FlexRAM: Toward an Advanced Intelligent Memory System[1]

Yi Kang, **Wei Huang**, **Seung-Moon Yoo**, **Diana Keen**,
Zhenzhou Ge, **Vinh Lam**, **Pratap Pattnaik**[2] and **Josep Torrellas**

Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801
yi-kang,weihuang,yoo,dmfrankl,zge,lam,torrellas@cs.uiuc.edu pratap@us.ibm.com
http://iacoma.cs.uiuc.edu

Abstract

Major advances in Merged Logic DRAM (MLD) technology coupled with the popularization of memory-intensive applications provide fertile ground for architectures based on Intelligent Memory (IRAM) or Processors-in-Memory (PIM). The contribution of this paper is to explore one way to use the current state-of-the-art MLD technology for general-purpose computers. To satisfy requirements of general purpose and low programming cost, we place the PIM chips in the memory system and let them default to plain DRAM if the application is not enabled for intelligent memory. Since wide usability is crucial, we identify and analyze a range of real applications for PIM. Based on the requirements of these applications and current technological constraints, we design a PIM chip and a PIM-based memory system. We call the chip *FlexRAM*. We describe *FlexRAM*'s design and floorplan, and the resulting memory system. Evaluation of the system through simulations shows that 4 *FlexRAM* chips often allow a workstation to run 25-40 times faster.

1 Introduction

Advances in VLSI technology are delivering dramatic increases in the number of transistors that can be integrated on a chip [36]. Given that current computers waste much time transferring data between compute and storage units, it is appealing to combine significant processing power and a large amount of DRAM memory in the same chip. This approach has been called Intelligent Memory (IRAM) [27] or Processors-in-Memory (PIM) [19].

Integrating logic and DRAM in the same chip is accomplished with a Merged Logic DRAM (MLD) process and has considerable technological challenges [30]. While there have been some early PIM research prototypes [19], it is only recently that MLD technology has been considered promising enough to be strongly supported by big foundries. Indeed, Mitsubishi, IBM, Samsung, Toshiba, and others are able to fabricate MLD chips in 0.25 or 0.18 μm technology [15, 25]. Consequently, given that we have available a large silicon area that can integrate dramatic compute and storage capabilities, the question arises as to how to exploit this technology best?

There has been much recent work in this area [29] (see Section 7). One approach is to combine a good-sized processor, caches, and much DRAM on a chip and make the chip the main compute engine in the machine [4, 19, 27]. A difficulty with this approach is that traditional processors are not designed to exploit a huge amount of very close DRAM any differently than a large on-chip L2 cache. As a result, the system delivers only incremental speedups compared to a processor with a large on-chip L2 cache [4]. This problem can be partially addressed by adding a vector processor [21] or extra processors [19] on chip: more bandwidth can now be extracted from the memory. However, the system may now be hard to program.

Another approach is to use the MLD technology to build a specialized engine. Such an engine could, for example, run vector applications [17], process data beside the disk [28], or control ATM switches [5]. Finally, a third approach, which we and other groups [9, 26] take, is for the PIM chips to take the place of memory chips in a workstation or server. The PIM chips can then process the most memory-intensive parts of the application.

Interestingly, advances in MLD technology come at a time when the application base is evolving toward domains that can exploit the new technology well. Indeed, many of the applications in relatively new domains like multimedia or data mining are quite memory intensive. This makes them amenable to PIM computation.

In this paper, we explore one way to use the current state-of-the-art MLD technology for general-purpose workstations and servers. Because we are interested in general purpose and low programming cost, we place the PIM chips in the memory system and let them default to plain DRAM if the application is not enabled for intelligent memory. Because we examine present-day technology, we do not consider any reconfigurability like the Active Pages system [26]. Furthermore, since wide usability is crucial, we spend much effort identifying and analyzing a wide range of real applications for PIM. Based on the requirements of these applications and current technological constraints, we design a PIM chip called *FlexRAM* and its memory system. We describe *FlexRAM*'s design and floorplan. Finally, evaluation of the system through simulations shows that 4 *FlexRAM* chips often allow a workstation to run 25-40 times faster.

This paper is organized as follows: Section 2 describes our approach to intelligent memory; Section 3 describes a range of applications for intelligent memory and their architectural requirements; Section 4 describes the resulting architecture; Section 5 presents a chip floorplan and discusses implementation issues; Section 6 evaluates the architecture; and finally Section 7 discusses related work.

2 Principles

The design of our intelligent memory is guided by the following principles.

Extract high DRAM bandwidth. Simply including a conventional wide-issue superscalar in a DRAM chip has delivered disappointing performance [4]. For higher performance, we need to extract more DRAM bandwidth. Consequently, we embed in the DRAM chip many simple processing elements, all of which can access memory concurrently.

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9612099, and MIP-9619351, DARPA Contract DABT63-95-C-0097, and gifts from IBM.

[2]Pratap Pattnaik is with IBM T.J. Watson Research Center, Yorktown Heights, NY.

Run legacy codes. Since many existing programs cannot be recompiled, PIM chips cannot generally replace the main commodity microprocessor of the workstation. Instead, we propose that these chips take the place of memory chips and appear as plain DRAM to applications that are not enabled for intelligent memory.

Minimize DRAM cost increase. The processing engines embedded in the DRAM must be extremely simple to minimize losses in memory density, performance, and power consumption.

Be general purpose. The PIM chips should not hard-wire a few algorithms and become special-purpose co-processors. Instead, the in-memory processing should be usable in as wide a range of algorithms as possible.

Overall, we envision a chip that is connected as a plain DRAM chip and can appear as one to the applications. We call it *FlexRAM*. It contains many very simple compute engines called *P.Arrays* that are finely interleaved with DRAM macrocells. To avoid incorporating extensive interconnection among P.Arrays, we restrict each P.Array to see only a portion of the on-chip memory. To increase the usability of P.Arrays, we also include a low-issue superscalar RISC core on chip. This processor, called *P.Mem*, coordinates the P.Arrays and executes serial tasks. Without the *P.Mem*, these tasks would need to be performed by the commodity microprocessor in the workstation or server (*P.Host*) at a much higher cost. Many *FlexRAM* chips can be connected to the commodity memory bus of a workstation or server. A general view of the envisioned architecture is shown in Figure 1.

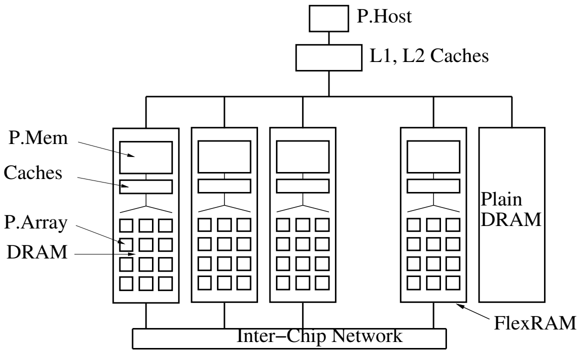

Figure 1: Overall organization of a *FlexRAM*-based memory system.

3 Application Requirements

Because our intelligent memory should be as general-purpose as possible, we use a wide range of application domains to flesh out its architecture. Our goal is to determine what architectural features are required. In this section, we outline several memory-intensive applications. Appendix A presents the detailed mapping of their algorithms to *FlexRAM*. Based on the considerations in Appendix A, Section 3.1 summarizes the architectural requirements on *FlexRAM*.

