Appears in the 2000 International Conference on Computer Design, Austin.

Reducing Cost and Tolerating Defects in Page-based Intelligent Memory

Mark Oskin, Diana Keen, Justin Hensley†, Lucian-Vlad Lita‡ and Frederic T. Chong
Department of Computer Science
University of California at Davis

Abstract

Active Pages is a page-based model of intelligent memory specifically designed to support virtualized hardware resources. Previous work has shown substantial performance benefits from off-loading data-intensive tasks to a memory system that implements Active Pages. With a simple VLIW processor embedded near each page on DRAM, Active Page memory systems achieve up to 1000X speedups over conventional memory systems [1].

In this study, we examine Active Page memories that share, or multiplex, embedded VLIW processors across multiple physical Active Pages. We explore the trade-off between individual page-processor performance and page-level multiplexing. We find that hardware costs of computational logic can be reduced from 31% of DRAM chip area to 12%, through multiplexing, without significant loss in performance. Furthermore, manufacturing defects that disable up to 50% of the page processors can be tolerated through efficient resource allocation and associative multiplexing.

1 Introduction

Accessing and manipulating data has become increasingly expensive as the gap between microprocessor and memory system performance has widened. Rapid advances in DRAM density have led to several proposals to move computational logic into the memory system [2] [3] [4]. We focus on Active Pages [5], a page-based model of computation which associates simple functions with each page of memory. For example, an Active Page may contain pixels of an image and support functions to transform and filter those pixels. The key feature of Active Pages is that they are designed to be implemented in commodity memory technology and provide inexpensive resources to exploit large amounts of parallelism. The more memory involved, the more Active Pages that are available to speed up the computation. In fact, our initial studies have shown from 10 to 1000X speedups over a conventional uniprocessor system running data-intensive applications such as sparse-matrix multiplication, MPEG encoding, protein sequence matching, and unstructured database searching.

Our initial work described an implementation of Active Pages which integrated a block of reconfigurable logic with each subarray in a DRAM chip. Although this approach was shown to be extremely promising, the reconfigurable logic was expected to consume 50% of the total available chip area. To reduce this logic area requirement, a VLIW processor was designed to replace the reconfigurable logic [1]. Active Page memory using this VLIW processor requires only 31% of the chip area for computational logic. Further area reduction, however, is desirable to make Active Page memories viable in the commodity market.

In this paper, we assume a multitasking environment. As such we expect some applications that utilize Active Page memory and others that are purely conventional in nature. Hence we do not expect full utilization of the available computational logic resources. For this study, we capitalize on this fact to demonstrate that further reductions in computational logic area are possible with only minimal performance impact.

We propose virtualizing the computational logic across individual Active Pages within the chip. Virtualization is accomplished by time-slicing a VLIW processor across one to eight Active Pages. We call this time-slicing *multiplexing* of the computational logic. This paper presents an analysis of multiplexing and its effects on performance in a multi-process environment. In addition, we look at how varying page-processor VLIW instruction width affects performance. By combining these approaches, we demonstrate that multiplexing is a more effective technique for reducing logic-area requirements than reducing individual page processor performance.

We conclude this work by focusing on another major factor in cost: manufacturing defects. DRAM architectures use redundant cells to tolerate defects, dramatically increasing chip yields and reducing cost. Embedded processors, however, do not have an analogous unit of redundancy. While multiplexing several Active Pages with one embedded processor reduces chip area, multiplexing each group of pages with two processors allows each group to tolerate a processor failure. This *associativity* requires some additional interconnect, but tolerance to randomly distributed processor failures increases from 33% to over 50% in our workloads.

In the next section, we briefly describe Active Page memory operation and how it differs from a conventional memory system. Section 3 describes the hardware implementation of Active Pages and how virtualization is accomplished. In Section 4, we describe the simulation environments and methodology used in this study. Section 5 presents our results. In Section 6, we examine related work. Finally, Section 7 describes future work, while Section 8 presents our conclusions.

2 Background

The Active Page project builds upon several ground-breaking studies on intelligent memory. In particular, the Berkeley IRAM project has demonstrated many of the benefits of integrating processors with memory in upcoming DRAM technologies [2] [6]. However, Active Page architectures have three key features which distinguish them from other intelligent memory proposals.

Acknowledgements: This work is supported in part by an NSF CAREER award to Fred Chong, by NSF grant CCR-9812415, by an NPSC fellowship to Diana Keen, by grants from Mitsubishi and Altera, and by grants from the UC Davis Academic Senate. More info at http://arch.cs.ucdavis.edu/AP

†Now at the University of North Carolina, ‡Now at Carnegie Mellon Univerisity

First, Active Pages leverage conventional page-based memory mechanisms to virtualize hardware for memory-based computation. Computations for each page can be suspended, restarted, and even swapped to disk. Computations for several pages can be multiplexed on a single embedded processing element. Furthermore, Active Pages use the same interface as conventional memory systems. Active Page data is modified with conventional memory reads and writes; Active Page functions are invoked through memory-mapped writes. Synchronization is accomplished through user-defined memory locations.

Second, Active Page memory systems are intended to enhance microprocessor performance in a conventional processor-memory architecture. This is in contrast to designs, such as IRAM [2], which focus on replacing conventional architectures with single-chip systems. While such systems have great potential for portable personal devices, memory requirements for desktop applications are likely to stay ahead of single-chip capacities.

Finally, Active Pages can exploit large amounts of parallelism. A memory system typically contains hundreds to thousands of pages of physical memory. Active Page systems can potentially support simultaneous computations at each of these pages. This page-based computation supports data parallelism similar to supercomputers of the past, but in a ubiquitous technology and for commodity applications.

