEEE Computer* (Sept. 1997), 86–93.

[48] Wulf, W., and McKee, S. Hitting the memory wall: Implications of the obvious. *Computer Architecture News 23* (Mar. 1995).