Data Mining: Tree Generation, Tree Deployment, and Neural Networks. Two major groups of classification algorithms in data mining [3] are those that process decision trees and those that process neural networks. A decision tree is a tree-shaped data structure that, when applied to a record, determines whether or not the record belongs to a certain subgroup. The tree nodes are questions about the data in the record. The main tree problems are tree generation (assembling the tree by examining a subset of the records) and tree deployment (classifying the records by applying the tree to all the records) [31].

Neural networks classify records according to their proximity with each other in the N-dimensional space. The BSOM algorithm is frequently used for this problem [23].

Computational Biology: Protein Pattern Matching. This important problem in computational biology consists of matching an amino acid query string against a protein database. We look for sections of the query string that are similar to sections of proteins. Because of mutations, the problem is more complicated than simply performing sequential comparisons. The BLAST algorithm is frequently used [1].

Decision Support Systems: TPC-D. TPC-D is a standard decision support system application [37]. It includes several queries that are broken down into memory-intensive operations like scan, join, sort, group, and aggregate. Due to its size, TPC-D needs several *FlexRAM* chips to hold the data. In our analysis, we assume that we have enough physical memory to hold the whole database or that I/O operations are cleverly overlapped to eliminate I/O stall.

Multimedia: MPEG-2 Motion Estimation. Multimedia has many memory-intensive, stream-based problems that are suitable for PIM. Motion estimation is a popular kernel. Its goal is to find the differences between two pixel images. The most common algorithm used compares the two images on a block-by-block basis.

Financial Modeling and Molecular Dynamics. Unfortunately, these applications are floating-point intensive and our target technology is currently not dense enough to include heavy floating-point logic on the memory chip.

3.1 Architectural Requirements

Based on the analysis of the applications in Appendix A, *FlexRAM* should include the following support:

P.Array Engines. The high parallelism of the applications suggests including many P.Arrays. However, to keep the ratio of logic area to DRAM area moderate, each individual P.Array must be very simple. For this reason, we can only support integer arithmetic. Some of the applications perform significant computation, which suggests using 32-bit arithmetic in P.Arrays. The neural network would benefit from multiplication support in P.Arrays. However, given the area cost of multipliers, it is best if several P.Arrays share a multiplier. Division is too rare and expensive to support in hardware. Finally, to effectively support a wide range of applications, the P.Arrays should work in a Single Program Multiple Data (SPMD) mode; plain Single Instruction Multiple Data (SIMD) mode is too restrictive and inefficient [13].

P.Mem Processor. P.Mem can be a low-issue superscalar, with floating-point support and small primary data and instruction caches. A secondary cache would take too much area. For maximum programmability, P.Mems and P.Arrays should use virtual addressing, and share some virtual addresses with the P.Host.

Memory Structure. *FlexRAM* is a memory device in the first place, replacing regular DRAMs without modifying existing system specifications. However, it should fit in a memory standard that includes additional power and ground signals, so on-chip processing can be enabled. One such standard is Rambus [6, 7]. Consequently, we include a Rambus-compatible interface on chip.

To provide high bandwidth to a set of P.Arrays, a multi-arrayed DRAM architecture should be used. Fortunately, as memory technology advances to Gbit generations, such architectures become common. In addition, in many applications, each P.Array works on several different localities at the same time. For example, it often accesses a small, reused data structure, and a large database-like structure that is hardly reused. It may also access several scalar variables. Consequently, to intercept most data accesses, the memory array associated with a P.Array has several row buffers. No caches are used.

Each P.Array only needs a small instruction memory because the codes running on P.Arrays are short. In addition, given the simplicity of P.Arrays, we can use 16-bit instructions. Finally, to save space, several neighbor P.Arrays can share the instruction memory.

Communication between P.Host and *FlexRAM*. The P.Host should start the P.Mems with a simple write to a special memory-mapped location. The P.Host should pass the address of the routine to start executing in memory. A master P.Mem should inform the P.Host when the job is completed. However, P.Mems cannot directly invoke the P.Host because memories cannot be masters of the memory bus. Consequently, to receive information from the P.Mems, the P.Host or the memory controller must poll on a location that the P.Mems can set. We must make use of the programmability offered by Rambus to ensure that this polling is efficient.

Intra-Chip Communication. The applications exhibit several different communication patterns. One of them is global communication between the P.Arrays. In this case, the P.Mem must shuffle data between memories visible to different P.Arrays. Consequently, the P.Mem must be able to access all the on-chip memory and communicate with any individual P.Array in a write-read step through memory.

In applications like protein matching and tree generation, we need an inter-P.Array ring connection, so that a P.Array can communicate with its left and right neighbors. This communication is enabled by allowing each P.Array to see its two neighbors' memory. The motion estimation application could benefit from a more connected network but, if each P.Array gets a set of full rows of pixels, a ring connection suffices. Aside from motion estimation, we have not found applications that could benefit from more complicated networks like a mesh. Even if we found them, however, it is unclear whether the costly message buffering and routing support required in meshes would be worth its area consumption.

In addition to these patterns, many applications require efficient broadcast from the P.Mem to all P.Arrays. Furthermore, a fast notification mechanism from each P.Array to the P.Mem is also useful. The combination of both primitives can be the basis for a global P.Array barrier.

Inter-Chip Communication. For applications that do not fit in a single chip, a P.Mem must be able to access data from other *FlexRAM* chips. However, since a P.Mem cannot be the master of the memory bus, we need an additional interconnection between *FlexRAM* chips (Figure 1).

I/O Bandwidth. Finally, the applications analyzed are not generally I/O bound. The neural network, tree generation, and motion estimation perform a significant amount of processing per data element. Other applications like the tree deployment and TPC-D do less but still execute many instructions per load of input data. In any case, there are ways to overlap the I/O in some chips with computation in the same or other chips. Furthermore, if the input data fits in memory and is reused across queries, the I/O time is negligible.

4 Architectural Design

Based on the application-driven rationale just described, we proceed to a detailed design of the *FlexRAM* architecture. We assume 0.18 μm MLD technology with 400 MHz logic.

4.1 Memory

Each *FlexRAM* chip has 64 Mbytes of memory that are organized as 16Mx32 bits. We estimate that, to P.Host accesses, the chip with a Rambus interface offers an access time of 40 ns for row misses and 20 ns for row hits at an I/O frequency of 400 MHz.

In the inside, the DRAM is organized in 64 1-Mbyte banks. Each bank is associated with one P.Array and has a single port. In addition, since P.Arrays do not have caches, each bank has several row buffers. Based on an analysis of the applications, a good design includes 3 2-Kbyte row buffers per bank [13]. We use random row buffer replacement. These row buffers, although costly, are useful to capture important program localities [35]. A P.Array access to memory should take 10 and 20 ns in a row buffer hit and miss respectively.

With so many processing units on chip, contention for memory may occur. Specifically, a DRAM bank may be accessed by the P.Host, the local P.Mem, or a remote P.Mem through the global on-chip bus. It can also be accessed by the local P.Array or by a neighbor P.Array. There is a single port per bank and a switch that connects the port to one of these sources. Consequently, the finite state machine in the switch must select the correct source.

The *FlexRAM* chip also contains SRAM instruction memory to hold the P.Array code. Each group of 4 P.Arrays shares an 8-Kbyte 4-ported instruction memory. While sharing instruction memory increases port requirements, it saves overall area. 8 Kbytes can store a sizable program of 16-bit wide instructions. We have aggressively assumed that the SRAM can have a 2.5 ns access time to match a 400 MHz P.Array. These instruction memories are loaded by the P.Mem.