The unconventional nature of Active Pages also implies a different model of computation. To use Active Pages, computation for an application must be divided, or *partitioned*, between processor and memory. For example, we use Active Page functions to gather operands for a sparse-matrix multiply and pass those operands on to the processor for multiplication. To perform such a computation, the matrix data and gathering functions must first be loaded into a memory system that supports Active Pages. The processor then, through a series of memory-mapped writes, starts the gather functions in the memory system. As the operands are gathered, the processor reads them from user-defined output areas in each page, multiplies them, and writes the results back to the array data structures in memory.

3 Active Pages

The focus of this paper is to evaluate the effect of multiplexing computational logic across Active Pages. The potential benefit to such an approach is reduced hardware cost. Previous work described an Active Page architecture that integrated reconfigurable logic with DRAM. The Semiconductor Industry Association (SIA) predicts that 1G-bit DRAMs will be produced in commodity by the year 2001 [7]. To reduce signal delay and power consumption, a 1G-bit DRAM is expected to be divided into 512K-byte subarrays [8]. Therefore, we assume each Active Page will consist of one of these subarrays. Using reconfigurable logic, we find that approximately 256K transistors are required per Active Page for the logic alone. This logic is the minimum amount required to execute basic Active Page functions [5]. Using a 512K-byte page size, we estimate such a design would require fully 50% of the available chip area for computational logic. All of the area figures in this paper assume a DRAM-only process technology. Merged-DRAM-Logic (MDL) technologies would reduce our numbers by a factor of two if they become available in commodity memories [9].

Further study has shown that instruction-level parallelism, not hardware specialization, is the key to our success with reconfigurable logic. Consequently, we have designed a simple VLIW embedded processor to exploit this parallelism without the hardware costs of reconfigurable circuitry [1]. The VLIW processor has support for addition, subtraction, shift, load, store, and branch instructions only (31 instructions total). This approach reduces Active Page cost to 31% of the area in a conventional gigabit DRAM. In the cost-driven commodity DRAM market, however, further cost reduction is desirable for widespread acceptance.

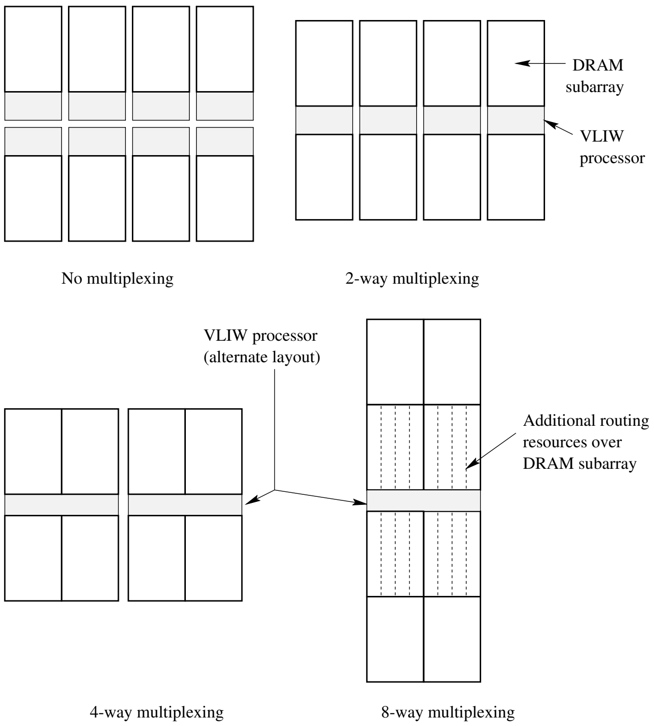

Figure 1: Multiplexing Active Page logic

We examine two approaches to reducing cost. The first is multiplexing, and the second is shortening the VLIW instruction width. The current VLIW design uses an instruction width of four. In Section 5 we explore the effects on performance of varying instruction width and degrees of multiplexing. We also examine a combination of these two techniques.

3.1 Time-sharing computational logic

Previous work [10] describes hardware mechanisms and operating system support necessary to implement virtual memory for Active Pages. This paper looks at virtualizing the page processors. To minimize logic area, several pages would share a single page processor. By multiplexing, or time-slicing, the page processor across Active Pages, the system gives the programmer the illusion of one page processor per Active Page.

Figure 1 illustrates how page processors can be multiplexed between pages. With four or fewer pages sharing one processor, the logic can be synthesized to adjoin each page, minimizing additional routing resources. For eight pages or more, additional routing is required. Since the page processors operate at only 100 MHz, we assume no performance penalty for multiplexing signals to each page processor. However, we conservatively assume up to a 20% logic area overhead for added wires, arbitration logic, and potential aspect ratio problems in synthesis.

The logic is then time-sliced to give each subarray a “vir-

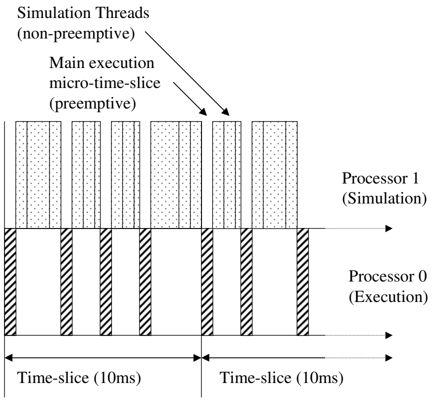

Figure 2: Emulator operation

tual" processor. To support multiplexing, the page processor must support context switching and protection mechanisms. In this study, we have assumed these mechanisms exist in hardware. If necessary, simplified task-switching handlers may be installed in each Active Page, thus providing some degree of software control over the process. Active Pages may be executing entirely different user processes, so protection must be provided. A small DRAM access hardware arbitrator and cache flushes between context switches are necessary.

Additionally, we investigate multiplexing with associativity for the purposes of defect tolerance. Specifically, we multiplex eight pages with two embedded processors instead of four to one. If a processor is defective due to a manufacturing error, it is still possible for the remaining page processor to provide Active Page functionality to the set of multiplexed Active Pages. Hardware support for associativity will incur area costs for interconnect, but our results will show that dramatic increases in defect tolerance will more than offset these costs.

