EXPLOITING APPLICATION PARALLELISM USING ADVANCED INTELLIGENT MEMORY - THE FLEXRAM APPROACH

BY

WEI HUANG

B.Eng., Tsinghua University, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

ABSTRACT

The state-of-the-art microprocessor employs tens of millions of transistors on a single chip, most of them are used to tackle the problem of slow memory. While these processors demonstrate great benchmark ratings, they do not necessarily work well on a lot of real world applications. On the other hand, memory components offer huge numbers of transistors that could be utilized to do significant work thanks to major advances in Merged Logic DRAM technology. In this thesis a novel architecture is proposed to bridge the memory/processor speed gap in a much more cost effective way than the traditional central processor. Also parallelism is exploited using simple processor arrays thus surmounting the limit of ILP. The combined effect has very promising results. Assuming conservative parameters for the proposed system while using aggressive parameters for a traditional reference system, we get an average speedup of 7.8 and best case of 36 for five benchmarks only with a single memory chip.

To My Parents, for their endless love and support.

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor Josep Torrellas for his guidance and support over the past few years, which made the completion of this thesis possible.

Secondly, my gratitude goes to all the IACOMA group members, current and past, Sujoy Basu, Qiang Cao, Marcelo Cintra, Zhenzhou Ge, Yi Kang, Diana Keen, David Koufaty, Venkata Krishnan, Vinh Vi Lam, Jose Martinez, Anthony-Trung Nguyen, Jose Renau, Yan Solihin, Pedro Trancoso, Liuxi Yang, Seung-Moon Yoo, and Ye Zhang. Their precious help is what one needs in graduate study. Working with these smart fellows is a pleasure for me.

I would also like to thank all my friends in Champaign-Urbana who made my otherwise monotonous life colorful. Among others I wish to thank Yuan Lin, Jun Wang, Yingli Ma. Special thanks to my officemate Yi Kang, discussion with such a knowledgeable person is inspiring, also, party at his house is always the best way to spend a weekend.

In addition I want to thank my friend Zhihan Xu who gave me more nicknames than I can remember. Without his reminder, I would have missed the opportunity to meet Professor Torrellas and would not have been here to do all the interesting work. I thank Fan Jing for her constant supervision on the progress of my thesis without which I would procrastinate forever. I also thank Huhu, a girl who is often confused, who sees the world through different eyes, talking with her helped to solve a lot of problems.

Last and most importantly, I wish to thank my parents. I owe them everything, and it is beyond my power of word to express my love and thankfulness towards them.

TABLE OF CONTENTS

Chapter

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Ever since the first electronic general-purpose computer ENIAC was built in 1946, technology in computer hardware has been changing steadily. Technologies, especially VLSI technology have such a great impact on computer industry that nearly every architectural innovation in computer are not only enabled by changes in this technology but also *initiated* by them.

On the other hand, computer architecture serves as a way to improve system performance of computer applications, which is the ultimate goal of the computer user. Thus a computer architecture design should be driven by the demands from application domain.

Computers brought revolution to science, but they are not limited to this domain. We are seeing a wider and wider usage of computers in everyday life, – business, entertainment and so on. Great improvements in computer performance for these domains could bring revolutionary changes to the whole society. For example, today people have already changed the way to run supermarket with the help of data mining. Biologists, among others, also benefit from the unprecedented computing power that enables them to advance their research.

As the computing power becomes more widely available, new computer applications evolve. Unlike in the old domain of scientific computing where programs tend to be compute-intensive floating-point applications, many new computer applications tend to be memory-intensive integer programs. Caches are used to bridge the speed gap of fast processor and slow memory, but they are not always effective. Modern processors sometimes dedicate more than 80% of the chip area to caches, we still see programs with high miss rates. Naturally, people start to think of putting logic inside the memory [11] to exploit the low latency and high bandwidth of memory macro cells. This thesis summarizes some of the results of an ongoing research project that quantitatively evaluates the design trade-offs in implementing advanced intelligent memory. The design of the architecture is driven by these applications that we envisioned important and not well addressed by current architectures. The architecture embeds two levels of processors in the memory chip to exploit application's parallelism while at the same time keep the hardware overhead to a minimum. Simulation of the proposed architecture shows an average speedup of 6.3 and, in one application, over 30.

The rest of the thesis gives an overview of the application domains and specific benchmarks we used (chapter 2), a detailed description and some justification of the architecture (chapter 3), a summary of the simulation results of the benchmarks (chapter 4) and related works (chapter 5). In the end I conclude in chapter 6.

Chapter 2

Applications

Computers are no longer used exclusively for scientific computing, where floating-point compute-intensive programs dominate Many other applications are becoming more and more important. Some of them do not a have very complicated algorithm, but do have very large data sets to process. We choose some of the applications from two fields, namely *Data Mining* and *Computational Biology*, which we believe to be very important and not studied much.

2.1 Data Mining

Nowadays, nearly every business uses computers to track its data. Although much of this data is collected to run the business, it has the potential to be exploited to *manage* the business (to discover trends, to find new opportunities etc). Thus *Data Mining* is defined by some [7] as:

the process of extracting previously **unknown**, **valid**, *and* **action-able** *information from large databases and then using the information to make crucial* **business decisions**.

Everyday, a large amount of data is produced and stored in corporate data warehouse. To get strategic information out of these huge databases people have to resort to various techniques in Data Mining[2]. Also known as *Knowledge Discovery in Databases*, Data Mining uses the research in the 1980's and early 1990's to extract information automatically from large databases, as opposed to an earlier time when data analysts were hired to use verification based tools try to validate their intuition on the relationship of data. For example, a data analyst would guess: "most people who bought bread would also buy cheese" and set out to validate that using tools. Data mining tools, on the other hand could report automatically that, for instance, 80% of the sales involving bread also involve cheese. In reality, data mining tools can induct much more complicated rules, which are hard to develop by a human analyst.

2.1.1 Different Classes of Data Mining Problems

From strategic information aid in a department store to insurance fraud detection, data mining can be applied to various domains yet they all have similar underlying techniques. Numerous ways exist to categorize these approaches, here is one of them:

- *association:* The above mentioned example is a typical one for association. In this approach, items kept in database are associated with each other in the form of *antecedent* (e.g. bread), *consequent* (e.g. cheese) and *confidence* (e.g. 80%). Each element of the 3-tuple can be fixed in a trial to find the valid rules.

- *sequence:* While association captures the static relationship among transactions, sequence based analysis deals with data with temporal connections. For example, it will give more insight when stock market transactions are analyzed on the time scale to find chain effects.

- *classification:* Classification is considered by many to be the most commonly applied data mining technique. It involves finding rules that partition the given data into disjoint groups. Usually, people use pre-classified data to do training and then use the trained entity (a decision tree or neural network) to further classify data with unknown classes.

- *clustering/segmentation:* Slightly different from *classification* (some consider clustering as a kind of classification), clustering doesn't have a picture of all the classes a priori, the classes emerge after clustering. Clustering is used, for example, to group similar customers together so that marketing messages can be targeted at homogeneous groups to increase effectiveness.

We choose two programs, one doing classification the other doing clustering, as our benchmark to study the applications' need from architecture.

2.1.2 The BSOM Algorithm

The clustering operation can be implemented in different ways. One of them is by employing a Kohonen feature map neural network [22] and the model is known as *Self Organizing Map.* [25]

When doing clustering, we want similar input record to be grouped together, intuitively, these record are close to each other in terms of Euclidean distance in the attribute space. Thus when an input is presented to the neural network, all the node in the network compete with each other to be the "winner". The winner is the closest node to the input vector. The weights of the winner is adjusted (moved to the direction of the input vector) by an predetermined learning rate factor. The factor is chosen to make the repeated process converge.

The algorithm is applied to the database without a training process, this is also referred to as unsupervised learning. The database is swept through for many times to adjust the weights of the neural network nodes. Each run is called an *epoch.* The number of nodes determines the number of resulting clusters. A weight vector is associated with each node $w_k \in R^n$ (n is the number of field in the record). This vector is used to compute the Euclidean distance when processing a new input record:

$$d_k(t) = \|x(t) - w_k(t)\|^2$$

(because the self adjusting feature these value are changing with processing time t) The winning node is

$$d_c(t) = \min_{0 \le kl < n} d_k(t)$$

After finding the winning node, the weight vectors are updated:

$$w_k(t+1) = w_k(t) + \alpha(t)h_{ck}(t)[x(t) - w_k(t)]$$

$$h_{ck}(t) = e^{-\frac{\|r_k - r_c\|^2}{\sigma(t)^2}}$$

$\alpha(t)$ is the monotonically decreasing learning factor, $\sigma(t)$ is another monotonically decreasing factor: the width function. It goes from the effective width of the lattice(the dimension of the neural network) in the first *epoch* to the width of a cell (i.e. 1) in the last *epoch* and r_k, r_c are the 2D coordinate of node k, and the winning node c. We can see that every node is actually affected, with the winning node's change and that of those near neighbors being much larger than that of the far away nodes.

The first N input vectors are used to initialize the N weight vectors. This flavor of self-organizing map is called online SOM, since at time $t+1$ when the next input is processed the weight vectors are adjusted by input processed at time t. This creates an unnecessary dependency and thus serialization of otherwise data parallel algorithm. In another flavor of the algorithm, (batch SOM) the weight vectors are not updated until the end of the epoch:

$$w_k(t_f) = \frac{\sum_{t'=t_0}^{t'=t_f} \bar{h}_{ck}(t')x(t')}{\sum_{t'=t_0}^{t'=t_f} \bar{h}_{ck}(t')}$$

The new distance is computed by: $\bar{d}_k(t) = \|x(t) - w_k(t_0)\|^2$. t_0 and t_f are the time for processing the first and last input record in an epoch respectively. So the weights are updated at the end of each epoch, while during the epoch the weights used to compute distance is kept constant. In this manner, multiple processors can be employed to process portion of the input record independently, and communicate only at the end of each epoch. Unlike supervised learning neural network, BSOM (batch SOM) algorithm is to be deployed on the real data sets directly whose size should be huge and increasing, this renders the communication relatively inexpensive. We will see later in Chapter 4 that even in our simulated program using reduced data set the communication is not a significant part of the total execution time.

2.1.3 Decision Tree

A different approach to classify the data is by using a decision tree like the one shown in figure 2.1. A customer record is checked when traversing the decision tree. Depending upon the result of test on each node different paths are selected. This process is repeated until a leaf node is reached where the class of the record is decided.