4.2 P.Array

Each chip has 64 P.Arrays. Each P.Array is a very simple, 32-bit fixed-point RISC engine. It has a 4-stage pipeline with 16 general-purpose registers, no caches, and a 1-entry store buffer. Each P.Array shares a multiplier with 3 other P.Arrays. P.Arrays cycle at 400 MHz and have 28 16-bit instructions. Each P.Array is associated with 1 Mbyte of DRAM and can also access the 1 Mbyte of its two neighbors, forming a logical ring. While neighbor P.Arrays communicate through shared-memory, non-neighbor communication requires the involvement of P.Mem, which can access all memory. Figure 2 shows the P.Array datapath.

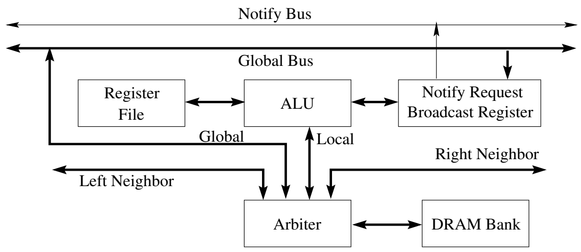

Figure 2: P.Array datapath.

There is a construct for a global P.Array barrier that uses two primitives: a *notification* from each P.Array to the P.Mem, and a *broadcast* from the P.Mem to all P.Arrays. Notification uses lines that go from each P.Array to one bit in P.Mem's Notify Register. Each P.Array can set one bit. The P.Mem can poll the register or be interrupted when certain bit patterns occur. The P.Mem can broadcast a 32-bit word to all P.Arrays. In each P.Array, a register receives the word and a broadcast flag is set. The P.Array can poll the flag to detect when the data has arrived. Broadcasting into memory instead of into a register is not supported because the memories of some P.Arrays could be busy, forcing the broadcast operation to wait.

4.3 P.Mem, Network, and Interface

The P.Mem is a two-issue superscalar with floating-point support like the IBM PowerPC 603 [8] and with 16 Kbyte I- and D-caches. We expect that it can cycle at 400 MHz. The processor interface is modified to support the broadcast and notification primitives via memory-mapped locations.

P.Mems communicate with each other via an inter-chip network. We minimize the network logic included on chip to make the network topology more flexible. Flexibility is important because different topologies are best with different numbers of chips. Consequently, each chip only includes an In and an Out SRAM queue and simple message packaging logic. Routing is performed by an off-chip router IC. The In and Out ports have 16 data pins each and cycle at 800 MHz. Each of the on-chip queues is 32-bit wide and can hold two 64-byte cache lines.

In a multichip *FlexRAM* memory, all memory is shared and visible to all P.Mems. Requests between chips are transferred through the inter-chip network. When a P.Mem references a location, it caches the memory line. However, there is no hardware to enforce cache coherence between P.Mems or between P.Mems and P.Host. It is up to the programmer to flush the data from the cache before it is used by another processor. A higher-end design could provide coherence support.

Finally, communication between P.Host and P.Mems is implemented by using special features and reserved code words from the Rambus definition. We use Rambus because it allows two-way control signals. We program the Rambus memory controller to support the following protocol. The P.Host starts each P.Mem by writing on a predefined register in the Rambus interface of each chip. The value written is the address of the code to execute. When the P.Host needs to wait for the P.Mems, it reads another predefined register of the Rambus interface in the master P.Mem chip. If the master P.Mem is not finished, a special acknowledgment is returned to the memory controller. The latter buffers the message and keeps retrying the read at regular intervals. This retry operation is transparent to the P.Host. When a retry finds that the master P.Mem is finished, the controller informs the P.Host. This message constitutes the reply to the initial P.Host request.

4.4 Address Translation

To enhance programmability, P.Mems and P.Arrays use virtual memory. For a given program, they share a range of virtual addresses with the P.Host. In the program, the programmer specifies how the data structures are distributed. Eventually, when compiler technology is good enough, this will be done by the compiler.

In each P.Mem, the virtual to physical translations are stored in the TLB and are backed up in a page table in memory shared by all P.Mems. In each P.Array, to minimize area overhead, these translations are stored in an 8-entry fully-associative TLB. If a TLB miss occurs, the P.Array accesses the memory area that keeps the complete mapping information for its own DRAM bank and its two neighbors'. Unlike in the TLB, the mapping information for P.Arrays in the memory is not organized in a table of virtual and physical page numbers. Instead, it is organized in a table with base and limit page number for each data structure. Given that each data structure within a bank is allocated in a contiguous manner, such organization of the mapping information is quite efficient. It minimizes the memory space and the time necessary to sequentially traverse the mappings. Once the correct data structure is found in the table, a simple computation will produce the correct entry to enter in the TLB. Note that the P.Array TLB is accessed by data references only; instruction fetches proceed to the instruction memory without translation. Consequently, the TLB uses very little area.

Finally, in our system, we try to avoid replacing pages that contain shared data. If any of these pages were replaced, we would need to send interrupts to invalidate TLB entries to keep the page mappings consistent. To avoid these problems, at the beginning of the program, we pin in memory the pages with shared data. Pages with private data can be replaced.

5 Chip Implementation

Implementing *FlexRAM* requires careful circuit design and much process technology support. There are still issues open to research, which are beyond the scope of this paper, like reducing the impact of the heat and noise induced by the logic portion of the chip on the memory part of it. Here, we concentrate on floorplan and clock issues. We also estimate the area required and the power consumption.

5.1 Floorplan

The chip layout, shown in Figure 3-(a), is composed of the P.Mem block and 16 replicated basic blocks with P.Arrays and memory. The P.Mem and its caches are located in the middle of the chip to reduce the load capacitance of broadcasting signals and to minimize signal skews. Broadcast data, address, and control signals are stretched into 8 100-bit busses. Each bus is about 1.5 cm long, with a line capacitance of about 3 pF including gate load capacitance [2]. The Rambus interface blocks are located on both sides of the P.Mem.

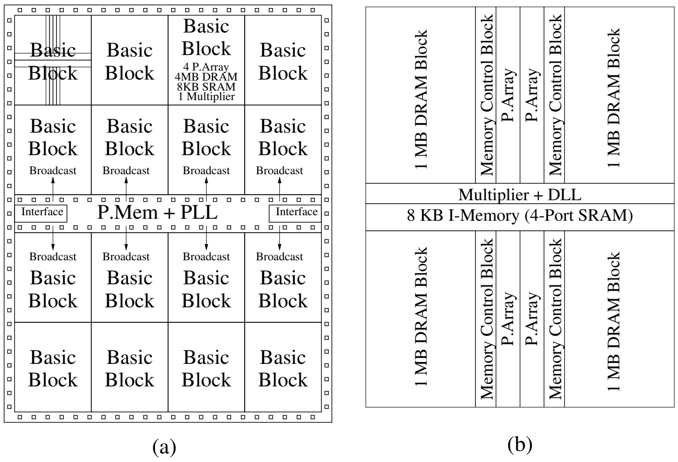

Figure 3: Layout of the *FlexRAM* chip (a), with a basic block blown up to show the detail (b).

Each basic block, blown up in Figure 3-(b), contains 4 P.Arrays, 4 1-Mbyte data memory blocks, one instruction memory, one multiplier, and one Delay-Locked Loop (DLL). The 4 P.Arrays are located in the center and share the 4-port SRAM instruction memory and the multiplier. Each 1-Mbyte memory block contains memory control logic and four 256-Kbyte blocks. Each of the latter is composed of 512 rows and 4 K columns. Such dimensions are chosen considering the ratio between the DRAM bit line (C_b) and cell (C_s) capacitance and the effect on chip size. Finally, as a common practice in the DRAM industry, we add 8 redundant rows and 32 redundant columns per 256-Kbyte block to replace defective memory cells.