Active Pages is a partitioned programming model. In such a programming model, components of the application execute within each Active Page, and a central processor thread exists to control and process data as needed for each page. A multi-tasking operating system would ordinarily time-slice the central processor. This leads naturally to exploring the possibility of time-slicing the Active Page memory.

4 Methodology

This study examines relative application performance in a multi-programmed environment. Previous single-process Active Page studies [5] used a cycle-level processor-memory simulator based upon the SimpleScalar toolset [11]. This simulator was extended with hooks to support a custom operating system called ActiveOS. Although ActiveOS was a functionally complete operating system that supported multiprocess workloads, simulation time limited applications and data-sets to small "proof of concept" sizes. While the results were promising for Active Page memory systems [10], simulation speed clearly needed improvement.

4.1 Memory OS Simulator (MOSS)

Parameter	Value
Processor	450mhz PentiumII
Memory	512MB 100mhz SDRAM
Active Page size	512K-bytes
Active Page processor speed	100mhz
Active Page processor width	Varied 1-8
Active Page multiplexing	Varied 1-8

Table 1: MOSS simulation parameters

For this study, we adopted an emulation approach to achieve the cycle times necessary to run large, multiprogrammed workloads. The emulator is a dual-processor Pentium II system and a customized Linux 2.2.0 kernel. The customized Linux kernel utilizes the dual-processor system to simulate a single-processor system with an Active Page memory. One processor (main) is utilized as the single-processor emulator, while the other (simulation) processor simulates the work done by the Active Page memory system. For simplicity, execution on the two processors is prevented from occurring simultaneously. In this manner, the cache and memory traffic of the main processor are not perturbed by the simulation processor.

The custom Linux kernel controls the hardware to synchronize these two processors and the system real-time clock. The system APIC (Advanced Programmable Interrupt Controller) is configured to interrupt the main processor every 100 microseconds. The interrupt triggers the Linux kernel to enter a simulation state. Once in the simulation state, the Linux kernel controls the second processor to simulate activity for each Active Page. The execution of the emulator is depicted in Figure 2.

Memory allocation was also optimized to scatter Active Pages uniformly across hardware to maximize multiplexing effectiveness and defect tolerance. The machine and simulation parameters are summarized in Table 1.

4.2 Applications

In this study, two workloads are used for evaluation: "engineering" and "commodity." Both workloads consist of a mixture of applications which use Active Page computation and conventional applications which do not. Note that when Active Page memory is used as conventional memory, clearly multiplexing is an advantageous mechanism to reduce hardware cost with little performance impact.

The engineering application suite contains array, sparse-matrix, and dynamic-programming (DNA sequencing) Active Page applications. The array application utilizes a C++ STL array class which implements parallel insert, delete and find in Active Pages. The application simulates a simple database and performs insert and delete operations. The sparse matrix application performs a number of matrix-vector multiplies, where Active Pages perform index comparison and gather-scatter matrix values. The dynamic-programming application uses the longest-common-subsequence algorithm to determine the optimum splicing of a number of string sequences, where Active Pages perform parallel dynamic programming on string matches. These applications run concurrently with three conventional applications which do not use Active Page computation: gcc, perl, and SimpleScalar.

The commodity workload contains three Active Page applications: array, 3D volume rendering, and an MPEG encoder. The array application is described above. The 3D vol-

Application	Memory usage	Page computation time (ms) 1-wide	2-wide	4-wide	8-wide
Array	78M	5.5	3.0	1.8	1.8
DNA Seq.	113M	6.8	3.9	2.8	2.4
Sparse-Matrix	251M	0.50	0.43	0.40	0.39
Volume Render	256M	140	77	58.4	58.4
MPEG Encode	8M	7400	4500	3900	4000
gcc	2.5M	-	-	-	-
perl	1.0M	-	-	-	-
SimpleScalar	24M	-	-	-	-
gzip	0.5M	-	-	-	-

Table 2: Application characteristics

ume rendering application is executing the back-projection algorithm. The MPEG encoding application reads in a number of frames and computes the optimum matrix pixel shift and temporal shift between the frames. The majority of the computations for both volume rendering and MPEG motion estimation are performed in Active Pages with high parallelism. The commodity suite uses three conventional applications: gcc, perl and gzip.

4.3 Evaluation Technique

In order to measure the performance of our workloads, each Active Page application was first written for the Active Page simulator based on the SimpleScalar toolset. The application was hand-partitioned between the main processor and Active Page memory system. Next, the Active Page component was constructed in C. After translation into assembly language using the gcc compiler with full optimizations enabled, the resulting assembly was further hand-optimized.

Hand-optimization of the assembly code included loop unrolling, software pipelining, and code scheduling. After producing an efficient conventional implementation, the code was then transformed into VLIW implementations for various VLIW-processor widths.

Once optimized Active Page code was constructed, the resulting application performance was measured using the detailed cycle-by-cycle SimpleScalar simulator. The resulting execution times are used to calibrate the MOSS-based application simulations. We summarize these calibrations and the memory usage of the applications in Table 2. Note that workload sizes were chosen to fit within the physical memory of our emulated system. Consequently, swapping pages to disk is not a factor in our study. Previous work evaluated the swapping characteristics of an Active Pages system [10].

Finally, each workload was executed using the MOSS simulator, and "steady-state" behavior was measured. For each workload, all six applications were executed concurrently. After each application finished, it was immediately restarted, thus maintaining a nearly constant execution of all six processes. The first execution of each application was discarded to remove any startup effects.