How is the decision tree constructed? Unlike in the self organized map a relatively small number of training cases with predetermined classes are used to build the tree. By using divide and conquer, the current set can be divided into subsets, the process is repeated until all the members belong to the same class. In a pathological extreme case, each node consists only one training case which surely belongs to one single class. Such a decision tree would be too large to be practical. Thus it boils down to the question of how to build a tree with minimum nodes that can correctly classify the training set. This problem, not surprisingly, is NP-complete, and naturally an approximation algorithm is used. C4.5 is a classical solution that applies a non-backtracking greedy algorithms. The heuristic for selecting the current test to break the set into subsets is to maximize information gain:

$$Gain(X) = Entropy(T) - \sum_{i=1}^{n} \frac{|T_i|}{|T|} \times Entropy(T_i)$$

where T is the original set, divided by test X into n subsets T_1 through T_n, and $Entropy$ is based on the classes C_i in the set T:

$$Entropy(T) = -\log_2 (\frac{|C_j|}{|T|})$$

$$\bigcup_j C_j = T$$

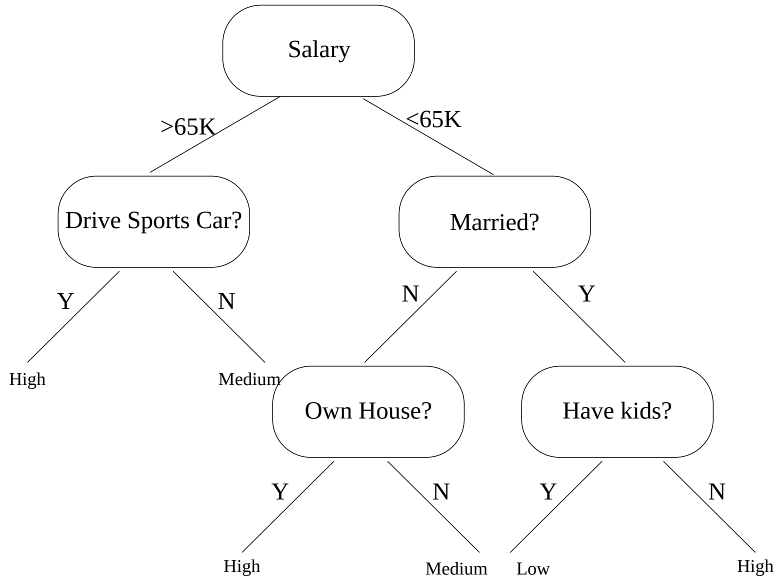


Figure 2.1: Decision Tree: Probability of Buying Life Insurance

$$C_i \cap C_j = \phi, (i \neq j)$$

In reality the program uses adjusted *gain ratio* to reduce the bias toward tests with large numbers of outcome.

$$GainRatio(X) = \frac{gain(X)}{-\sum_{i=1}^{n} \frac{|T_i|}{|T|} \times \log_2(\frac{|T_i|}{|T|})}$$

The algorithm of tree generation is simply to pick the test that yields the maximum *gain ratio* for the current training set and divided the current set according to the test result and repeat the process to each subset until there is only one single class inside.

2.2 Computational Biology

One very important aspect of biology is to understand better the job of gene and protein, and further utilize this knowledge to cure diseases, alter feature etc. The primary structure of gene or protein is expressed as a sequence of amino acid. A very important class of computational biology problem is to detect the similarity of related proteins or genes. The characteristic of this class of problem is simple operation (fixed point arithmetic only, usually only addition/subtraction and table lookup), large data volume(search against large and expanding databases. As far as computer architects are concerned the algorithms all look very similar [32] [3]. In the 1980s, people use supercomputers and sometime special hardware to deliver acceptable performance. The availability of these expensive facilities limits the capability of scientists. Now, some of the algorithm and databases are publicly available [13], researchers can access the web page to submit a query and wait for the result. The server of such site could become very crowded and slow, thus using cost-effective hardware to reduce latency and increase throughput of these server is certainly important and welcome.

2.2.1 BLAST

We choose to study the BLAST [3] algorithm because of its public availability of both code and databases.

BLAST (Basic Local Alignment Search Tool) is an optimized algorithm for fast sequence comparison. The algorithm tries to align two sequences (either of nucleotide or amino acid) so that part of the two sequences reaches maximum similarity.

Similarity is measured approximately by a matching score. Different individual amino acid demonstrates different but sometimes close functionalities. The closeness of two amino acids is measure by a predetermined score. These scores form a matrix. Different matrices exist for different purposes. The similarity of two sequences aligned in a certain way can thus be approximated by the sum of scores of matching amino acids. By trying different alignment, and different configuration of segments (since score can be negative, sometimes a subsegment of the sequence may yield a larger total of matching score) a maximum can be found. However, a exhaustive trial is impossible since it is too computationally expensive. Mathematical research has shown that using some very simple heuristics to cut lots of search branches the amount of computation is greatly reduced and yet the result is same or very close to that of the exhaustive search statistically.

In a query, often the submitted sequence is to be compared with many sequences in the database, so the the query string is broken into seeds of predefined width, the offset of such seeds are also recorded. Then a database string is analyzed from the beginning. If a segment of database string matches a seed in the query string then the two strings are aligned according the seed's offset. This segment is then extended to both directions. Extension is stopped if the accumulated matching score is dropped below a threshold. The database string is scanned from left to right and may contain many seeds to be extended, each such extension resulted in an MSP (maximal segment pair) beyond which extension will actually reduce the score. The MSP with the highest score is the final result for this pair of strings.

In practice there are some complications to the algorithm, but the big picture remains as described above.

2.3 Multimedia

The popular MPEG2 encoder is also included in the applications to study effectiveness of the intelligent memory on multimedia applications.

Chapter 3

Architecture

Introduced in the 1970's, the microprocessor has become the dominating technology to build all levels of computers. The growth of performance since mid 1980's has been much higher than that in the past, reaching more than 50% a year. A very large fraction of the growth comes from the improvement of VLSI technology. Recently, it is widely agreed that the increasingly complex architecture of microprocessors are showing diminishing returns due to the large and widening gap between the performance of the processor and that of the memory system. In this chapter a new architecture is proposed to provide an efficient means of utilizing the huge amount of transistors that current technology offers to deliver performance improvement.

3.1 Technology Trends and Their Implications

An effective architecture calls for balance among the various parts of a computer system. Any bottleneck in one part of the system could easily cut out the performance gained laboriously from other parts. For example, if a four-issue processor operating at 500MHz is stalled by a memory access for 150ns then 300 instruction issue slots are wasted. Thus for this processor it's obvious that to make the processor cache lock-up free is much more important than to make the processor core even wider.

Reconsidering the past twenty-five years we see these facts:[14]

- Microprocessor performance increases 35%-60% each year.
- DRAM density quadruples every three years.
- DRAM access time decreases one third over ten years.

This shows that the widening gap of processor and memory is not a transient phenomena. Thus it has to be bridged architecturally.

3.1.1 Latency

At the beginning of microprocessor's history, accessing memory was as fast as executing an instruction, while now memory latency is more than an order of magnitude longer than processor cycle time. This calls for caches to reduce the effective memory latency by exploiting program's spatial and temporal locality. The higher the hit rate the lower the chance that the processor has to experience the full latency of the memory access. Hence larger caches with higher associativity are favorable. Unfortunately these two factors hurt the speed of the cache and thus increase the hit time. To solve this problem, multi-level caches are developed. Caches closer to the processor are smaller, usually with low associativity and thus faster. Caches closer to the memory are slower, but larger and have higher associativity thus reducing the miss rate. Even with these caches, there are still cases when the processor has to go all the way to the main memory. Caches don't work for all applications. There are programs in which data are not reused much (thus lacking temporal locality) and the footprint is very large. In some cases, loading the data into the cache is even worse, since it displaces other useful data already in the cache. Moreover, checking tags of all levels of caches only to find that the needed data is not present actually makes the latency even longer.

3.1.2 Bandwidth

When the latency can not be reduced, it has to be hidden or tolerated. Out of order execution of instructions and multiple outstanding memory access all help hide the latency by overlapping memory access with other computations and/or with other memory accesses. These mechanisms all increase the bandwidth requirement of the processor. Moreover, recent processors have incorporated new features like speculative execution which further increases bandwidth requirement. With more transistors available on the processor, it can allow more overlap to exist, reducing stall due to memory access latency. This makes stalls due to bandwidth more obvious and important.

3.2 Ironies in Existing Architectures

If we examine the architecture of current computer system based on commodity DRAM and wide superscalar processors we can find that there are some ironies in the design.

3.2.1 Thrown-Away Bandwidth

Figure 3.1 is a typical DRAM structure. In the center there is a large rectangle of memory cell array where the actual information is stored. To the left there is the row decoder, which decodes the row address and selects one row of memory cells to drive the bit lines. The data on the bit lines is stored in the row buffer at the bottom of the array, and a multiplexer further selects a bit or several bits out of this row buffer according to the column address.

Row buffer size varies from design to design, but usually it's several kilo bits or more. The output width of a chip however is much smaller. A "wide" chip used for graphics may have 16-bit output while others usually only provide 1 or 4 bits. This means, by going off-chip, more than 99% of the achieved bandwidth is thrown away.

Moreover, with the size of the memory growing, it is now impossible to have a single memory cell array inside the chip, because memory access time is dominated by the RC delay of the bit lines. If the dimension of the memory cell, array is too large, then the access time would be too slow. Thus the chip consists of many macro cells, each with its own decoder and row buffer (logically so, physically the circuits implementing those function are usually grouped together) as shown in Figure 3.2. In a memory access only

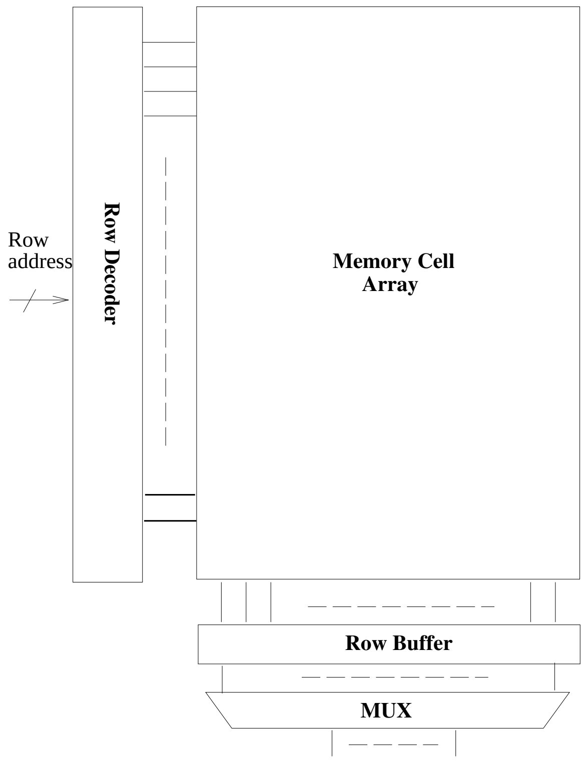


Figure 3.1: Structure of a DRAM Macrocell

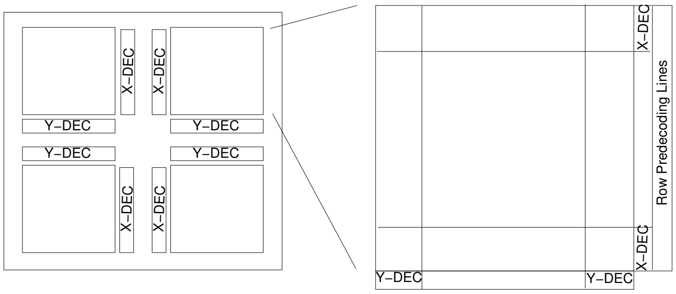


Figure 3.2: Structure of a Multi Macrocell DRAM Chip

one of such macro cells is active (if the chip has only one data output pin). Take a chip design by Samsung for example [36]. The chip has 1Gb and the cell array is 256kb, data width is 16b. This means every access only 64 bits are needed (assuming a typical burst length of 4), while there are 4096 cell arrays, this translates to wasting 98% of the potential bandwidth.