Each 1-Mbyte memory block contains 3 2-Kbyte row buffers under random replacement [13]. In addition, for higher memory bandwidth, a multiple DRAM data line structure is used [16]. DRAM data lines run in parallel with DRAM bit lines and are located at every 32 columns, which results in 128 DRAM data lines per 1 Mbyte data memory block. Consequently, the maximum on-chip memory bandwidth becomes about 102 Gbyte/s with a 10 ns row buffer hit.

5.2 Clock Issues

An important concern for a DRAM process is density, while a major one for a processor logic process is speed. Consequently, a MLD process has to balance both demands. As a result, the logic in a MLD chip is likely to run slower than in a logic-only chip. In the 0.18 μm MLD technology that we use, we have assumed a logic frequency of 400 MHz [15].

For our chip to work, it is necessary to transmit a balanced clock signal to all P.Arrays. However, given the large chip size and the high clock frequency, this is challenging. We start by placing the central clock recovery circuit, a Phase-Locked Loop (PLL), at the center of the chip, in P.Mem. The PLL generates clocks to control the memory interface and the memory access operations with reference to an external clock. The PLL is also used to lower the clock frequency when the processors in the chip are idle and a power-save mode is entered. We also place one dedicated Delay-Locked Loop (DLL) in each basic block. The DLLs minimize clock skews on the chip and control the activity of the P.Arrays. The DLLs are in stand-by mode when the P.Arrays are not active. When the P.Arrays are activated, the DLLs generate local clocks for the P.Arrays and for the local memory accesses with reference to the PLL.

5.3 Area Estimation

The chip size at 0.18 μm is estimated by extrapolating existing data. A PowerPC 603 plus the caches as used in P.Mem has a size of about 80 mm^2 in 0.5 μm [34]. By considering appropriate shrink factors, P.Mem takes about 12 mm^2 in 0.18 μm. Since, for this technology, the DRAM cell size is about 0.34 μm^2 [18], the total area for the 512 Mbits of DRAM in our design, including row buffers, decoders, control logic, and interface circuits between the P.Array and the DRAM block in each bank is estimated to be 330 mm^2. The 4-port SRAM cell used for instruction memory is about 30 μm^2 [14]. Based on this, the total instruction memory plus its control logic is about 34 mm^2. A coarse placement of the elements of a P.Array takes about 1.5 mm^2. Since there are 64 P.Arrays, the total P.Array area is about 96 mm^2. Finally, we estimate the area of a 32-bit multiplier to be about 0.6 mm^2 [10]. Since we have 16 multipliers, the total area is about 10 mm^2.

Around the P.Mem, we have a Rambus interface, the network interface circuit for communication between P.Mems, and the DRAM refresh control circuits. One 16-bit Rambus interface block uses about 1.7 mm^2 [6]. Since *FlexRAM* needs a 32-bit interface, the total interface area is about 3.4 mm^2. Finally, we allocate 20 mm^2 for pads, network interface, and refresh circuits.

Overall, we get a chip of about 505 mm^2, which can be fabricated with advanced KrF technology [18]. Of this area, logic, including pads, takes 28%, while SRAM memory takes 7% and DRAM 65%.

5.4 Power Estimation

The expected operating voltage of the chip is 1.8 V [36]. We estimate the power consumed by one *FlexRAM* chip in two scenarios: when it is used as plain memory and when it is used as intelligent memory. In the first case, power is consumed when the P.Host accesses memory. The power is consumed in two main areas: the DRAM cells activated in the access, and the logic for clock generation and memory interface. Assuming that refreshing occurs 16,000 times per 128 ms [36], 32 K cells are activated during the access. Since the capacitance of a DRAM bit line is about 350 fF [18], the power consumed by the DRAM cells activated in an access to our 512 Mbit DRAM at 25 MHz is about 0.6 W. This includes driving the control circuitry. The power consumption in the clock generator and memory interface logic is about 0.1 W [12, 32]. Overall, 0.7 W are consumed.

When the *FlexRAM* chip is used as intelligent memory, the worst case occurs when the P.Mem and all the P.Arrays are active, and all the P.Arrays miss in the row buffers and access a new row from memory. The P.Mem plus its caches, clock generator, and memory interface consume about 4 W at 400 MHz [36]. Each P.Array is estimated to consume about 0.2 W. Since there are 64 P.Arrays, the total P.Array power consumption is about 12.8 W. This includes the instruction memories too. The multiplier and the local DLL in each 4 Mbyte block together consume about 0.1 W [16, 24]. Since there are 16 such modules, the total power consumption is 1.6 W. Finally, we need the power consumed by all 64 P.Arrays accessing a row in the DRAM. In this case, 2-Kbyte DRAM cells are activated in each 1-Mbyte DRAM block. The power consumed per block, including driving control circuitry, is about 0.27 W at 25 MHz. Since there are 64 blocks, the total consumption is about 17.3 W. Overall, adding all contributions, the power consumed in these conditions is nearly 36 W. This represents the worst possible case.

To manage this power, a pair of V_{dd} and V_{ss} pads are assigned for each P.Array and for each 1 Mbyte DRAM block. These power pads are located along the edge of the chip. All control pads are placed around the memory interface logic. The total pin count is expected to be 400-500. A Ball Grid Array (BGA) package [15] can be used so that this design meets pin and power requirements.

6 Evaluation

6.1 Methodology

To evaluate the proposed architecture, we compare a workstation with a *FlexRAM* memory system to a plain workstation. While we would like to perform a cost-performance comparison, the cost of hypothetical commodity PIM chips is too hard to quantify. Hopefully, the cost difference between an intelligent and a plain memory system will eventually be modest compared to the full cost of a powerful workstation or server. Consequently, we focus on performance only.

To evaluate performance, we use detailed software simulations. We model a workstation with a 800 MHz six-issue dynamic superscalar (P.Host). The architectural parameters are listed in Table 2. Since we can only simulate modest problem sizes, we use a relatively modest size for P.Host's L2 cache,

Applic.	What It Does	Problem Size	Global Miss Rates (%)				P.Array Executable Size (KB)
			Plain Workstn.		*FlexRAM* Workstn.		
			P.Host L1	P.Host L2	P.Mem	Row Buffers	
GTree	Tree Generation	5 MB database, 77.9 K records, 29 attributes/record.	10.0	8.3	6.0	8.2	3.7
DTree	Tree Deployment	1.5 MB database, 17.4 K records, 29 attributes/record.	1.4	0.3	-	22.7	3.3
BSOM	BSOM Neural Network	2 K inputs, 104 dimensions, 2 iter, 16-node network, 832 KB network.	1.6	0.4	-	0.6	2.0
BLAST	BLAST Protein Matching	12.3 K sequences, 4.1 MB total, 1 query of 187 bytes.	5.7	0.5	-	36.5	1.4
MME	MPEG-2 Motion Estimation	1 1024x256-pixel frame plus a reference frame. Total 512 KB.	0.0	0.0	-	2.8	1.7
TpcdQ3	TPC-D Query 3	10 MB database of which about 9.3 MB are accessed.	4.1	2.5	6.0	-	-

Table 1: Applications used. For the *FlexRAM* system, applications run on a single *FlexRAM* chip. The exception is *TpcdQ3*, which we run on 16 *FlexRAM* chips using P.Mems only. The P.Mem cache is practically unused in 4 applications.

namely 256 Kbytes, so that the problem size does not fit in it. Table 2 also lists the architectural parameters for the P.Mem, P.Arrays, and memory.