5 Results

In this section, we present our experimental results. We begin by examining performance in relation to a conventional memory system. Then we look at the effects of time-slicing 4-wide VLIWs across one to eight Active Pages. We investigate how varying the width of individual page processors affects application performance. We then vary the VLIW processor width while using multiplexing to maintain a fairly constant logic area requirement. What is interesting in this experiment is that the underlying trade-off is really between coarse-grained parallelism, in the form of individual Active Page function invocations, and fine-grained parallelism, in the form of conventional instruction level parallelism (ILP). A wide VLIW processor allows for a high degree of fine-grained parallelism, but since it is multiplexed across multiple pages, fewer pages may be active in the system simultaneously, leading to a lower degree of coarse-grained parallelism. Finally, we explore a defect tolerant Active Page memory configuration.

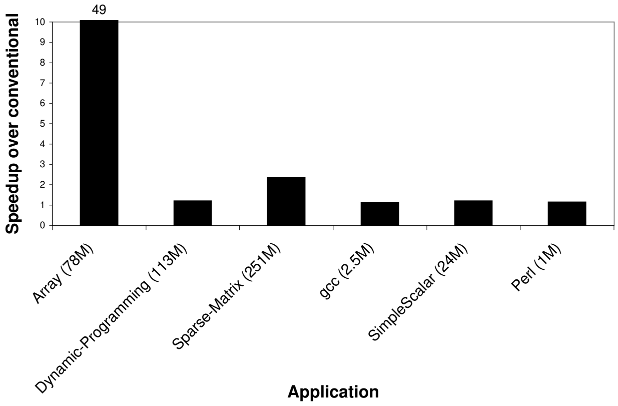

Figure 3: Speedup over conventional for the engineering workload

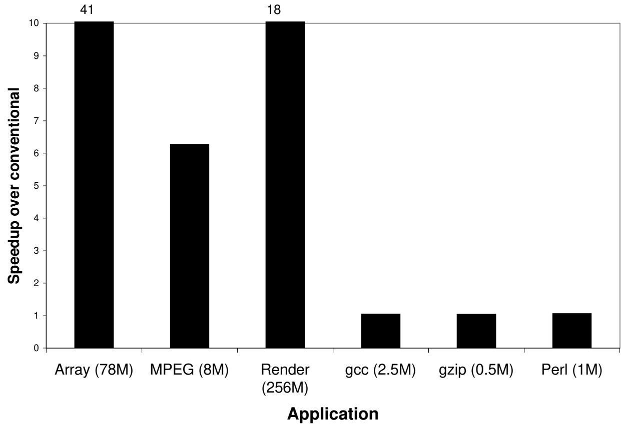

Figure 4: Speedup over conventional for the commodity workload

5.1 Speedup over Conventional

Figures 3 and 4 depict application speedup when applications use an Active Page memory system. Speedup is measured in terms of wall-time for the application in a conventional memory system divided by its wall-time using an Active Page memory system. We observe that Active Page applications continue to show substantial speedups when executed in a

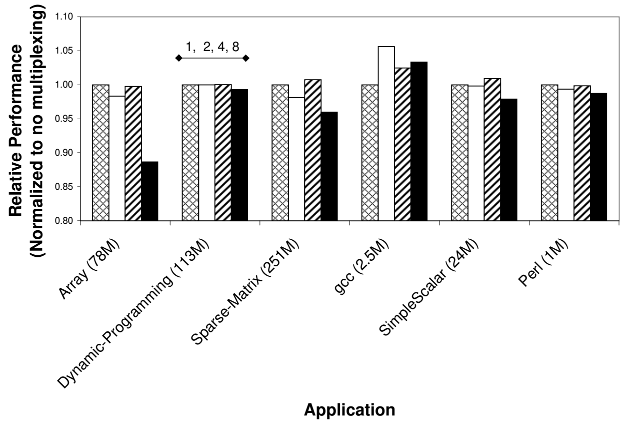

Figure 5: Performance versus hardware-multiplexing for the engineering workload

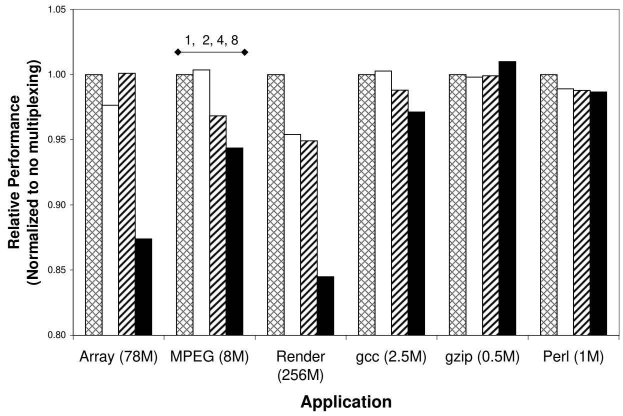

Figure 6: Performance versus hardware-multiplexing for the commodity workload

multiprocess environment. The current Active Page user library does not support exponential backoff for synchronization between processor and memory computations. It is expected that such functionality will free up main processor resources, and thus increase performance for conventional applications.

5.2 Multiplexing Performance

Figures 5 and 6 depict relative application performance as the degree of multiplexing is increased. We normalize our results to a configuration with no multiplexing, where a one-to-one relationship exists between 4-wide VLIW processing elements and DRAM subarrays. Multiplexing factors of two, four, and eight make up the remaining data points. We note that hardware multiplexing of eight incurs no more than a 17% performance penalty, and a multiplexing factor of four incurs no more than a 6% performance penalty for all Active Page applications studied in both workloads.