To sum it up, by going outside of the memory chip, we reduce the internal bandwidth by nearly four orders of magnitude.

3.2.2 Investment of Transistors

Although VLSI technology is advancing steadily, the increase of transistor count does not translate into performance gain linearly. In other words, microprocessors are seeing diminishing returns as far as transistor count is concerned. Current microprocessors are wide-issue, out-of-order RISC processors. A lot of the die area is devoted to instruction issue and graduation control, trying to maintain the program's sequential execution syntax while exploiting instruction level parallelism (ILP) at the same time. The core of the processor can manage a pool (also termed as buffer, window) of instructions and issue independent instructions out of the program order. However, the complexity of the logic grows super-linearly, if not exponentially, with the number of instructions in this pool. The issue logic, renaming mechanism and instruction retirement control grows exponentially with the issue width. Although even integer programs demonstrate ILP of up to several hundred of instructions, those independent instructions are usually too far away to fit in the window making the processor unable to exploit them. Take two MIPS processors for example. R5000 is an in-order processor while R10k is an out-of-order processor. The die size of R10k excluding that occupied by cache and TLB (leaving only the logic) is 6.3 times larger than the logic area of R5k, yet the performance of R10k on SPECint_base95 is only 60% better. The irony of this design trend is that more and more transistor as well as human resource (R10k costed 300 man year while R5k, 60) is used to exploit ILP which is harder to exploit, while the much easier data parallelism existing in many applications are ignored.[1]

On the other hand, there are these huge on-chip caches that took a lot of die size, and part of their function is to regain the bandwidth that was thrown when data leaves the memory chip.

3.3 New Architecture

The above mentioned ironies suggest two things:

1. Processor-Memory integration.
2. Processor to exploit data parallelism.

These observations are not new at all. People have been proposing similar ideas a long time ago. However, since logic process, on which microprocessor is based, emphasizes speed over density while memory process weighs density over speed, using any of these processes would not result in a good implementation of the proposed architecture. Only until recently has Merged Logic DRAM (MLD) [33] process become mature enough to implement serious architectures [28] [17]. Moreover, our design was not driven by the traditional benchmarks but rather driven by applications that we envisioned to be important in the near future.

[1] By uniprocessor.

3.3.1 System Architecture

In designing the new architecture, we adopted an evolutionary approach rather than a revolutionary one. We do not want to require the system to have so many changes in operating system, compiler support and application development, thus we make the processing in memory an advanced option that can be activated or deactivated. This means our PIM stays in the memory system as opposed to overhauling the system's processor subsystem. Ideally the chip of our new design would be installed in the memory slot and defaults to plain DRAM. When activated the chip would perform either part of application or execute generic intelligent memory operation functions. We call such a chip **FlexRAM** chip.

Another reason for assuming a more memory-like behavior is that the process of MLD is not able to produce very fast processing logic yet, thus building a CPU out of this logic would have performance disadvantage over the state-of-the-art CMOS microprocessors like Alpha on a variety of very important applications. Hence our architecture is cooperating rather than competing with current and future microprocessors which people are spending fortunes designing. A secondary advantage of the design is its portability onto different platform thanks to the standard DRAM interface.

Figure 3.3 shows the architecture of a computer system employing FlexRAM chips. The system is largely unchanged except the DRAM chips are substituted by FlexRAM chips which jointly provide the same memory interface as usual. There is also another optional interconnection besides the memory bus which connects the FlexRAM chips to form an effective multiprocessor in a single system. This interconnection is transparent to the host processor and only visible to the processor inside the FlexRAM chip. A low end system without such interconnect would be more restrictive in parallelism supported but would be much simpler and cheaper. I will not further discuss the interconnect in the thesis.

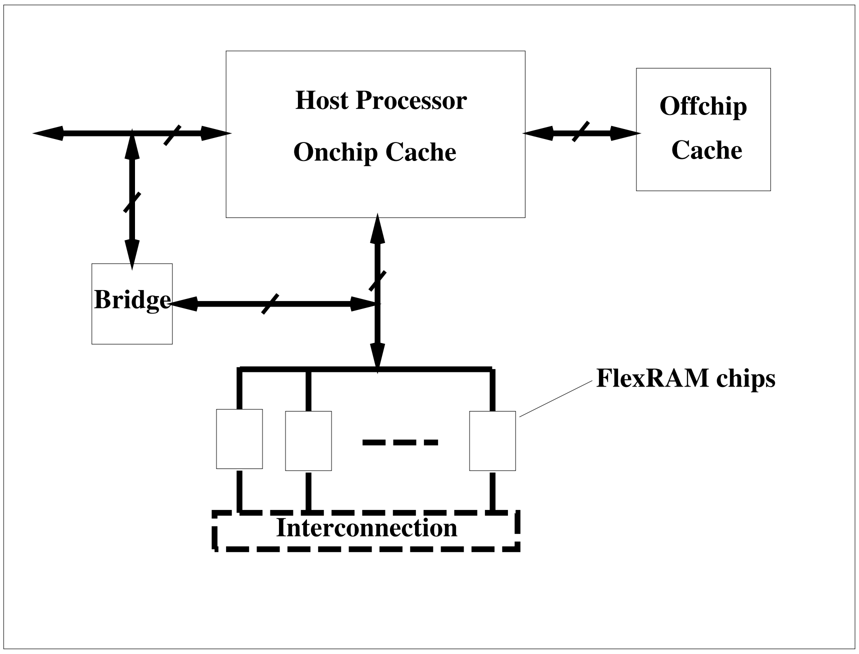


Figure 3.3: Structure of a System with FlexRAM Chips

3.3.2 Memory Chip Architecture

Now that we have defined the system architecture, and decided to integrate logic into DRAM parts, the question becomes what kind of processing logic to build into the memory chip. It has to be:

1. Simple, so that chip size does not increase too much and we do not have to sacrifice the size of the memory.

2. Able to exploit the bandwidth of the memory.

3. Able to exploit data parallelism in a variety of applications effectively.

These requirements suggest that we should build processor array that closely couples with memory macrocells and supports SPMD (single program multiple data) programs efficiently. Each processor should be integer only (floating point unit is currently too expensive since we have an array of processors), scalar processor without caches.

One decision we had to make was to choose between a SIMD processor array and a MIMD one. This is a trade-off between flexibility and complexity(die size).

SIMD architecture "Single Instruction Multiple Data" is an important category in parallel computer. Past architectures include Staran [5, 4], DAP [16], and of course CM-2 [15]. These architectures differ but all share the same features of a SIMD computer:

- Processor is bit-wised and connected with an array of memory bits.
- All the processors execute exactly the same instruction stream but on different data stream which comes from its own memory. Instructions are decoded by a controller and broadcasted to the processing element to avoid duplicating the decoding logic. The "instructions" that are broadcasted are very close to hardware control signals (e.g. an instruction is 96 bit wide in CM-2). Processors can be activated or deactivated by certain instructions.
- Processors are connected via some sort of interconnect.
- Communication patterns are usually fixed and synchronization support is minimal assuming mostly synchronous execution.

MIMD architecture The processors in a MIMD system are often much more loosely coupled. MIMD system can be shared memory machine or a shared-nothing machine, but regardless of this, the processors execute on their own thread and there is no guarantee of the relative speed of the processors.

Usually a MIMD node consists of an off-the-shelf microprocessor, a memory module (which could be part of the whole system's memory space) and a communication subsystem to connect to other fellow nodes. Depending on the efficiency of the communication system, a MIMD machine can range from fine grain parallel system to coarse grain one.

Different processors in MIMD machine can execute the same code, different code fragment for the same application or completely different tasks. Thus the processor has to have its own instruction stream.

FlexRAM When we compare the architecture of SIMD and MIMD computer we can see that there are some differences resulting from tradeoff between cost and flexibility.

When we design our architecture, we are not limiting ourselves to a certain paradigm, instead, we try to find some middle ground to combine their advantages.

1. Multiple instruction streams vs. single instruction stream

   SIMD processing element is simpler in hardware than a MIMD processor since lots of logic for decoding are shared among a large number of processors. This simpleness enabled SIMD computers to reach massive parallelism within reasonable cost.

   Our architecture is meant to exploit data parallelism in a cost-effective way, the SIMD execution mode is too limiting for us. It is designed to use the huge number of processors to offset the effect of low utilization. There are researches on simulating a MIMD mode using SIMD hardware [1], but not to satisfaction. The reason behind these researches is how to use the existing (and expensive) hardware to do something more then it is meant to. Thus we are not designing SIMD processing element, but rather have each processor has its own execution control and instruction decoder. Besides, we are at an age in which duplicating these logic is no longer costly. As can be seen from the layout, processor occupies reasonable die area.

2. word sliced vs. bit sliced

   SIMD computer goes a step further in reducing the cost of a single element by using only one bit processor core. A wider computation is achieved by taking multiple steps over different bit of the operand. Moreover, once upon a time, the speed of the carry bit propagation is the limiting factor of processing, thus a four bit processor might be two to three times as slow as a one bit counterpart, thus it favors using bit processor.

   Since memory access is the dominating factors of processors' speed, it is absolutely better not to process data bit-wisely. Speed of the carry bit is no longer important. All the factors leading to bit sliced processing have gone.

In fact, our decision was rather on whether to use 16 bit hardware or 32 bit ones. Since we are in an time of 32/64 bit processing and we are looking at a three year time frame to actually fabricate the chip, 32-bit makes more sense.

3. local control vs. shared control

   Since SIMD computers are used in single instruction stream mode, every process element execute the same instructions at any cycle, thus duplicating the instruction decoder does not make sense.

   As is shown, our processors would have multiple instruction stream, we have local control for them. We do, however, share some resources to minimize the die size devoted to processing. Instruction memory, for instance, is shared by four processors. Since we are looking mainly at SPMD programs (data parallel), sharing instruction memory saves greatly, because to fulfill the processor's execution pipeline, instruction memory has to be SRAM which is much less dense than DRAM.

   Multiplier is another shared resource. In the beginning, we are not including it in the design, but studying of the code shows relatively frequent usage of multiplication, without hardware support, the speedup gained by multiple processor would easily be destroyed by the slow software emulation of multiplication. Thus we included a multiplier for ever four processors.

4. asynchronous vs. synchronous execution

   Since every PE operates at the same clock speed, it is obvious that when one of them finishes an instruction every other PE has finished it also. Thus, most of the time, the program runs in exact synchronous mode, no explicit synchronization is required. On the other hand, for MIMD computer, even if the processor are executing the same program, there is no guarantee that they operate at the same speed, thus explicit synchronization has to be made.