P.Host	P.Host L1 & L2	Bus & Memory
Freq: 800 MHz Issue Width: 6 Dyn Issue: Yes I-Window Size: 96 Ld/St Units: 2 Int Units: 6 FP Units: 4 Pending Ld/St: 8/8 *BR* Penalty: 4 cyc	L1 Size: 32 KB L1 *RT*: 2.5 ns L1 Assoc: 2 L1 Line: 64 B L2 Size: 256 KB L2 *RT*: 12.5 ns L2 Assoc: 4 L2 Line: 64 B	Bus: Split Trans Bus With: 16 B Bus Freq: 100 MHz Mem *RT*: 262.5 ns
P.Mem	**P.Mem L1**	**P.Array**
Freq: 400 MHz Issue Width: 2 Dyn Issue: No Ld/St Units: 2 Int Units: 2 FP Units: 2 Pending Ld/St: 8/8 *BR* Penalty: 2 cyc	L1 Size: 16 KB L1 *RT*: 2.5 ns L1 Assoc: 2 L1 Line: 32 B L2 Cache: No	Freq: 400 MHz Issue Width: 1 Dyn Issue: No Pending St: 1 Row Buffers: 3 *RB* Size: 2 KB *RB* Hit: 10 ns *RB* Miss: 20 ns *BR* Penalty: 2 cyc

Table 2: Parameters of the architecture simulated. In the table, *BR* stands for branch, *RT* for contention-free round-trip latency from the processor, and *RB* for row buffer.

The simulations are performed using a MINT-based [38] execution-driven simulation system that models out-of-order superscalar processors [22]. It includes a module that schedules RISC instructions for superscalar processors at run time. Different application threads in a multi-threaded application can have different issue width and issue policy (in-order or out-of-order). In our architecture, one thread is scheduled to run on a 6-issue dynamic superscalar (P.Host), one or more threads to run on 2-issue static superscalars (P.Mems), and 64 threads to run on single-issue static processors (P.Arrays). The instructions of the P.Array threads go through a translation step where they are translated from the MIPS ISA processed by MINT to the simpler P.Array ISA.

The applications that we run on the simulator were described in Section 3. To make a fair comparison, for each application, we prepare two versions, each one modestly optimized to run on different hardware: *FlexRAM* chips, or a plain workstation with a deep cache hierarchy. The *FlexRAM* version of each application is partitioned by hand, identifying and separating the work to be done by the P.Mem and the P.Arrays. In the future, we hope that compiler and programming language extensions can help in the partitioning. The *FlexRAM* version of each application runs on a single *FlexRAM* chip. The exception is TPC-D, which we run on 16 *FlexRAM* chips using P.Mems only. Table 1 lists, for each application, the name, what it does, the problem size, the miss rates, and the size of the P.Array executable.

We now explore how the *FlexRAM* memory system is used and the speedups that it delivers.

6.2 How the Intelligent Memory is Used

We break down the normalized execution time of the P.Arrays and P.Mems in Figures 4-(a) and 4-(b) respectively. As shown in Figure 4-(a), a P.Array can be executing useful instructions (*Busy*), waiting for memory (*Memory*), waiting for other P.Arrays (*PA/PA Wait*), waiting for the P.Mem (*PA/PM Wait*), or stalled in pipeline hazards (*Hazards*). In turn, as Figure 4-(b) shows, a P.Mem can be executing useful instructions (*Busy*), waiting for memory (*Memory*), waiting for P.Arrays, other P.Mems, or the P.Host (*Wait*), or stalled in pipeline hazards (*Hazards*).

According to Figure 4-(a), it is common for P.Arrays to be *Busy* around 40-60% of the time. The rest of the time is often spent waiting for memory. The fraction of *Memory* time, however, is not correlated with the miss rate of the row buffers (last to one column of Table 1). The reasons are that different applications access memory with different frequencies and that the difference in latency between a row buffer hit and a miss is small.

In some applications, P.Arrays waste time waiting for the P.Mem or for other P.Arrays. In *GTree*, P.Arrays are waiting on the P.Mem for about 45% of the time. The reason is that *GTree* has a large serial section, where only the P.Mem is busy (Figure 4-(b)). This results in unutilized P.Arrays. Fortunately, in the other applications, there is no major serial section and the P.Mem activity is largely limited to a few reductions and broadcasts. As a result, Figure 4-(b) shows that the P.Mem is largely idle. *TpcdQ3* is a special case in that P.Arrays are unused. *TpcdQ3* runs on 16 *FlexRAM* chips and only uses P.Mems. It achieves a good 40% *Busy* time. *Memory* time is significant in *TpcdQ3* because the small P.Mem caches suffer frequent misses (Table 1) when running the parallelized *TpcdQ3*.

6.3 Speedup Over Plain Memory

To understand any speedup delivered by *FlexRAM* over plain memory, we first examine the performance of the applications on the plain workstation, and then analyze the *FlexRAM*

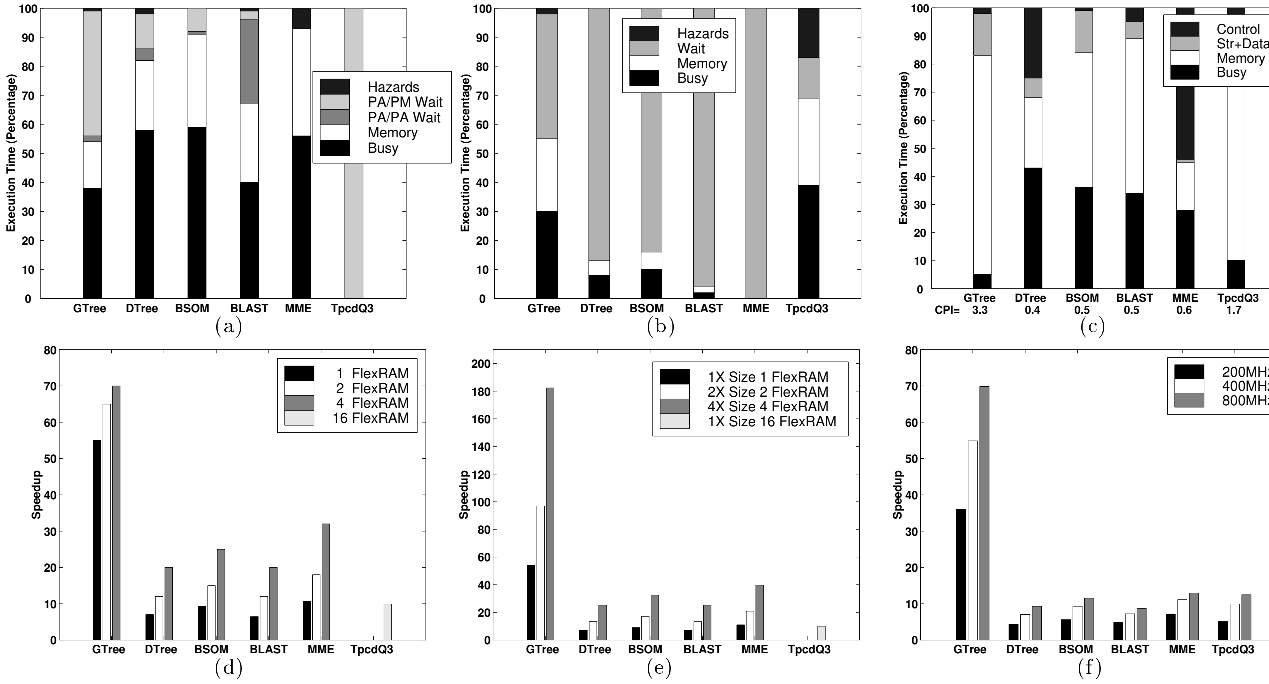

Figure 4: Evaluating *FlexRAM*. Charts (a), (b), and (c) break down the execution time of P.Arrays, P.Mems, and (in the plain workstation architecture) P.Host respectively. Charts (d), (e), and (f) show the speedups of the *FlexRAM*-based system over the plain system for different conditions: different numbers of *FlexRAM* chips for a constant problem size, different numbers of *FlexRAM* chips for a proportionally-scaled problem size, and different MLD logic frequencies respectively. *TpcdQ3* does not use P.Arrays.

speedups.