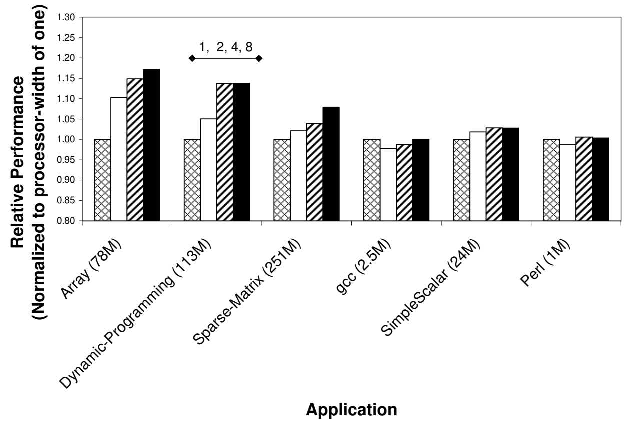

Figure 7: Performance versus Processor Width for the engineering workload

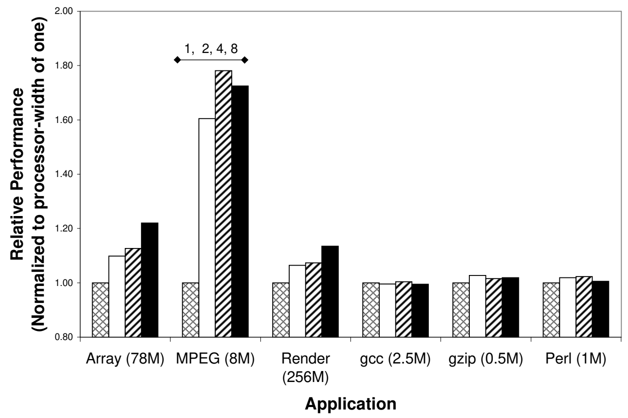

Figure 8: Performance versus Processor Width for the commodity workload

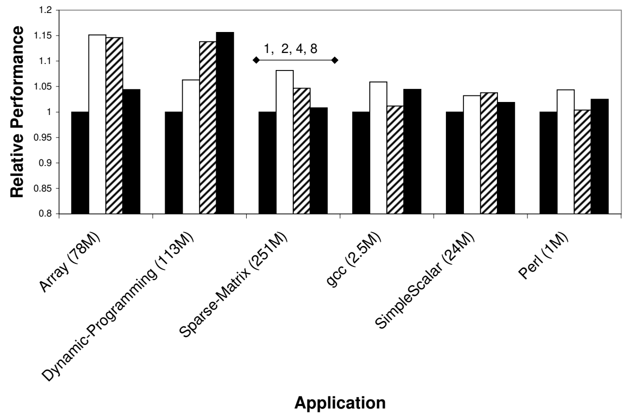

Figure 9: Performance of Multiplexing versus VLIW Processor Width

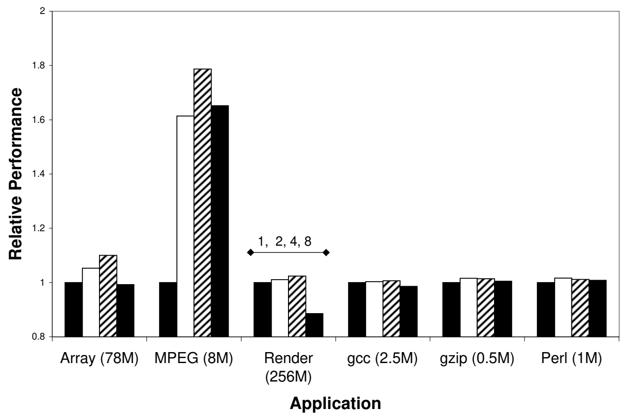

Figure 10: Performance of Multiplexing versus VLIW Processor Width

5.3 Processor Width Performance

Figures 7 and 8 depict relative application performance as VLIW processor width is varied. Here, processor widths of one, two, four and eight were evaluated. In general we observed Active Page application performance to increase up to 20% for the engineering workload, and between 20-80% for the commodity workload. It should be noted that MPEG suffers adverse cache effects with a VLIW width of eight, thus lowering performance relative to a four-wide VLIW. However, we expect a compressed code stream, or more compact VLIW instruction encoding would eliminate this decrease in performance, at the cost of an increased logic area requirement. In general, the largest performance gains due to VLIW processor width are achieved with processor widths of two and four, and not by eight.

5.4 Processor Width versus Multiplexing

Taking another look at Figures 7 and 8, we find that the Active Page applications do not have the static instruction level parallelism (ILP) to utilize much beyond a four-wide VLIW processor. In addition, Figures 5 and 6 show that the degradation due to multiplexing is superlinear, suggesting that too much coarse-grained parallelism exists within the application workloads to substantially multiplex the processor resources.

An experiment designed to compare these two forms of parallelism is depicted in Figures 9 and 10. Here we compare an Active Page device utilizing a single-issue processor with no multiplexing against a device utilizing a two-wide VLIW with two-way multiplexing, a four-wide VLIW with four-way multiplexing, and an eight-wide VLIW with eight-way multiplexing.

In the Active Page applications, a two-wide VLIW with two-way multiplexing shows a performance gain. This implies that the gain from the increased instruction-level parallelism outweighs the reduced coarse-grained parallelism. Because several conventional applications are active in the workloads, this makes sense, as many of the pages do not need the page processors. For the commodity workload, a four-wide VLIW with four-way multiplexing is optimal. No clear configuration appears optimal within the engineering workload but a four-wide VLIW processor with four-way multiplexing is a suitable compromise across the applications studied. Hence, we will use this configuration in the remainder of our study.

To describe why multiplexing performs well in a multiprocess environment, we identify three key factors:

- *Non-active memory* helps mask the performance degradation due to multiplexing. By definition, all pages of memory in a conventional application require no computation in memory. Some pages in an Active Page application also require no memory computation.

- *Active Page processing time* is the amount of time spent by the page-processor executing without main processor intervention. This time varies with page processor performance and is listed in Table 2. For most engineering applications, we find Active Page processing time to be fairly low. This is consistent with a tightly-coupled application in which the central processor is necessary to complete part of a complex computation. For commodity applications, simple data manipulations are easily off-loaded to the memory system. This leads to longer per-page computation times, most notably MPEG, with Active Page processing time on the order of seconds.