   Given the processors are executing under the same clock input and they execute the same program code, people are arguing, some synchronization effort in code for MIMD computers are due to conservativeness rather than necessity [30] [29]

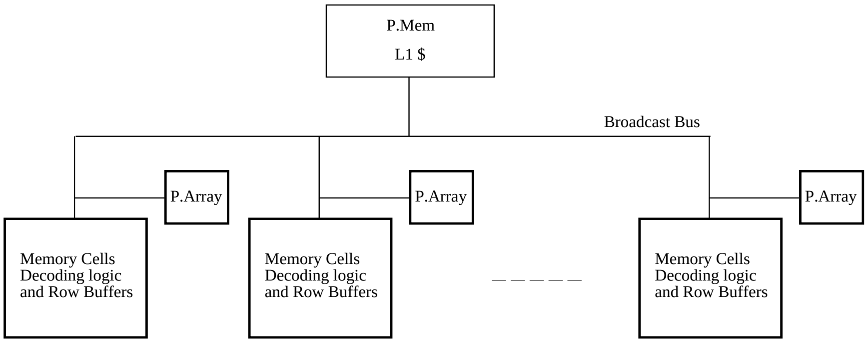


Figure 3.4: Structure of FlexRAM Chip

[8]. This observation also applies to our architecture and compiler can do their best in eliminating unnecessary synchronizations. Besides, when operating on the same chip, synchronization can be done much faster than cross chip or via software. Some simple yet effective hardware synchronization mechanism will be shown later.

The processor array in FlexRAM are essentially MIMD processors, they do not share memory, but each processor would be able to access the memory of its two neighbors. This sharing is not enforced to be coherent, so it is up to the software to enforce certain properties.

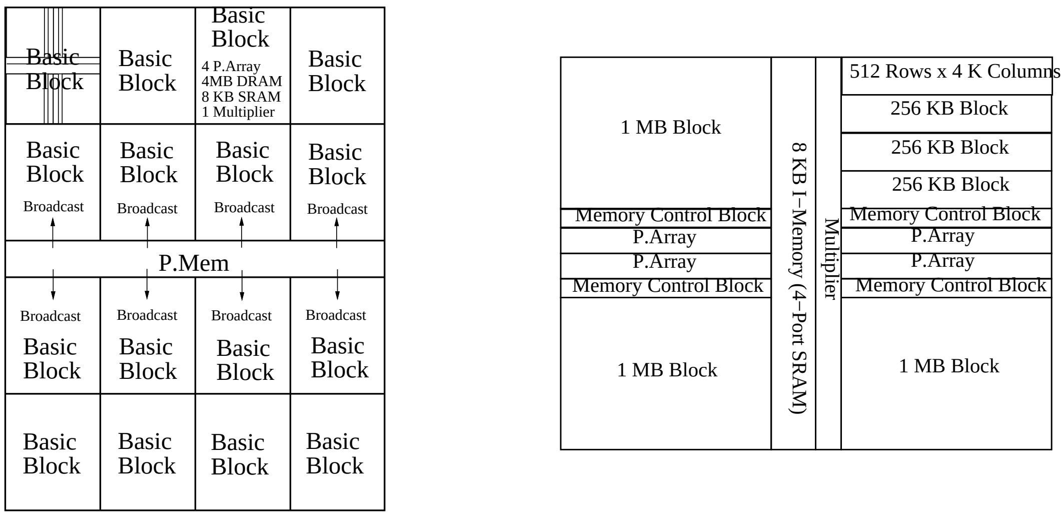
Given the inability of a array processor to access memory of a remote processor, certain communication and synchronization would be very complex to implement in software. Thus we also include a chip processor inside a FlexRAM chip, which we call P.Mem standing for processor in memory. This processor can access any location of the memory within the chip and thus can serve the purpose of global reduction etc.

Figure 3.4 shows the structure of one FlexRAM chip, and Figure 3.5 shows the layout of a prototype that we are building right now.

3.3.2.1 Array Processor

The processor core of P.Array is just a 32-bit ALU with 16 General purpose registers and some special registers for synchronization and other controlling purposes.

Hardware multiplier is expensive in terms of die size, but without which, a software emulation of multiplication is very slow, and multiplication in applications is not rare



(a)chip (b)basic block

Figure 3.5: Layout of the FlexRAM

enough to justify the software approach. Thus we let 4 P.Arrays share a multiplier. With the improvement in technology, later version of FLexRAM would be able to avoid sharing multiplier.

Hardware divider, on the other hand, is beyond question. It is considered too costly and would stay idle for most of the time.

As explained earlier, the logic overhead should be kept minimum in that we are enhancing a memory component. This makes it clear that hardware floating point can not be supported. Also, floating point performance is not so critical to a vast number of current and future applications. Besides, we are enhancing a computer system who does a good job in floating point computation.

The P.Array core is a very simple pipelined RISC core, with four pipeline stage: Instruction Fetch, Decode, Execution and Write Back. Memory access would incur an additional cycle if it hits the row buffer (3 if missed). Memory access simply stalls the pipeline.

To make sure instruction fetch does not affect the performance of P.Array we have to use SRAM for instruction. There are two approaches. One is to use SRAM as cache to hold the most recently used instructions from the DRAM, the other is to use SRAM to hold the code entirely, and this address space is not part of system's. Of the applications we studied, none of them require a large code space, (in fact many of them are one to two KB in size) this favors the latter approach. Using SRAM as memory also eliminates the overhead of a cache devices such as tag. Not being a part of the system's address space, SRAMs are assigned special I/O address space to facilitate uploading code by the host processor or the chip processor inside the FlexRAM chip (3.3.2.2) to load the code. Actually, since the P.Arrays are meant to execute in SPMD mode, uploading code are done in a broadcast manner, much more efficient then letting the processor writing to individual areas in the DRAM. Using four-port SRAM, we can let four processors share the code memory. In our first implementation, the size of this memory will be 8KB.

3.3.2.2 Memory Processor

P.Mem is a very simple RISC or first generation static superscalar processor core. We are not designing such a processor because any available core would be good enough to be the chip processor in our architecture. Even inside DRAM chip, memory access is still relatively slow so we add a very small first level cache to the core in the hope to reduce memory time. Also as illustrated above, P.Arrays can not afford to have floating point support, thus P.Mem comes with floating point unit. These expensive (in terms of chip area) devices can only be attached to P.Mem ,which there is only on per chip, not to P.Array.

P.Mem mainly acts as a coordinator to help P.Arrays to do jobs that is otherwise difficult to accomplish, but given many FlexRAM chips, there is a number of P.Mems in them and can become a multiprocessor.

Of many jobs P.Mem could do, reduction is a very important one. Very often, data parallel program needs to summarize a series of computation by doing a reduction like adding the numbers in an array together, finding the largest number among many. For example, in the BSOM algorithm (2.1.2), the weight vectors needs to be updated after each epoch. This is to sum up the results computed by the parallel processors. In our architecture, P.Arrays would do the local accumulation, and then after barrier, P.Mem can read the partial result of every processor, add them up and write it back to the memory of all the processors. A hardware reduction mechanism would be much faster than the software approach, but since reduction comes in different forms such as finding the maximum/minimum number, accumulating partial result etc., it is costly to implement a general enough mechanism, again, the driving force of the design – application needs do not justify such an approach. However, nothing prevents the reduction hardware to become inexpensive in under a future technology.

3.3.2.3 Synchronization Logic

Barrier Unless executing program in exact synchrony, multiple processors have to synchronize with each other before entering another stage of computation. In Shared Memory multiprocessors, by using software barriers processors can synchronize in flexible ways. In our system, however, P.Array's can only access the memory of its nearest neighbors, using this way to synchronize the whole chip often results in slow speed and complicated software solutions. By adding very simple hardware, we can achieve very effective hardware solution that can satisfy most of the applications requirements.

There are more than one ways to have a hardware barrier, here are some of them:

1. **Counter based barrier**

   Naturally developed from the software barrier, we can think of a counter in the center of the chip. Every P.Array has a dedicated line connecting to the counter. Also, there is a result line extending to every processor, once a processor reaches the barrier, it simply asserts the line indicating arrival at the barrier and then spins on the result line. In practice, these lines are hardwired to certain bits of the special purpose register of each P.Array. Initially the counter could be reset to the number of processors to synchronize at the barrier, then it will decrement once a line is asserted, when the counter reaches zero it can flip the result line permitting all the processor to proceed over the barrier. Having passed the barrier, each P.Array resets its bit for the next barrier. Like in software barrier, the technique of sense reversing [14] is applied. P.Mem is responsible for setting the barrier number for each occurrence.

The advantages of counter based barrier are:

- it allows arbitrary number of P.Arrays inside a chip to synchronize.
- the distribution of the processors to synchronize has no limitation

The disadvantage of counter based barrier are:

- P.Mem is virtually spinning all the time preparing the counter
- only one group of barrier is allowed since there is only a centralized counter.
- complex control logic to count the asserted lines and then do the subtraction to the counter.
- complex wiring: a lot of lines from every corner to the central counter.

2. **All-all barrier**

   By using the circuit in Figure 3.6 we can build a very simple all to all barrier for eight processors. The circuit has the feature that only when all input turns to logic one does the output change to one, and the output keeps at logic one until all inputs become zero when the output will become zero. Using this barrier, each time a processor reaches the barrier it flips its synchronization bit, and spins on the result bit to be the same as its own synchronization bit.

   Applying two level of this circuit as in Figure 3.7 we can build a barrier for the whole chip. In addition to the global result bit we have eight local result bits. This allows processors to choose between a global synchronization and a local one.

   The advantages of all-to-all barrier are:

   - Very simple logic, and even without clock input, there is no glitch in the circuit.
   - Local wirings: only the result bit of the local barrier circuit travels to the center of the chip, making it easier to scale.

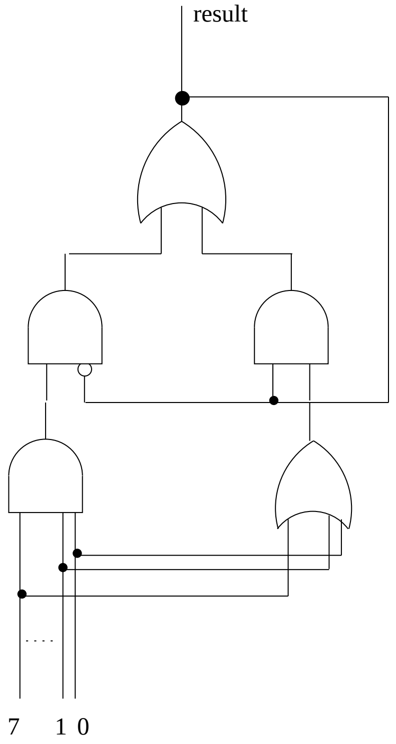


Figure 3.6: 8-Processor Barrier

- by wiring the local and global result bit to different bit of special purpose register, program has flexibility to choose.
- allows many simultaneous barriers.

The disadvantages of all-to-all barrier is:

- only all the processor in the group or the whole chip can synchronize which is less flexible than the counter based approach.

3. **Other barrier**

Since some of the processors do have physical shared memory, it is easy to implement software barriers for two or three processors. This will be useful for producer-consumer type of communications.