6.3.1 Plain Workstation Performance

To be fair in estimating speedups, we have tried to run the applications efficiently on the plain workstation. For example, the P.Host's L1 and L2 miss rates are small for most of the applications (Table 1). However, it is hard to keep a six-issue processor highly utilized while running memory-intensive applications, given the large memory latencies of modern systems. Figure 4-(c) breaks down the execution time of the applications into instruction execution (*Busy*), waiting for memory (*Memory*), structural and data pipeline hazards (*Str+Data*), and control hazards (*Control*).

From the often modest fraction of *Busy* time, we can compute the CPIs, which are shown at the base of the bars. The CPIs range from 3.3 to 0.4. The large *Memory* time in *GTree*, *BLAST*, and *TpcdQ3* shows that these three applications are largely memory bound. They are also the ones with the highest miss rates in Table 1. *GTree* follows long lists of records with poor locality such that, when a record is accessed for a second time, it has already been displaced from the cache. *TpcdQ3* has a large memory time because data is not reused. While the application has spatial locality, cache lines have a modest size (64 bytes). In addition, because the database is relatively small, there are start up misses. *BLAST* accesses a very large hash table with poor locality.

6.3.2 Speedups

The *FlexRAM* speedups are shown in Figure 4-(d) as the leftmost bars of each application (1 *FlexRAM* for all applications except for *TpcdQ3*; 16 *FlexRAM* for *TpcdQ3*). We can see that, with one *FlexRAM* chip, we get speedups of around 7-11. The exception is *GTree* which, due to its poor performance in the plain workstation, has a speedup of about 50. *TpcdQ3*, which runs with 16 *FlexRAM* chips, has a speedup of about 10. Overall, these are good speedups.

It is interesting to see the speedups as we add more *FlexRAM* chips. We consider two scenarios: keeping the problem size constant (Figure 4-(d)) and increasing it proportionally with the number of chips (Figure 4-(e)). The legend in Figure 4-(d) refers to the number of *FlexRAM* chips used; the legend in Figure 4-(e) refers to the total size of the problem running, and the number of chips used.

Increasing the number of chips for constant problem size (Figure 4-(d)) delivers good speedups. In most cases, 4 chips run about 3 times faster than 1 chip, delivering speedups of 20-30 over the plain system. The reason is that these applications are fairly parallel and need little communication between chips. The exception is *GTree*, which has a serial section that can only run on a single P.Mem. As a result, the speedup for *GTree* increases only slowly.

If we increase the problem size as we increase the number of chips (Figure 4-(e)), we see even better speedups. Many of these applications are mostly data parallel, so speedups scale well. Furthermore, *GTree* now delivers good speedups. The reason is that the P.Mem section takes the same amount of time irrespective of the number of records; it depends on the attributes per record. The plain workstation, however, runs twice slower with twice more records. Overall, we can see that 4 *FlexRAM* chips often allow the workstation to run 25-40 times faster than without intelligent memory. For *GTree*, the speedup is 180.

Finally, we examine the effect of the logic speed in the MLD process. We assume that the speed of the logic in the *FlexRAM* chip can be changed from 400 MHz to 200 or 800 MHz. The DRAM size and speed are unchanged. Furthermore, we unrealistically assume that the speed of the SRAM in the *FlexRAM* (instruction memory for the P.Arrays and cache for P.Mem) also changes with the logic frequency, so that the access time is always 1 cycle. Under these conditions, Figure 4-(f) shows the speedups of the intelligent workstation over the plain one for *FlexRAM* logic speed of 200, 400, and 800 MHz. One *FlexRAM* chip is used for all applications except for *TpcdQ3*, where 16 *FlexRAM* chips are used. The figure shows that higher speedups are delivered as we go from 400 to 800 MHz. However, the increases are smaller than the ones delivered as we went from 200 to 400 MHz. This is because logic speed is only one of several factors that contribute to *FlexRAM* speedups. Overall, it appears that 400 MHz logic in MLD is a good design point for systems with 800 MHz commodity processors.

6.4 Cost-Effectiveness of *FlexRAM*

It is hard to justify a chip like *FlexRAM* given today's low DRAM prices. However, we believe that, with the fast growth of chip density, integrating more logic with memory is the only way to alleviate the memory bottleneck, and possibly the best way to exploit the huge number of transistors available. MLD technology will continue to show breakthroughs, which will make fabricating a *FlexRAM*-like chip only moderately more expensive than an advanced DRAM part. If we consider the low fraction of a system's cost that memory represents, and the good speedups that we get for key applications in the server domain, the near-future cost-effectiveness of something like *FlexRAM* looks promising.

7 Related Work

PIM or IRAM architectures are an active research field. Among the most prominent work, we have Notre Dame's Execube [19] and Petaflop [20] systems, UC-Davis' Active Pages [26], ISI-USC DIVA system [9], UC-Berkeley's IRAM [27] and ISTORE [28], MIT's Imagine [33] and Raw [39] work, and the vast body of work presented at the First Workshop on Mixing Logic and DRAM [29]. Some of these projects, including Imagine and Raw, use SRAM as the large on-chip storage. Roughly speaking, the work can be classified based on the role of the PIM chip: main processor (or processors), special purpose (or co-processor), and memory system. Our work falls in the latter, together with the Active Pages and the DIVA project. The Active Pages work is different in that one of its major aspects is including reconfigurable logic in the PIM chip. Such a technology is a few years off in the future. Our purpose is to examine how to best use today's MLD technology. Furthermore, a crucial part of our work is to find and understand real applications for PIM. We have used them to perform a detailed design and layout of the chip. To our knowledge, there is no published work on DIVA, but some of the work appears similar. Finally, a survey of the issues in embedded DRAMs can be found in [30].

8 Conclusions and Future Work

This paper has addressed how to best use the current state-of-the-art MLD technology for general-purpose computers. We wanted a general-purpose system that supported a wide range of applications. The paper has described and justified its architecture and proposed a chip layout. A simulation-based evaluation showed that 4 intelligent memory chips often allow a workstation to run 25-40 times faster.

Many issues are open for research. Perhaps the most challenging one is to provide easy-to-use and efficient programming support for the architecture presented. Another issue is to make the system more usable by enhancing and giving flexibility to the memory management and virtual memory systems. Finally, another issue is to examine in detail the I/O subsystem support required for this architecture. Since data is consumed efficiently, it must be loaded into memory efficiently too.

Acknowledgments

Some of the ideas in this paper owe to discussions with A. Chien, M. Horowitz, D. Padua, and D. Reed. We also thank our colleagues at IBM, especially K. Ekanadham, R. Isaac, B. Lim, and R. Miller. Finally, we thank the referees and the graduate students in the I-ACOMA group. J. Torrellas is supported in part by a NSF Young Investigator Award.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. In *Journal of Molecular Biology*, pages 403–410, 1990.

[2] J. Alvarez et al. A 450MHz PowerPC Microprocessor with Enhanced Instruction Set and Copper Interconnect. In *ISSCC Digest of Technical Papers*, pages 96–97, February 1999.

[3] M. Berry and G. Linoff. *Data Mining Techniques*. John Wiley & Sons, Inc., New York, NY, 1997.