  The combination of low Active Page processing times and context switching in the main processor hides the effects of multiplexing in the memory system. In the absence of multiplexed Active Pages, when the main processor switches to another process, the Active Pages associated with the previous process quickly finish their work and stall until the process regains control of the central processor. Multiplexing allows efficient utilization of page processors by context switching them to another Active Page process when they would otherwise be idle.

  In an environment with Active Page processing times larger than a central processor time-slice, such as those observed in MPEG, one would expect multiplexing to degrade performance. Within this study, however, degradation is minimal due to the relatively low memory requirements of MPEG and the effects of non-Active memory.

- *Partitioning* is the process of dividing an application into work done in Active Pages and work done in the main processor. As long as the main processor can keep up with the Active Pages, an application is *scalable* and will exhibit linear speedup as its dataset grows and more Active Pages are used. Once the main processor becomes *saturated* with work, however, performance will no longer increase as more Active Pages are used.

  We find that multiprocess environments change the position at which an application transitions from scalable to saturated. Multiprocessing time-slices the central processor, which may be viewed as artificially slowing down the central processor from the perspective of a single process. This will shift the scalable-saturated point towards smaller problem sizes.

  We may utilize multiplexing to reverse this shift. Multiplexing essentially slows down the Active Page computation, shifting the scalable-saturated point back towards larger problem sizes.

Because of these properties of multi-programming environments, we observe that multiplexing is an efficient mechanism for reducing logic area requirements in an Active Page

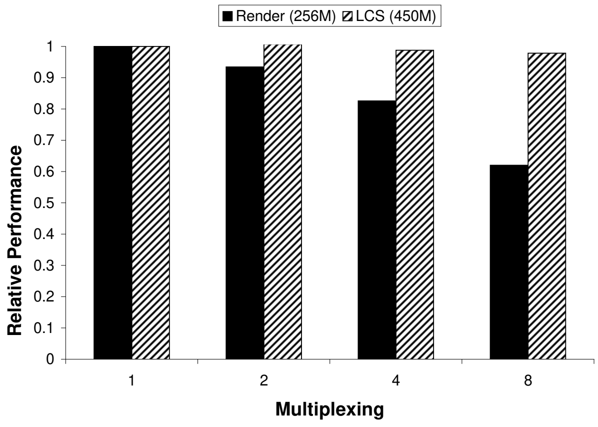

Figure 11: Performance of the rendering application versus hardware-multiplexing (single process)

memory device. A four-way multiplexed four-wide VLIW Active Page device is estimated to require 12% of the available chip-area for computational logic while still providing substantial performance gains. This estimate is based upon the reduced logic area coupled with a 20% logic area increase due to additional interconnect requirements.

Note, however, our success with multiplexing stems from a multiprogrammed environment in which the pages are spread throughout the system and each page processor services pages from separate processes. In a single process environment, the performance of multiplexing is dependent on application characteristics. We illustrate this with two applications, Rendering and Dynamic-Programming. Three-dimensional rendering has large processing times and needs all of the computational units at the same time. Dynamic-Programming uses only a fraction of the Active Pages at the same time as the computation wave-front passes through the DRAM. Their single process performances are depicted in Figure 11. Relative performance is depicted for one-way, two-way, four-way, and eight-way multiplexing. When the only application in the system is three-dimensional rendering, we see that eight-way multiplexing decreases application performance by 40%. Recall that in the multiprocess environment, it only exhibited a degradation of 17%. Dynamic-Programming, on the other hand, sees little degradation because it can share the page processors between its own Active Pages that are active and inactive at any given time.

5.5 Defect Tolerance

In this section, we use associativity to increase the defect tolerance of our system. We focus upon manufacturing defects that render our embedded processors inoperative. Our goal is to provide some degree of processor redundancy under the assumption that memory cells already have their own redundancy techniques.

Instead of four Active Pages sharing one four-wide VLIW processor, we allow eight pages to share two processors. We study the effect of randomly distributed processor failures upon this associative system. If a group suffers two failures, the operating system will only map conventional pages to that group (pages with no computation).

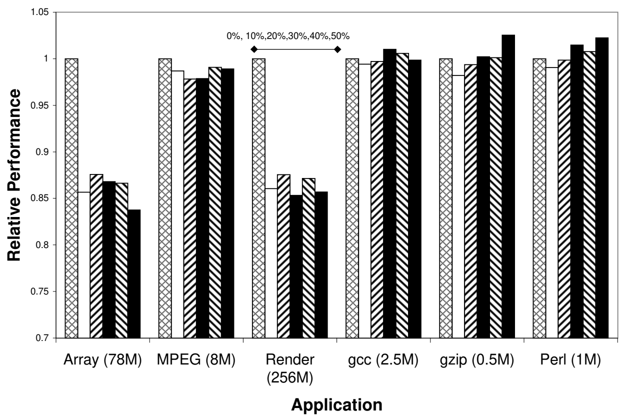

Figure 12: Performance versus random processor failures for the commodity workload

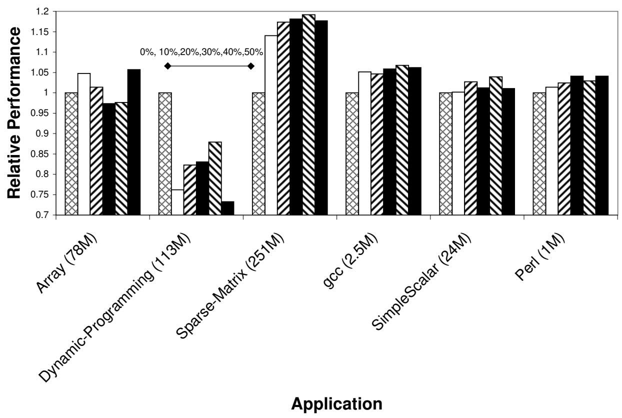

Figure 13: Performance versus random processor failures for the engineering workload

For our commodity application workload, the performance degradation due to randomly distributed processor failures is depicted in Figure 12. We note that up to a 50% defect rate is tolerated. Increasing the defect rate to 60% decreased the number of functional Active Pages below that required by the workload without page swapping. Virtualizing Active Pages to disk was studied in [10], and a similar mechanism can be utilized to further increase defect tolerance.