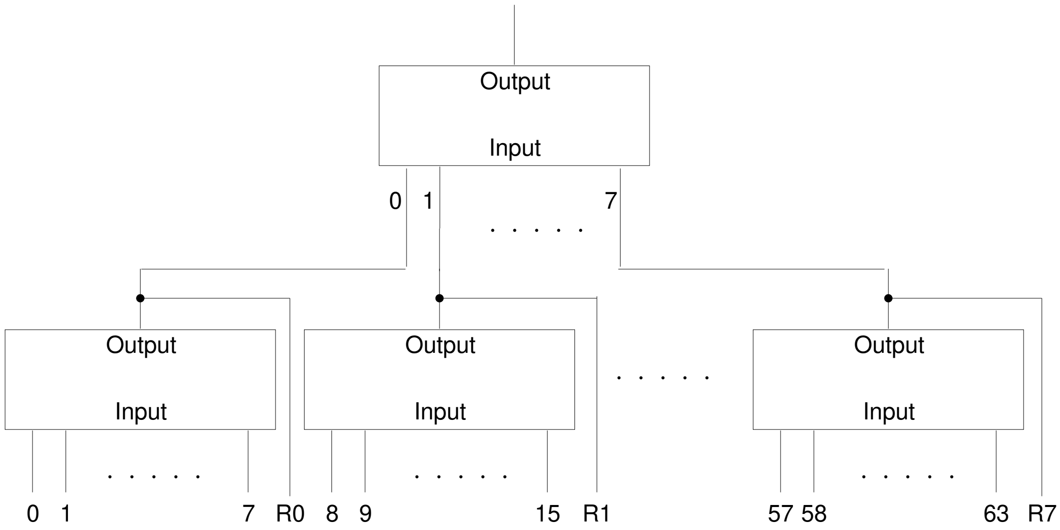


Figure 3.7: Two Level Barrier

Lock Unlike communication incurred synchronization where mostly one processor reads and the other one writes, shared writable variable/data structure require lock-based mechanism to ensure atomicity in memory update.

A hardware lock protect a memory block from being locked by another processor. Once a hardware lock is acquired by processor A, processor B can not acquire the lock, however, it can still read write the memory. In this way, program has to be sure any update to shared writable data structure has to be preceded by a lock acquire to that memory block. A read of shared writable data structure also has to be protected by lock if data consistency is required. A lock forbids other processor from access the shared data structure in the whole memory macroblock, so if this activity is frequent, and sharing is extensive software locks can be allocated to protect different data structure, once a software lock is acquired, the processor can release the hardware lock to the whole memory block and continue to update the data structure. Consequently, access to unrelated critical data structure will not result in unnecessary serialization.

A lock is another pair of bits in the special purpose register, processor asserts one bit to acquire the lock, it then spins on the other bit to wait for the lock to become available.

The circuit in Figure 3.8 is a simple one for three processors. When requests to the lock comes in the same cycle there is a statically assigned priority to the break the tie,

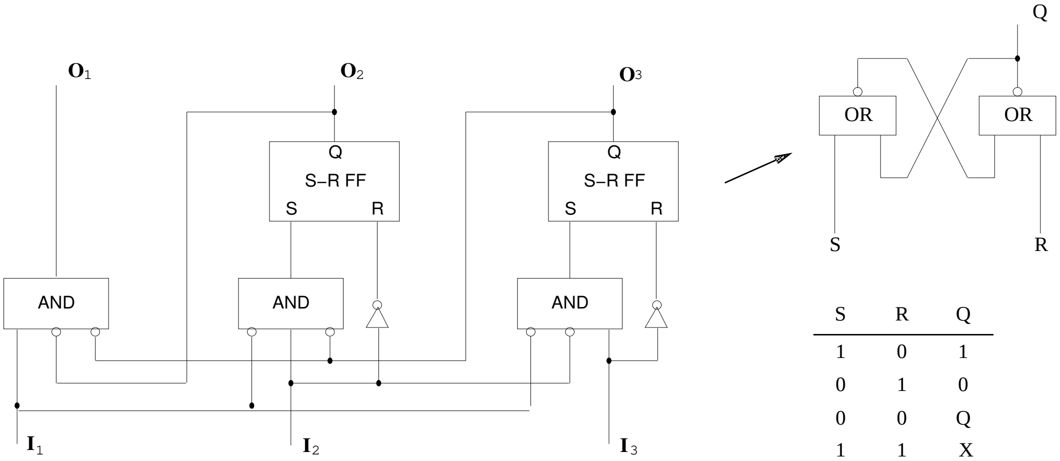


Figure 3.8: Lock Arbiter

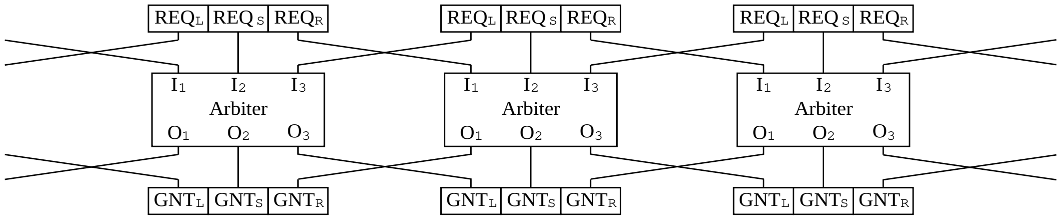


Figure 3.9: Connection of Arbiter

this, in a very rare situation would cause starvation to the processor with lowest priority. A more complicated circuit can implement a round robin priority, but in reality the simple circuit is enough.

The inputs and outputs are hardwired to the special purpose register, so the program just set the bit and spins on the special register. Since this step is inherently clocked, the arbitration logic are purely in combinational logic to be simple. Also, the S-R flip flop can tolerate glitch when the input is deasserted.

Figure 3.9 shows how to connect the processor core with the lock arbiter shown in Figure 3.8. REQ and GNT show the hardwired bits from the special purpose register, REQ for request bits and GNT for the grant bits. The subscript L, R, S shows the bit for the memory block of its left neighbor's, right neighbor's and that of its own respectively. The mapping between the bits and the input of the arbiter can be arbitrary thus implementing any static priority order.

3.3.2.4 Row Buffer

Even the latency to access DRAM core inside the DRAM chip is significantly smaller than that of accessing from off chip, it still pays to have row buffers to cache the recently accessed data. Nowadays it is common to have more than one row buffer. A memory access starts by checking the row buffer, if the row buffer hits, the DRAM access is avoided. When there are more than one row buffer, they are checked simultaneously. One complication in introducing more row buffers is displacement. We will see in chapter 4 how different replacement policies will affect the hit rate of row buffer.

Chapter 4

Evaluations

The potential speedup of a program's execution on a system with FlexRAM can come from two parts. One is by pushing operation close to the memory, data-intensive operations get executed faster because of the lower latency accessing memory inside the chip, also processing in memory avoids polluting the cache with data that is not reused much. The other one is by exploiting parallelism using the many P.Arrays in the memory chip. The first approach though promising has difficulties demonstrating its potential for many reasons:

- Memory accesses are all associated with some processing and current technology still can not make the processing logic in the DRAM process fast enough to compete with state-of-the-art microprocessors, so the time saved in accessing memory might be wasted executing instructions.

- Modern processors is becoming more and more capable of tolerating the long latency by increasingly aggressive out-of-order design. Also, technology improvements has allowed processor larger and larger transistor budget to allocate to on-chip cache memories. Though processing in memory is a much more cost-effective solution, from a pure performance perspective, the traditional system still seems competitive in many application domains.

- It is still a great challenge to solve all the problems in dispatching code to the memory like which part is to be done in the memory, how to make the data coherent, and how to synchronize efficiently.

In this thesis, I will concentrate on the second part of the speed up, that is, speedup from the exploiting the parallelism. [1] When looking at the execution results, one has to keep the following in mind: unlike in the case of start-of-the-art microprocessor design, there are no SPEC like benchmark suite to evaluate an intelligent memory performance, it still remains a research topic to find the most suitable applications to exploit the IRAM philosophy. The application we used is just to give an initial insight in the potential of general purpose IRAM approach and to pave a way for quantitative evaluation of design trade-offs. There are still a lot work to do to make the applications more mature, for instance to revise the algorithm and introduce IRAM-aware compiler optimizations.

4.1 Evaluation Methodology

Throughout the project we use different flavors of MINT [34] based performance simulator to evaluate the architecture. To simulate state-of-the-art speculative wide superscalar processors we used an enhanced front end[24], and modifications are made to allow the simulator to simulate different configurations for different processors in a single run of simulation. [2]

Two versions of each program are used to test different architecture, one is the program in its original form, this is run on an aggressive platform with a six-issue out-of-order core. Another modified program is used to simulate the execution inside a single FlexRAM chip, basically with an two-issue static processor as P.Mem and 64 P.Arrays.

[1] Strictly speaking the performance gain comes from the combined effect of faster close-to-memory access and parallelism since the fact the P.Arrays has low latency accessing memory does help the overall performance.

[2] Recall that the host processor (P.Host) is wide-issue speculative out-of-order processor, while P.Mem is a narrow issue in order execution engine and the P.Arrays are effectively pipelined scalar RISC processor.

The program includes all the necessary overhead such as synchronization and communication among the processors. The detailed configuration of the two architectures is summarized in table 4.1.

P.Host	P.Host L1/L2	Memory
Frequency: 800 MHz Issue Width: 6 Dynamic Issue: Yes Instr. Window Size: 96 Ld/St Units: 2 Int Units: 6 FP Units: 4 Outstanding Ld/St: 16	L1 Size: 32 KB L1 Latency: 2.5 ns L1 Assoc: 2 L1 Line: 64 B L2 Size: 256 KB L2 Latency: 10 ns L2 Assoc: 4 L2 Line: 64 B	Bus Type: Split Transaction Bus With: 16 B Bus Frequency: 100 MHz L2 Fill Time: 190 ns
P.Mem	P.Mem L1	P.Array
Frequency: 200 MHz Issue Width: 2 Dynamic Issue: No Ld/St Units: 1 Int Units: 2 FP Units: 1 Outstanding Ld/St: 8	L1 Size: 8 KB L1 Latency: 5 ns L1 Assoc: 2 L1 Line: 32 B L2 Cache: No	Frequency: 200 MHz Issue Width: 1 Dynamic Issue: No Outstanding St: 1 Row Buffers: 3 Row Buff Size: 2 KB Row Buff Hit: 10 ns Row Buff Miss: 20 ns

Table 4.1: Parameters of the Architecture Configuration

4.2 Execution Results

The benchmarks that we selected are mostly parallel in nature, some of them are data parallel. The way we parallelized the code is to try to evaluate the potential of the processor array, so we try to let these simple processor execute as much code as they can, they are limited mainly by two things, communication and floating point capability, and here is what the P.Mems come into play. [3] The algorithm is also modified to use the architecture efficiently. The execution is thus roughly separated into stages, in which

[3]The only exception to this execution model is GTREE: decision tree generation, in which the P.Arrays are only involved in counting the number for each class at each tree node, so that P.Mem could gather all the information and determine the test that can maximize the gain ratio. (section 2.1.3)

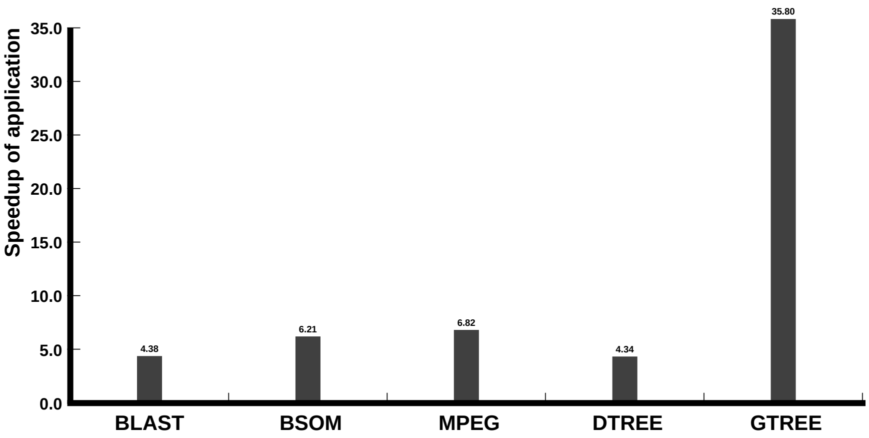


Figure 4.1: Speedup of Benchmarks

P.Arrays execute independently and then synchronize to allow P.Mem to do some data shuffling, reduction and floating point processing if necessary. We do not involve the host processor. The unmodified serial version of the code is simulated on the host processor as a reference point.