[4] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of Existing Architectures in IRAM Systems. In *First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[5] A. Brown, D. Chian, N. Mehta, Y. Papaefstathiou, J. Simer, T. Blackwell, M. Smith, and W. Yang. Using MML to Simulate Dual-Ported SRAMs: Parallel Routing Lookups in an ATM Switch Controller. In *First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[6] R. Crisp. Direct Rambus Technology: the New Main Memory Standard. In *IEEE Micro*, pages 18–28, November 1997.

[7] J. A. Gasbarro. The Rambus Memory System. In *International Workshop on Memory Technology, Design and Testing*, pages 94–96, 1995.

[8] G. Gerosa et al. A 2.2 W, 80 MHz Superscalar RISC Microprocessor. In *IEEE Journal of Solid-State Circuits*, December 1994.

[9] J. Granacki et al. Data Intensive Architecture: DIVA. http://www.isi.edu/asd/diva/, 1998.

[10] Y. Hagihara et al. A 2.7ns 0.25um CMOS 54x54b Multiplier. In *ISSCC Digest of Technical Papers*, pages 296–297, February 1998.

[11] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, January 1993.

[12] M. Horowitz et al. PLL Design for a 500MB/s Interface. In *ISSCC Digest of Technical Papers*, pages 160–161, February 1993.

[13] W. Huang. Exploiting Application Parallelism Using Advanced Intelligent Memory - The FlexRAM Approach. MS Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[14] IBM Microelectronics. Databook for 5S Technology. 1998.

[15] IBM Microelectronics. Blue Logic SA-27E ASIC. In News and Ideas of IBM Microelectronics, http://www.chips.ibm.com/news/1999/sa27e/, February 1999.

[16] K. Itoh et al. Limitations and Challenges of Multigigabit DRAM Chip Design. In *IEEE Journal of Solid-State Circuits*, pages 623–634, May 1997.

[17] S. Kaxiras, R. Sugumar, and J. Schwarzmeier. Distributed Vector Architecture: Beyond a Single Vector IRAM. In *First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[18] K. Kim et al. Highly Manufacturable 1 Gb SDRAM. In *Symposium on VLSI Technology Digest of Technical Papers*, pages 9–10, June 1997.

[19] P. Kogge. The EXECUBE Approach to Massively Parallel Processing. In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.

[20] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, pages 88–97, October 1996.

[21] C.E. Kozyrakis et al. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, pages 75–78, September 1997.

[22] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[23] R. Lawrence, G. Almasi, and H. Rushmeier. A Scalable Parallel Algorithm for Self-Organizing Maps with Applications to Sparse Data Mining Problems. Technical report, International Business Machines, January 1998.

[24] T. Lee et al. A 2.5V DLL for an 18Mb, 500MB/s DRAM. In *ISSCC Digest of Technical Papers*, pages 300–301, February 1994.

[25] Mitsubishi Corporation. http://www.mitsubishichips.com/eram/eram.htm. 1998.

[26] M. Oskin, F. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, June 1998.

[27] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Tomas, and K. Yelick. A Case for Intelligent DRAM. In *IEEE Micro*, pages 33–44, March/April 1997.

[28] D. Patterson et al. ISTORE: Intelligent Store. http://iram.cs.berkeley.edu/ istore/index.html, 1998.

[29] D. Patterson and M. Smith. First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. June 1997.

[30] S. Przybylski. Embedded DRAMs: Today and Toward System-Level Integration. Tutorial with the Annual International Symposium on Computer Architecture, May 1997.

[31] J. R. Quinlan. *C4.5 - Programs for Machine Learning*. Morgan-Kaufmann Publishers, Inc., San Francisco, CA, 1993.

[32] Rambus Inc. 16/18Mbit (2Mx8/9) and 64/72Mbit (8Mx8/9) Concurrent RDRAM Data Sheet, DL0029-05. In *Rambus Inc.*, January 1997.

[33] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *31st International Symposium on Microarchitecture*, November 1998.

[34] H. Sanchez et al. A 200MHz 2.5V 4W Superscalar RISC Microprocessor. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 218–219, February 1996.

[35] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 90–101, May 1996.

[36] Semiconductor Industry Association. The National Technology Roadmap for Semiconductors, 1998. http://notes.sematech.org/ ntrs/PublNTRS.nsf.

[37] Transaction Processing Performance Council. *TPC Benchmark D (Decision Support) Standard Specification Revision 1.1*, December 1995.

[38] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *MASCOTS'94*, pages 201–207, January 1994.

[39] E. Waingold et al. Baring It All to Software: Raw Machines. In *IEEE Computer*, pages 86–93, September 1997.

Appendix A: Applications

Data Mining

The nodes of a decision tree are predicates on one or more attributes of the record in a database. The outcome of the predicate on a record determines the subsequent branch of the tree to take while traversing from the root. The record finally reaches a leaf of the tree which classifies the record. For example, given records on individuals, a decision tree can be used to find target individuals for life insurance promotion. The nodes of the tree are questions about the family and income situation of the individual.

Tree Generation

To generate the tree, the algorithm has to determine which record attributes to query in the nodes, in what order and, for a given attribute, what values to use to split the records. The goal is to produce the most compact tree with the most confident answers [31].

We start from a preclassified set of the records. To generate the root node of the tree, we inspect each attribute and count, for each attribute value, the number of records that have each possible outcome. Based on this, we choose the attribute and attribute value that produces the best split, that is, resulting in largest information gain [31]. For each branch, we now repeat the process with only the records that contain the corresponding attribute value. This will give us the next pair of attribute and attribute value. This process continues until all the records in the branch have the same outcome or there are too few records to give a meaningful split.

This algorithm can be mapped to one *FlexRAM* chip as follows. The database of records is divided among the P.Arrays. The P.Mem controls the tree generation and makes all decisions. Initially, each P.Array counts, for the records it owns, the outcomes for each attribute and attribute value. Then, all P.Arrays synchronize. The P.Mem tabulates the results and makes the decision about the attribute and value on which to split. It then transmits this information to the P.Arrays, together with an indication of what branch to examine next. This cycle is repeated again and again except that each P.Array only works on the subset of its records that are being considered in this subtree.

In this application, P.Arrays have large communication-free sections. However, at regular intervals, they synchronize in barriers and communicate. For the communication, they can use nearest neighbor as they accumulate results for the P.Mem. Computation is all integer addition. As long as many records fit in each P.Array's memory, the communication time is small. So is the record load time, especially if the database is used to generate several trees. Finally, if the database does not fit in one chip, several chips can work in parallel. In each step, a master P.Mem can accumulate the partial data accumulated by each of the other P.Mems. This introduces more communication but, overall, communication is likely to remain modest.

Tree Deployment

This algorithm applies a tree to every single record of a large database to classify them [31]. In our architecture, we assign a portion of the database records to each P.Array and replicate the tree in all P.Arrays. Each P.Array sequentially processes its records. For each record, the tree is traversed, checking the conditions in the nodes, and reaching the outcome in the correct leaf. Finally, all P.Arrays synchronize in a barrier and the P.Mem (or P.Mems in case we are using several chips) accumulates the result.

This application is well-suited for *FlexRAM*. P.Arrays have practically no communication beyond two barriers. Their operations are mainly comparisons with scattered integer arithmetic. One concern is that the ratio of computation to I/O may be low. While the amount of computation per record depends on the input tree, the examples that we tried have about 40 P.Array instructions between two consecutive P.Array loads to the database data. If we add to this time any P.Array load imbalance, synchronization time, and non-overlapped P.Mem time, we see that the data processing time is larger than the data loading time with fast page-mode DMA. If, in addition, the database fits in memory, then the application reuses the data across queries and the initial I/O time is negligible.