Similar results are shown in Figure 13 for our scientific workload. We note that defects are again tolerated up to 50%. We observe that performance of the sparse-matrix application *increases* as more embedded processors fail. This application is limited by the performance of the central processor. Its performance increases because some of the other applications are not using as much central processor time. The array and dynamic programming applications are limited by Active Page speed. As the percentage of failures increases, their Active Pages slow down and the central processor has fewer Active Page results to process. This frees up cycles on the central processor and increases performance on sparse matrix. Unfortunately, this increase came at the expense of the other two applications, and overall performance is decreasing as resources fail.

For both workloads, associativity creates an increased tolerance to failures. The benefits are straightforward. Two processors must fail instead of one in order to disable any Active Pages. If 50% of embedded processors fail in our test system, we see that with two-way associativity, up to 75% of the memory will still be available for Active Page use.

Since not all of the system memory is required to be "active" at the same time, the operating system may map around defect areas and utilize fully defective functional groups for conventional applications. Furthermore, the workloads do not require the full 512MB available to the system, and the unutilized memory is available to map into defective regions. The operating system can tolerate some defects without associativity by taking advantage of underutilization and conventional applications.

5.6 Summary

By multiplexing four-wide VLIW processors with four Active Pages each, we achieve substantial area savings with little degradation in performance. Although the area of the logic is reduced by four times, the area necessary for DRAM remains constant. The percentage of computational logic in an Active Page DRAM device with no multiplexing utilizing a four-wide VLIW processor is 31%; The computational logic percentage in a similar chip with four-way multiplexing is 12%. Overall this yields a smaller device with lower fabrication costs. We further find that sharing two page processors between eight pages allows up to a 50% defect rate to be tolerated by our application workloads with only a 20% performance penalty. This is quite encouraging, and we expect operating system management of on-chip manufacturing defects to be a viable method to lower overall system cost.

6 Related Work

DRAM densities have made intelligent memory attractive as commodity components. Intelligent memory, however, was proposed well before the current commodity thrust. The SWIM project [12] combined reconfigurable logic and memory to perform fast protocol computations. The J-Machine integrated processor, memory, and network router in a single chip to form building blocks for a fine-grained multiprocessor [13]. The RAW [14], MORPH [15], and RaPiD [16] projects continue to explore the use of reconfigurable technology to exploit parallelism. The RAW project, in particular, has also examined issues of processor width, dynamically trading off ILP and speculation [17]. The HPAM project [18] takes a hierarchical approach to intelligent memory.

FlexRAM[19] is an intelligent memory project with similar design goals to Active Pages. While each individual page processor is weaker, they provide a strong central processor on each chip which facilitates on-chip communication as well as general computation. Work related to FlexRAM also evaluated the limited use of intelligent memory in DSMs [20]

The DIVA[21] intelligent memory project aims more at high-performance computing than at commodity desktops. To this end, they provide a high bandwidth inter-chip network, as well as global addressing and messaging capabilities.

The Impulse project [22] has similar goals to Active Pages but focuses on adding address manipulation functions to the memory controller. Their applications, such as gather-scatter for sparse matrix multiplied by dense vector, are also enhanced by more efficiently feeding the microprocessor with data. All of our applications, however, require some small computations which can not be supported without more generalized computation in the memory system than provided by Impulse.

In [23] a scheme is proposed to permit user-level control of physical memory management. Future exploration of Active Page memory allocation strategies would clearly benefit from more application-guided placement and paging control.

Intelligent disks [24] [25] have also been proposed. These systems are meant for streaming applications that do not fit in main memory. After being processed by an intelligent disk, data may still benefit from computations in an intelligent memory. A unified view of intelligence at all levels of the memory hierarchy may eventually prove useful.

7 Future Work

For Active Pages to become a successful commodity architecture, the application partitioning process must be automated. Current work uses hand-coded libraries which can be called from conventional code. Ideally, a compiler would take high-level source code and divide the computation into processor code and Active Page functions, optimizing for memory bandwidth, synchronization, and parallelism to reduce execution time. This partitioning problem is very similar to that encountered in hardware-software co-design systems [26] which must divide code into pieces which run on general purpose processors and pieces which are implemented by ASICs (Application-Specific Integrated Circuits). These systems estimate the performance of each line of code on alternative technologies, account for communication between components, and use integer programming or simulated annealing to minimize execution time and cost. Active Pages could use a similar approach, but would also need to borrow from parallelizing compiler technology [27] to produce data layouts and schedule computation within the memory system.

This study does not explore the implications of interpage communication. The current method of handling such communication requires that the main processor move the data between memory pages. Such operations are expensive and form the dominating component of computation time in communication-intensive applications such as dynamic programming. However, this method simplifies protection and security. The operating system can filter the transactions through the central processor, performing virtual address translation and protection checks. The drawback to such an approach is that data must travel off-chip, through the central-processor caches, and back out again to the appropriate DRAM device. Future work will explore adding various inter-page communication mechanisms on each Active-Page chip. The performance benefits, however, must be balanced with a weaker protection model and more radical changes to commodity DRAM designs.

The MOSS emulation environment enables more advanced application studies for Active Pages. Furthermore, MOSS' Linux foundation opens the door for exploration of Active Page functionality for operating-system-specific operations. We expect that performance gains are available by migrating common functions such as page management to an Active Page for processing, freeing the central processor for user computations.