The speedup of the five benchmarks are shown in figure 4.1. Except the GTREE which uses optimized algorithm the rest achieves speedup of around 5 to 7. Remember that compared to a six-issue processor running at 800MHz, P.Arrays are static single issue processors running at 200MHz only. The combined total issue capability of 64 P.Arrays is only about twice as large as that of the host processor P.Host. Thus speedup of 5 to 7 in the presence of overhead of parallel execution shows the efficiency of the architecture. The breakdowns of the execution time is in figure 4.2. PA/PA waiting time is the synchronization time for barrier among P.Arrays while PA/PM waiting time is the cycle spent waiting P.Mem to do data shuffling or floating point computation and so on. It can be observed that busy time constitute for more than 50% of the total execution time, this means when clock speed improves, further speedup can be achieved.

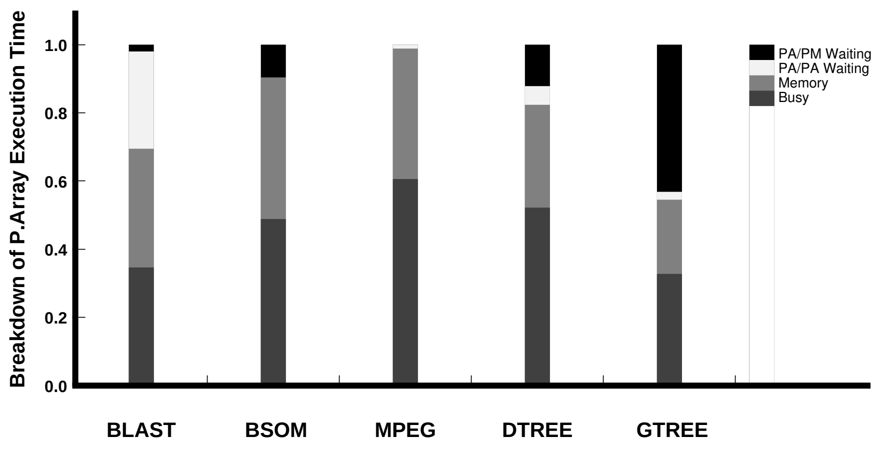


Figure 4.2: Breakdown of Execution Time on P.Array

4.3 Row Buffers

The P.Array processors are closely coupled with the memory macrocells, also unlike the host processor, a memory access does not have to go through a hierarchy of memory components thus it experiences a much shorter latency. Even so, the DRAM structure (thus the access method) makes even the access from P.Array slow compared to the already slow P.Array clock frequency. In the future, advanced MLD technology would render the clock frequency inside the chip comparable to that of an aggressive single processor, therefore this speed gap still needs to be addressed. A full blown cache to serve the purpose would make the design too complex and largely affect the cost-effectiveness of such a system, thus using the row buffer to address the problem is an obvious alternative. Row buffer is a special case of cache with the line size of the width of the memory array (usually very large), and very few, if not only one, lines. The good part of it is the large line size which exploits spatial locality, and unlike the cache line size, it does not lengthen the fill process, since the row buffer is directly connected with the cell array where as the cache line has to travel through a narrow bus with heavy load (thus slow), thus it comes for free [4]. The down side, however is more obvious: with very few buffers, access

[4]With multiple row buffers, however, the circuitry does contain extra overhead, but since we are using low clock frequency, such overhead is not a problem.

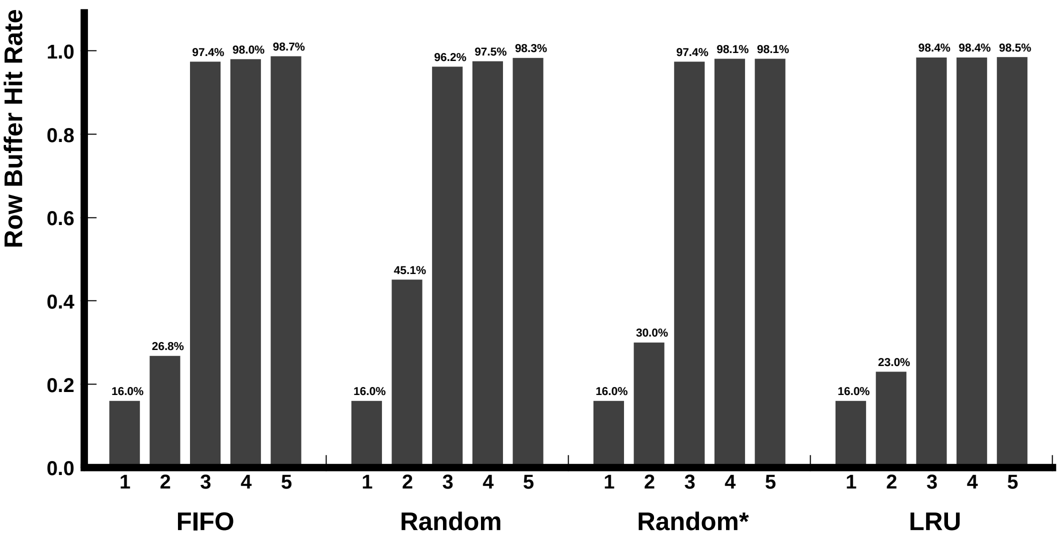


Figure 4.3: Row Buffer Hit Rate for BSOM

to different data structures from different place in the memory displace each other from the row buffer and offsets the benefit of large line size.

From the analysis of the source program we find that most program would benefit from at least two row buffers: if we user only one row buffer for MPEG it results in 100% miss rate (figure 4.7 while in the case of BSOM, (figure 4.3) the third row buffer brings the greatest hit rate improvement. Though there are many factors in deciding the number, organization and replacement algorithm of the row buffers, here we show the row buffer hit rate for several benchmarks assuming a fully associative structure. The figures shown are average of all P.Array row buffers. The replacement algorithm includes FIFO, LRU, Random and Random*. The "Random*" is introduced as a slight optimization over Random. A pointer is kept to point to the most recently used buffer, the random pick will avoid displacing this one. The Most Recently Used pointer is only updated during a read hit. This enables this scheme to outperform LRU in two row buffers configuration [5]. Notice that LRU usually does not show performance advantage and is, in some small row buffer number configuration cases, significantly worse that Random schemes . This

[5]Without this optimization, the Random* is exactly the same as LRU in two row buffers case.