Neural Networks

Neural networks also classify data. The Self-Organizing Map (SOM) algorithm takes as input a set of data points belonging to the N-dimensional space. It clusters them into groups of similar-looking points laid out in a 2D map [23]. To identify an arbitrary number

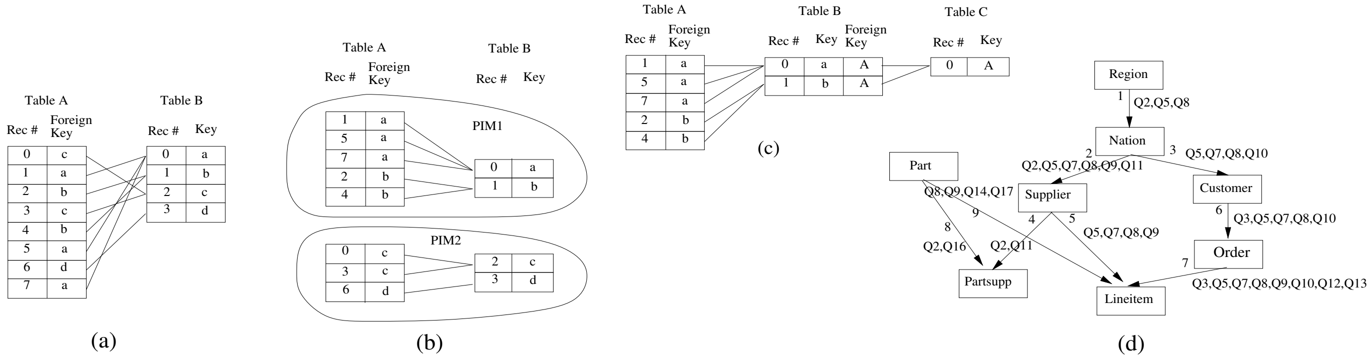

Figure 5: Laying out tables to minimize inter *FlexRAM* chip communication in joins.

of K groups, it proceeds as follows. Each of the first K inputs goes to a different group. Their coordinates become the initial weight vector of each group. Then, for each of the other inputs, we do as follows. We compute the euclidian distance between the input vector and each weight vector. The smallest distance indicates the group where the input belongs to. At this point, the weight of such a group is updated to be a weighted average of its initial weight and the new input. Intuitively, we have moved the center of gravity of the group to reflect the influence of this input. The other groups' weights are also updated to a smaller degree controlled by a correlation matrix. To force convergence, the whole process is repeated several times.

In practice, in the BSOM (Batch SOM) algorithm, no center of gravity is changed until all inputs have been processed. At that point, all the changes are performed. At the end, a clear pattern of K centers of gravity emerges.

In a *FlexRAM* chip, each P.Array starts with a copy of the initial K weights. Each P.Array processes a portion of the inputs, updates its local weights and synchronizes. Then, P.Mem reads all the local weights, combines them and broadcasts the result to all P.Arrays.

This is a compute-intensive algorithm with little P.Array communication. Theoretically, the algorithm uses floating point. However, given that floating point is so area-expensive, we can use integer units to do the integer arithmetic without losing too much precision. Multiplication is used to compute distances and, therefore, must be supported efficiently.

Computational Biology: Protein Matching

We look for sections of a query string that are similar to sections of a protein database. Since there may be mutations, similarity is measured with a function that takes into account the amino acids found in the two neighborhoods. The original algorithm, called BLAST [1], sequentially takes each amino acid in the proteins of the database and compares it to all the amino acids in the query string. The comparison is done in groups of 4 consecutive amino acids at a time. Each group is decomposed into 50 possible permutations to allow for mutations. If two 4-amino acid sequences in the mutated query and database are the same, a match is detected and it now needs to be extended. Extension implies comparing the next amino acids in sequence in both query and database. Depending on how similar they are, a similarity metric is increased or decreased. Such extension proceeds to the left and right of the original match until the similarity metric reaches a certain positive or negative threshold.

In our architecture, we load the protein database in the *FlexRAM* chips. Each P.Array is given a piece of the database. The query string is replicated in all P.Arrays. Each P.Array tries to match it against its portion of the database. For each match it finds, it extends it. Due to the small size of a protein, the extension will not cross *FlexRAM* chips but may cross P.Array memories. If the extension reaches a P.Array boundary, the P.Array can store the information gathered so far in the memory of the neighbor P.Array. After processing their protein chunks, all P.Arrays synchronize in a barrier. Then, each P.Array proceeds to extend the matches initiated by its two neighbors. These processing and synchronization steps interleave until all extensions conclude.

This application is highly parallel, fairly compute intensive, has a significant grain size and, at most, needs nearest-neighbor communication between P.Arrays. Computation is all integer. The I/O time to load the database is much smaller than the compute time and, if the database fits in memory and is reused across queries, totally negligible.

Decision Support Systems: TPC-D

We code TPC-D to use only P.Mems and intra-operation parallelism [11], whereby each processor operates on a section of the tables. We want to minimize communication between chips. We show how to do it for joins. For joins, communication is reduced dramatically if we optimize data layout. Each database table has its key, which takes a different value in each record. In all TPC-D joins, the key in a table is joined to a foreign key in another table. A foreign key usually takes the same value in several records. Figure 5-(a) shows a join. Thanks to the uniqueness of the keys, given two tables to be joined, we can always partition them between several PIM chips for no communication (Figure 5-(b)). If we want a load-balanced system, we may have to replicate some entries (for example, move record #4 of Table A from PIM1 to PIM2 and replicate record #1 of Table B in both PIMs). Finally, since a query has several joins, we repeat the partitioning across joins. For example, Figure 5-(c) shows the data in PIM1 after laying out a query with (A *join* B) *join* C.

We can lay out the TPC-D tables in a *FlexRAM* ensemble like this as long as different queries do not put conflicting layout demands on the same table. To see the problem, consider the TPC-D join graph (Figure 5-(d)). The graph shows all the TPC-D tables, how they are joined (an arrow between two tables indicates a join, with the source of the arrow indicating the table with the key), and what queries execute them. If we remove table *Part* and edges 5, 8, and 9 from the graph, we get a tree. For a tree, we can lay out all tables so that we need no inter-chip communication in any TPC-D join. To see why, start from the root and, in a depth-first manner lay out records from different tables in the *FlexRAM* chip. For example, lay out the first *Region* record, then find all its matches in *Nation*. For each of those, lay it out and find all its matches in *Supplier*. The process proceeds until the first chip is full; then move on to the next chip. Although some record replication may be necessary, none of the joins will require inter-chip communication with this layout. Finally, we distribute the *Part* records in chunked round-robin across chips. Joins 5, 8, and 9 will require inter-P.Mem communication. This communication can be supported well if P.Mems share all memory.

The initial and only loading of the data in the *FlexRAM* chips can be efficiently done with the help of hash tables. Finally, if an update query tries to add an entry, we need to store it in the right chip. However, updates are rare.

Multimedia: MPEG-2 Motion Estimation

Given an image, we take each 8x8-pixel block and compare it against many 8x8-pixel blocks in a reference image. The latter blocks include the one with the same coordinates as the original one, and its surrounding blocks. Block comparison involves integer arithmetic. In *FlexRAM*, we partition each of two images in the same way between P.Arrays. If the P.Arrays are connected in a 1D manner, each P.Array gets a set of contiguous rows. Each P.Array operates on its own section of the image. Communication is highly local: P.Arrays access their own memory and read the ones in their nearest neighbors.