Future work will explore using Active Pages for introspective computing. One such research area would be to use Active Pages to dynamically compact and reorder conventional data-structures in memory to more efficiently utilize the processor cache. In a multiprogramming environment, Active Pages could take advantage of the time the process is not scheduled to make changes to the data so that it does not disrupt computation of that process. Another area of research is to use Active Pages to profile memory accesses and intelligently prefetch data into the main processor cache. It is expected that intelligent algorithm-specific prefetching may further enhance processor prefetching performance over existing model-based prefetching schemes.

Finally, the effects of synchronization upon the data and instruction caches of the computational logic need to be studied. Current work is investigating options for efficient synchronization of processor caches and the Active Page memory system. All but the most basic synchronization protocols will require hardware mechanisms for operation. The cost of these mechanisms is not factored into this study.

8 Conclusion

Active Pages is a page-based computational model for intelligent memory which can lead to substantial performance benefits. This study has looked at a promising method of reducing the computational logic area requirements of an Active Page memory device. By multiplexing the computational logic across one to four Active Pages, hardware cost can be reduced with little performance impact in a multiprogrammed environment. Furthermore, we find that it is more important to have fewer, faster computational logic elements that are time-shared across pages than more abundant, slower ones available for direct computation at each page. With a four-wide VLIW processor multiplexed with every four Active Pages, computational logic area can be reduced to 12% of total chip area in a gigabit DRAM. We also find that multiplexing, associativity and clever operating system resource allocation can map around manufacturing defects with only a 20% performance penalty with 50% random logic failures. An Active-Page-aware operating system can be defect tolerant and allow a lower cost system to be developed by increasing manufacturing chip yield. These incremental costs make Active Pages an attractive memory-based computation model.

References

[1] M. Oskin, J. Hensley, D. Keen, F. T. Chong, M. Farrens, and A. Chopra, "Exploiting ilp in page-based intelligent memory," in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, November 1999.

[2] D. Patterson *et al.*, "The case for intelligent RAM: IRAM," *IEEE Micro*, April 1997.

[3] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: the Terasys massively parallel PIM array," *Computer*, vol. 28, pp. 23–31, Apr. 1995.

[4] K. Murakami, S. Shirakawa, and H. Miyajima, "Parallel processing RAM chip with 256Mb DRAM and quad processors," in *ISSCC Digest of Technical Papers*, 1997.

[5] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*, (Barcelona, Spain), 1998.

[6] R. Fromm *et al.*, "The energy efficiency of IRAM architectures," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, 1997.

[7] Semiconductor Industry Association, "The national technology roadmap for semiconductors." http://www.sematech.org/public/roadmap/, 1997.

[8] K. Itoh *et al.*, "Limitations and challenges of multigigabit DRAM chip design," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 5, pp. 624–634, 1997.

[9] M. Motomura *et al.*, "An embedded DRAM-FPGA chip with instantaneous logic reconfiguration," in *1997 Symposium on VLSI Circuits*, 1997.

[10] M. Oskin, T. Sherwood, and F. T. Chong, "ActiveOS: Virtualizing intelligent memory," in *ICCD*, October 1999.

[11] D. Burger and T. Austin, "The SimpleScalar tool set, v2.0," *Comp Arch News*, vol. 25, June 1997.

[12] A. Asthana, M. Cravatts, and P. Krzyzanowski, "Design of an active memory system for network applications," in *International Workshop on Memory Technology, Design and Testing*, pp. 58–63, IEEE Computer Society Press, 1994.

[13] M. Noakes, D. Wallach, and W. Dally, "The J-Machine multicomputer: An architectural evaluation," in *Proceedings of the 20th Annual International Symposium on Computer Architecture 1993*, (San Diego, CA), pp. 224–235, ACM, May 1993.

[14] W. Lee *et al.*, "Space-time scheduling of instruction-level parallelism on a Raw machine," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

[15] A. A. Chien and R. K. Gupta, "Morph: A system architecture for robust high performance using customization," in *Frontiers '96*, 1996.

[16] C. Ebeling *et al.*, "Mapping applications to the rapid configurable architecture," in *Symposium on FPGAs for Custom Computing Machines*, (Napa Valley, California), April 1997.

[17] W. Lee *et al.*, "Space-time scheduling of instruction-level parallelism on a RAW machine," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

[18] Z. Miled, R. Eigenmann, J. Fortes, and V. Taylor, "Hierarchical processors-and-memory architecture for high performance computing," in *Sixth Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

[19] Y. Kang, M. Huang, S.-M. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas, "Flexram: Toward an advanced intelligent memory system," in *International Conference on Computer Design*, ICCD, October 1999.

[20] J. Torrellas, L. Yang, and A.-T. Nguyen, "Toward a cost-effective DSM organization that exploits processor-memory integration," in *Sixth International Symposium on High-Performance Computer Architecture (HPCA6)*, January 2000.

[21] M. W. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCross, J. Brockman, W. Athas, A. Srivasava, V. Freech, J. Shin, and J. Park, "Mapping irregular applications to diva, pim-based data-intensive architecture," in *International conference on Supercomputing*, SC, November 1999.

[22] J. Carter *et al.*, "Impulse: Building a smarter memory controller," in *HPCA*, January 1999.

[23] K. Harty and D. Cheriton, "Application-controlled physical memory using external page-cache management," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1993.

[24] A. Acharya, M. Uysal, and J. Saltz, "Active Disks: Programming model, algorithms and evaluation," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

[25] G. Gibson *et al.*, "A cost-effective high-bandwidth storage architecture," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

[26] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems.* Englewood Cliffs, New Jersey 07632: Prentice Hall, Inc, 1994.

[27] M. Hall *et al.*, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, December 1996.