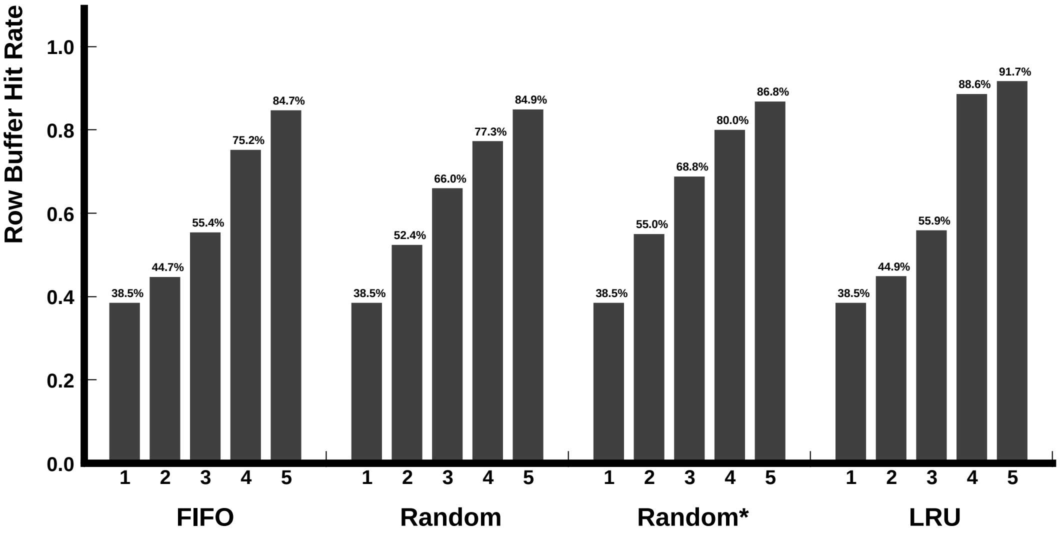


Figure 4.4: Row Buffer Hit Rate for BLAST

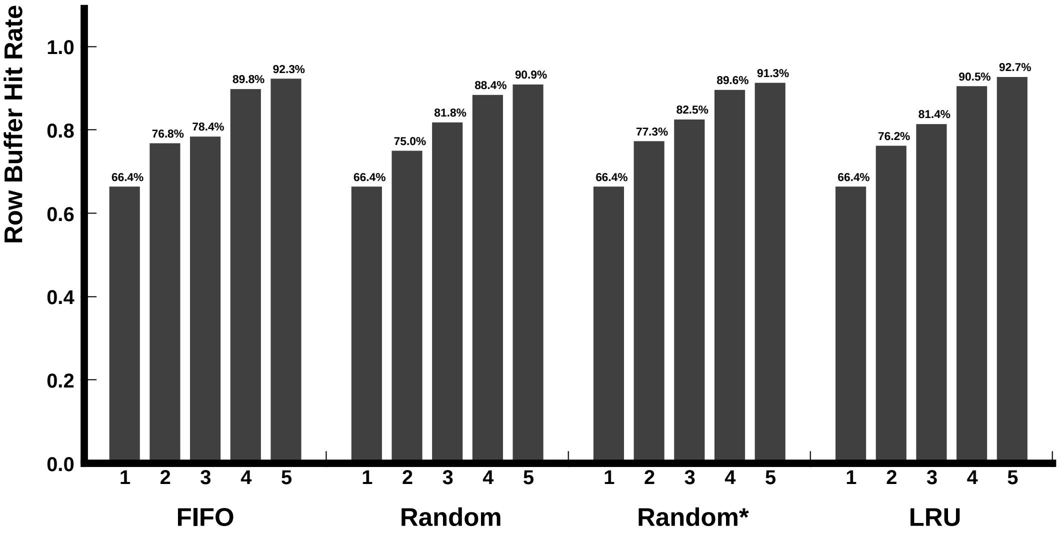


Figure 4.5: Row Buffer Hit Rate for DTREE

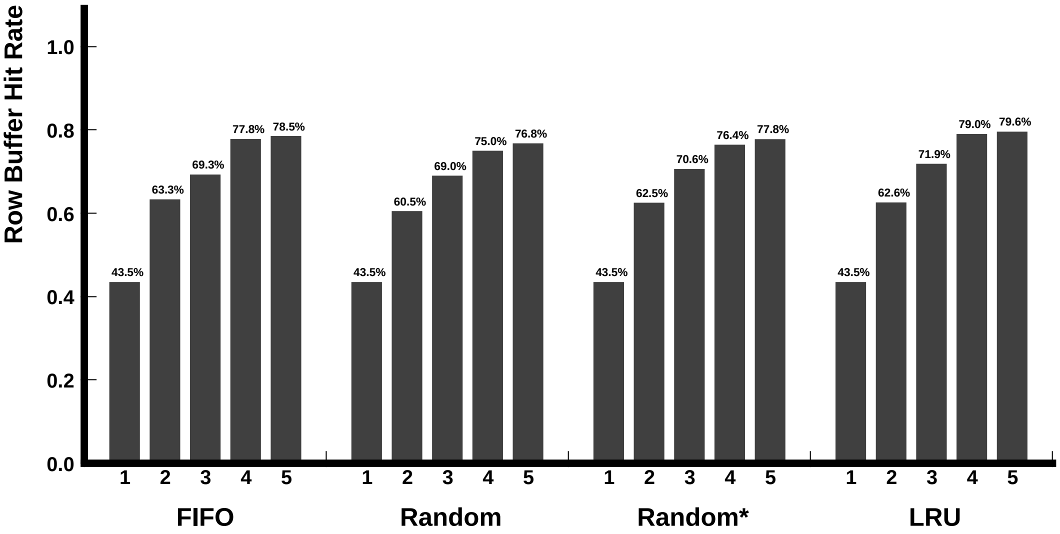


Figure 4.6: Row Buffer Hit Rate for GTREE

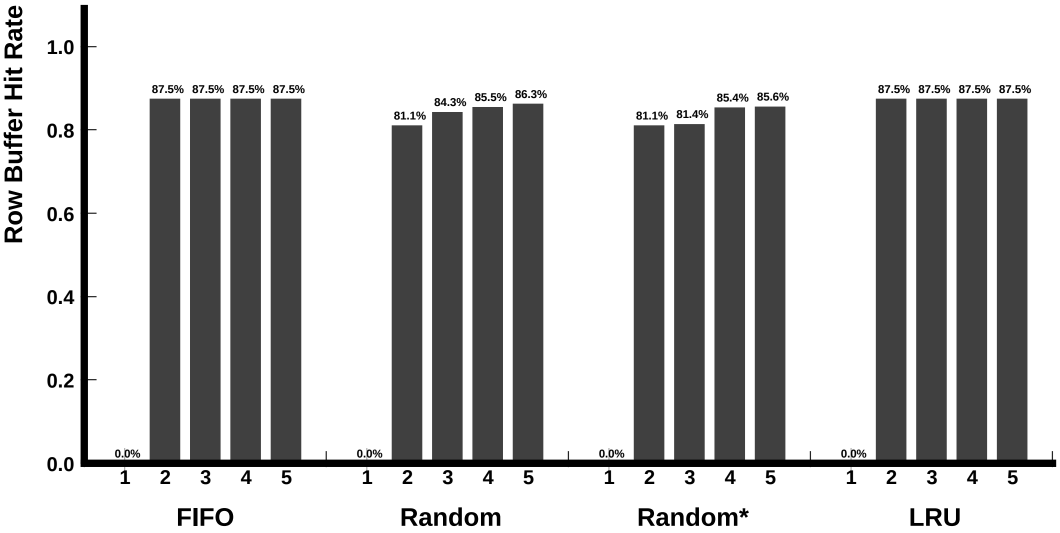


Figure 4.7: Row Buffer Hit Rate for MPEG

is because the small number of row buffer can not catch several localities at the same time, if we replace them strictly in LRU fashion we are guaranteed to displace what we need when well roll back to the first locality. For instance, if the program access three big arrays A, B, C in a loop, making the access appears:

$$A, B, C, A, B, C, A, B, C....$$

then for a two row buffers configuration LRU and FIFO guarantees that we miss every time, while random based algorithm will give some hits.

These results show that, if design headroom allows, we should include at least two row buffers, and preferably three. Beyond three, improvement is too little to justify the cost since modern DRAM architecture renders row buffer overhead relatively higher than in the past. On the other hand, with the improvement in technology, the speed of in memory processor will be steadily increased, this would make the memory performance more important, thus the single row buffer approach would be undesirable. As far as replacement algorithm is concerned, Random* gives more stable performance, but the schemes doe not make too much difference. Design complexity and cost of hardware should be used to determine the replacement algorithm.

Overall, the simulations showed that using a very simple processor array and a previous generation processor core, a FlexRAM chip can outperform, by a factor of 7, a next generation aggressive superscalar processor on a series of important applications. Given the ever increasing foot print of modern software, multiple FlexRAM chips can be used to produce even higher speedups. While we associate such aggressive features like multiple LD/ST units, very large instruction window [6] and very large outstanding memory operation queue, we are conservative in our FlexRAM parameters. The clock frequency

[6] 25% larger than the largest industry configuration yet to ship

is from current generation MLD [7], and the P.Mem core is previous generation product. All this demonstrates the potential of FlexRAM approach, and shows the promise of IRAM philosophy.

[7] Industry already have new generation MLD process that fabricate logic not much slower than that in a logic process.

Chapter 5

Related Works

Many projects work on bridging the processor/memory speed gap architecturally. Most of them suggest some form of processor/memory integration. In this chapter I list some of them and explain the difference of their scheme and ours.

5.1 DIVA

Data Intensive Architecture (DIVA) [12] [1] is perhaps the closest scheme to ours.

The slides shown on the web site [12] does not reveal too much detail about the project. From what is shown, it can be seen that DIVA's PIM chip is more complex than FlexRAM. First, virtual to physical address translation is done in hardware and there is TLB consistency management inside the chip. Second, there is floating point support on chip. Third, extra support for conditional SIMD such as masking and packing logic.

The DIVA slides also emphasize on the inter chip communication to make the system good for MPP.

The differences in the two systems result from different design points. We believe it is crucial to keep the PIM chip very simple and thus more cost effective, while the DIVA

[1] At the time this thesis is being written, there is no known publication about this project, even the web site referred by the "DARPA / ITO Data Intensive Systems Principal Investigator Meeting" web page is not functioning.

approach is more powerful. We did not intend to build the state of the art supercomputer out of FlexRAM but rather would bring the cost of a departmental server level machine down to that of a high end single processor server, thus we have opted to leave some non-critical functionality to software.

5.2 Active Pages

Active Pages [31] is also very close to our scheme. In their proposed implementation RADram (Reconfigurable Architecture DRAM), associated with each memory macrocell are 256 LEs (Logic Element). An LE is a standard logic block of FPGA. In RADram approximately half of the chip area is devoted to logic while the other half, DRAM cells.

When using Active Pages, the configurable logic is first programmed, then the pages are accessed by the FPGA logic and finally results are collected by the processor. There are no ways to communicate between two LEs the only means of synchronization between an LE and the main processor is through polling some predefined synchronization variables in the memory. There is no synchronization facilities to synchronize different LEs without involving the processor.

The basic ideas behind Active Pages are similar to ours: shifting some operation to memory where it can be done faster, also parallelism is exploited since there are moderate number of FPGAs.

The differences between the two approaches are quite secondary and implementation oriented.

First, we emphasis minimal modification to the existing system. While RADram also keeps the processor part of the system, they do require programmers to program application specific code in FPGA. In our system, user only have to write program in high level languages like C.

Second, our design are driven by studying real world whole application as opposed to laboratory-invented code or extremely tight kernels. Thus, it is imperative to make the processors more flexible and have the capability of doing basic communication and synchronization because otherwise the logic component inside the memory chips become so limited and either the parallelism supported becomes limited or the processor get involved all the time and renders a serialization whenever some kind of interactivity occurs. Unlike in our approach where a chip processor exists to help when the work requires more than the hardware can do, and there is a somewhat fixed ratio between the chip processor (P.Mem) and the array processor (P.Array) so the serialization involved is moderate, if any. In RADram however, whenever there is some interactivity, the main processor get involved, and notice that it is the only processor to help so many active pages in many chips. This either limit the application domain, or the system's scale, or worse still, both. Indeed we can see although the system demonstrates very impressive speedups (let us forget that the author are simulating a SCALAR reference processor which stalls on every memory access which exaggerated the problem of memory speed unnecessarily) they all come from tight kernels repeating many times to amortize the overhead whatsoever. In our evaluations, we choose real world program with shrank dataset to allow fast simulation, in other words, the overhead could be even smaller with real data set. It is our observation of real world applications instead of toy code that drives our design that includes some extra support other than a simple ALU that we believe is crucial to ease application development and deliver good performances.

Third, Our logic inside memory chip is a general purpose RISC core which is not special purpose engine. This is flexible and easy to use. RADram, on the other hand, uses FPGA. The benefit of using reconfigurable logic is its uniformity which gives higher yield of the chip, and perhaps in certain cases application specific FPGA works faster than a program on a general purpose computer. However, the price for these benefits is its significant limitation on application domain. Only very simple operations can be done by the hardware. As can be seen from the paper, the data structures are very simple.

5.3 Berkeley IRAM

Having looked at the potential performance benefit of IRAM architecture applied to existing architecture [6], Berkeley people proposed the IRAM processor architecture [23]. This architecture combines scalar CPU, caches, vector unit and DRAM on a single chip. Vector unit can better utilize the potential bandwidth of onchip DRAM, and it is also claimed vectorization is a very mature compiler technique and many application can be vertorized [11]. The reasons for choosing a vector approach also include consideration of energy efficiency and yield of the chip. While we do agree with these observations we still believe vector approach is more limited than the MIMD alternative. In retrospect of the past architectures we can see that nearly every architecture proposed is a good design and has no reason to fail in itself. Most of the failures are because they are not widely accepted. We therefore consider it a good trade-off to make the architecture as flexible as possible.

As pointed out in [11], IRAM architecture has many applications like: video game, PDA, intelligent disk, low-cost data server and low-cost super computer. We believe it is hard for a single architecture to dominate all these domains, therefore differences in implementation in these different applications are inevitable. In our case, we are aiming mainly at desktop/deskside low cost server for very large volume data-intensive processing, thus we believe, flexibility is a more important consideration than energy efficiency and the like.

5.4 Execube

Aimed at an efficient implementation of MPP, Execube [21] includes everything an MPP building block needs onto a single chip, this approach is necessary to implement an MPP at a very low cost. Given the industrial settings and the state-of-art in 1994, Execube has with each CPU memory of only the size of a first level cache for current processor also, it includes all the glue logic to make an MPP run. With a very small memory, there is really not much potential bandwidth to exploit, and there is not much room to trade off what to include in the system either. Consequently Execube is more a low cost solution for MPP than a processor in memory approach to tackle the memory wall problem and balancing the system.

5.5 Others

Apart from the above mentioned projects, many groups have proposed new architecture and carried out simulations on processor memory integration. Although few seemed to be revolutionary or inspiring they all contribute more or less to the quantitativeness of the big direction of integration that everybody agrees. I list a few of them here.

5.5.1 Chip Multiprocessor with DRAM

Given the ever increasing number of transistors one can put on a chip, and the diminishing return of wider processors, people are more and more reluctant to make their processor core fancier. A lot of people believe in *Simple and elegant* approach similar to the RISC philosophy. Chip multiprocess is proposed by such people. With still more space on a die and the universally acknowledged speed gap's expansion, they decided to put some memory on the chip. Under the current technology, it is obvious that there are two options: whether to integrate SRAM or DRAM with the processor. Intuitively when the working set is small and fit in the onchip cache, it should be better to make the onchip memory SRAM so that memory access is much faster. When the working set is large and an onchip cache can not capture the data, and results in large traffic to the slow off-chip DRAM then it becomes tempting to put much denser DRAM on chip so that much of the traffic are kept on chip. In [35] simulations affirm intuition.

5.5.2 C•RAM

C•RAM [9] is a simple proposal to put SIMD processor into DRAM IC. Even the author admit that if C•RAM was to run only user-written program there would be a small market indeed. This is limited chiefly by the narrow computation models SIMD allows.

Besides those mentioned here, there are also countless schemes for very special purpose of applications like multimedia [20], I/O [19], nearest neighbor searching [26], ATM table lookup [10] vector for large scientific computation [18], [2] etc.

5.5.3 RAMPage

RAMPage [27] is very different from all the processor memory integration proposed above, yet it is also a scheme targeted at the memory gap. The key observation is that DRAM is in a sense like a hard disk: it has long latency, it is better accessed in chunk to amortize the cost. Thus they propose to treat DRAM indeed as a disk, that is, organize the lowest level of SRAM as page based main memory and fetch in a whole page from the DRAM device when there is a page fault.

This changes several aspect of the lowest level of cache. First, this cache can be accessed using an address by the CPU and no longer transparent architecturally. Second, the memory is virtually fully associative since every page can be mapped everywhere in the address space. Notice that the page size here does not have to be as large as that in a traditional system it can be as small as a typical cache line. Third, when a miss happens it is served in software (handling a page fault) as opposed to hardware cache fill. Since the processor is becoming faster and faster, the DRAM access latency is becoming longer and longer relatively, thus slowdown caused by software handler becomes less obvious allegedly. By doing it in software flexibility is obtained.

The ideas seem to make some senses, but the problems are more obvious. SRAMs suffer from the density problem, so this new main memory would be significantly smaller

[2]The project called DIVA here is an acronym of DIstributed Vector Architecture different from the other DIVA project which stands for Data IntensiVe Architecture.

but the pages that have to be pinned in the system (page tables fault handler and other crucial OS data structures) remains large. Also, the small memory size would cause too many page faults(even a DRAM main memory could not hold large problems' working set).

The simulations shown in the paper actually argue against the author's claim for RAMPage. First of all, the authors simulated a non-pipelined, non-superscalar processor (which is the state-of-the-art 15 - 20 years ago), this strongly favors their solution because current processors can overlap a memory access with useful computation and also can overlap multiple memory accesses to significantly reduce the apparent memory latency. Apart from this, only when the reference system uses a *direct mapped* cache can the RAMPage outperform it, a simple 2-way set associative cache would easily wins over. This suggest although seemingly reasonable, RAMPage is the wrong way to tolerate latency problem under the currently typical configuration.

Chapter 6

Conclusion and Future Work

Given the current state of wide processor and memory performance gap, it is attractive to integrate processor core with DRAM on a single die. In this thesis the pros and cons of different integration approach are studied, leading to a flexible system organization using Merged Logic DRAM process technology. The architecture utilizes the huge number of transistors that modern process can integrate, in a cost effective manner. This thesis shows how the system could benefit from integration and exploiting data parallelism by detailed execution driven simulation of five representative benchmarks in important domains. Also, we show that in the presence of overhead from parallel processing and inefficiency, software could still benefit significantly from parallel architectures. In the simulation, assuming very conservative parameters for the proposed architecture and with unoptimized execution which spent between 1% and 46% of time in synchronization, four of the applications show an average speedup of 5.7 while the fifth shows speedup of 35 all with only one FlexRAM chip.

On the other hand, from an implementation point of view, the FlexRAM architecture is evolutionary rather than revolutionary, keeping the modifications to the system to a minimum. The memory system is backward compatible with state-of-the-art memory systems, enabling incremental performance gains without losing the ability to run legacy code. By avoiding putting large amounts of resources in already very aggressive out-of-order processor, and addressing one of the greatest bottlenecks in system with such processors, we achieve significant speedup , while in the mean time maintaining cost-effectiveness.

The work shown here, however, is still in a developing stage. There are still many optimizations to be done. For example, there is no overlap between the two levels of processors. There is no specific optimization in compiling the code for P.Arrays. For example we do not do data blocking or data placement optimizations to maximize row buffer hit rate. The code is now being manually transformed to run on different processors, while this could done at programming language/compiler level to minimize software development effort.

As we mentioned previously, the speedup shown here mainly comes from parallel execution of the code, which requires the code to be parallel in nature. A second source of speedup would be due to lower memory portion of the execution time. The traditional CPU, fabricated in process optimized for speed is fast as long as the program or segment of the program is compute intensive and does not have many cache misses, while the processor in memory, limited by the fabrication process can not be as fast. However, it will have much better memory performance. So if the program could be executed in such a way that compute intensive part remains in the processor and the memory intensive part is pushed to memory, then we could see speedup in overall execution time. This idea could be exploited by the combination of programming language, compiler and hardware, and is also part of future work.

Another feature to include to make FlexRAM a real product is to give an inexpensive solution to the address translation mechanism. The simulation in the thesis assumes that a very efficient virtual to physical address translation scheme exists for the P.Array [1]. In reality, there are many issues involved in the translation scheme, including context switch. This also has to be addressed in the future.

[1] an extreme of this would be to use physical address directly

Bibliography

[1] On the Complexity of Scheduling MIMD Operations for SIMD interpretation. *Journal of Parallel and Distributed Computing*, Vol 29, No. 1, Aug 1995.

[2] R. Agrawal, T. Imielinski, and A. Swami. Database Mining: A Perfromance perspective. *IEEE Trans. on Knowledge and Data Engineering*, 5(6), Dec 1993.

[3] 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.

[4] K. Batcher. Multidimension Access Memory in STARAN. *Transaction on Computer*, Feb 1977.

[5] K. E . Batcher. STARAN parallel processor system hardware. In *AFIPS conference proceedings*, volume 43, 1974.

[6] 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*, Jun 1997.

[7] P. Cabena, P Hadjinian, R. Stadler, J. Verhees, and A. Zanasi. *Discovering Data Mining: From Concept to Implementation.* Prentice-Hall, Inc., 1998.

[8] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle. Dynaimc Barrier Architecture for Multi-Mode Fine-Grain Parallelism Using Conventional Processors. In *Proceedings of the Int'l Conference on Parallel Processing*, 1994.

[9] D. Elliott, M. Stumm, and M. Snelgrove. Computational RAM: The case for SIMD computing in memory. In *ISCA Workshop on Mixing Logic and DRAM*, 1997.

[10] A. Brown et al. Using MML to Simulate Multiple Dual-Ported SRAMs: Parallel Routing Lookups in an AMT Switch Controller. In *ISCA Workshop on Mixing Logic and DRAM*, 1997.

[11] David Patterson et al. Intelligent RAM(IRAM): the Industrial Setting, Applications, and Architectures. In *International Conference on COmputer Design*, Oct 1997.

[12] John Granacki et al. http://www.dyncorp-is.com/darpa/meetings/dis98oct/.

[13] The National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/blast/.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[15] D. Hillis and G. Steele. *The Connection Machine.* MIT Press, Cambridge, MA, 1986.

[16] R. Michael Hord. *Parallel Supercomputing in SIMD Architectures.* CRC Press, 1990.

[17] IBM Corporation. http://www.chips.ibm.com/services/foundry/offerings/cmos/7ld. 1998.

[18] S. Kaxira, R. Sugumar, and J. Schwarzmeier. Distributed Vector Architecture: Beyond a Single Vector-IRAM. In *ISCA Workshop on Mixing Logic and DRAM*, 1997.

[19] K. Keeton, R Arpaci-Dusseau, and D.A.Patterson. IRAM and SmartSIMM: Overcoming the I/O Bus Bottleneck. In *ISCA Workshop on Mixing Logic and DRAM*, 1997.

[20] B. Kim, Y. H. Choi, and L. Kim. IRAM Design for Multimedia Applications. In *ISCA Workshop on Mixing Logic and DRAM*, 1997.

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

[22] T. Kohonen. *The Self-Organizing Map.* Springer, 1997.

[23] C. Kozyrakis and D. Patterson et al. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, Sep 1997.

[24] 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.

[25] 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.

[26] A. Lipman and W. Yang. The Smart Access Memory: An Intelligent RAM for Nearest Neighbor Database Searching. In *ISCA Workshop on Mixing Logic and DRAM*, 1997.

[27] P. Machanick, P. Salvedra, and L. Pompe. Hardware-Software Trade-Offs in a Direct Rambus Implementation of the RAMpage Memory Hierarchy. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–114, October 1998.

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

[29] M. T. O'Keefe and H. G. Dietz. Hardware Barrier Synchronization: Dynamic Barrier MIMD DBM. In *Proceedings of the Int'l Conference on Parallel Processing*, 1990.

[30] M. T. O'Keefe and H. G. Dietz. Hardware Barrier Synchronization: Dynamic Static MIMD SBM. In *Proceedings of the Int'l Conference on Parallel Processing*, 1990.

[31] 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.

[32] I. Rigoutsos and A Floratos. Combinatorial Motif Discovery In Biological Sequences Using the TEIRESIAS Algorithm. Technical report, IBM T. J. Watson Research Center, 1997.

[33] S.Przybylski. Embedded DRAMs: Today and Toward System-Level Integration. Half-day seminar, May 1997.

[34] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[35] T. Yamauchi, L Hammond, and K Olukotun. A Single Chip Multiprocessor Integrated with DRAM. In *ISCA Workshop on Mixing Logic and DRAM*, 1997.

[36] J. Yoo, C. Kim, and K. Lee et al. A 32-Bank 1Gb DRAM with 1GB/s Bandwidth. In *Proceedings of Int'l Solid-State Circuit Conference*, 1996.