Appears in the 1998 International Symposium on Computer Architecture, Barcelona.

Active Pages: A Computation Model for Intelligent Memory

Mark Oskin, Frederic T. Chong, and Timothy Sherwood
Department of Computer Science
University of California at Davis

Abstract

Microprocessors and memory systems suffer from a growing gap in performance. We introduce *Active Pages*, a computation model which addresses this gap by shifting data-intensive computations to the memory system. An Active Page consists of a page of data and a set of associated functions which can operate upon that data. We describe an implementation of Active Pages on RADram (Reconfigurable Architecture DRAM), a memory system based upon the integration of DRAM and reconfigurable logic. Results from the SimpleScalar simulator [BA97] demonstrate up to 1000X speedups on several applications using the RADram system versus conventional memory systems. We also explore the sensitivity of our results to implementations in other memory technologies.

1 Introduction

Microprocessor performance continues to follow phenomenal growth curves which drive the computing industry. Unfortunately, memory-system performance is falling behind. Processor-centric optimizations to bridge this *processor-memory gap* include prefetching, speculation, out-of-order execution, and multithreading [WM95]. Several of these approaches can lead to memory-bandwidth problems [BGK96]. We introduce *Active Pages*, a model of computation which *partitions* applications between a processor and an intelligent memory system. Our goal is to keep processors running at peak speeds by off-loading data manipulation to logic placed in the memory system.

Active Pages consist of a page of data and a set of associated functions that operate on that data. For example, an Active Page may contain an array data structure and a set of insert, delete, and find functions that operate on that array. A memory system that implements Active Pages is responsible for both the storage of the data and the computation of the associated functions.

Rapid advances in fabrication technology promise to make the integration of logic and memory practical. Although Active Pages can be implemented in a variety of architectures and technologies, we focus upon the integration of reconfigurable logic and DRAM. We introduce the RADram (Reconfigurable Architecture DRAM) system. On many applications, our simulations show substantial performance gains for a uniprocessor workstation using a RADram system versus a conventional memory system. RADram can also function as a conventional memory system with negligible performance degradation. As we shall see in Section 3, RADram is likely to have superior yield, higher parallelism, and better integration with commodity microprocessors when compared to architectures such as IRAM [Pat95]. Since memory technologies are a moving target, we measure the sensitivity of our results to the speed of Active Page implementations. This allows us to generalize to currently available technologies such as DRAM macrocells in ASIC (Application-Specific Integrated Circuit) technologies.

This paper starts with a description of Active Pages in Section 2, and continues with our RADram implementation in Section 3. We then describe our experimental methodology in Section 4 and our applications in Section 5. We continue with the reconfigurable logic designs for each application in Section 6. We present our results in Section 7 and generalize these results to other technologies in Section 8. Finally, we conclude with a discussion of related work in Section 9, future work in Section 10 and conclusions in Section 11.

2 Active Pages

Active Pages introduce new programming, system, and fabrication issues. In this section, we shall discuss programming issues which arise from the Active Page computational model. These issues are partitioning, coordination, computational scaling, and data manipulation. We will discuss system and fabrication issues in Section 3 where we introduce the RADram Active-Page implementation.

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 datastructures in memory.

Interface To simplify integration with commodity microprocessors and systems, the interface to Active Pages is designed to resemble a conventional virtual memory interface. Specifically, the Active Page interface includes the following:

- Standard memory interface functions: *write(vaddr, data)* and *read(vaddr)*
- A set of functions available for computation on a particular Active Page: *AP_functions*.
- An allocation function: *AP_alloc(group_id,vaddr)* which allocates an Active Page in group *group_id* at virtual address *vaddr*. Pages operating on the same data

Acknowledgements: Thanks to André DeHon, Matt Farrens, Lance Halstead, Tom Simon, Deborah Wallach, and our anonymous referees. This work is supported in part by an NSF CAREER award to Fred Chong, by Altera, and by grants from the UC Davis Academic Senate. More info at http://arch.cs.ucdavis.edu/RAD

will often belong to a *page group*, named by a *group_id*, in order to coordinate operations.

- A function binding procedure:
  AP_bind(group_id, AP_functions)
  which binds a set a set of functions *AP_functions* to a group *group_id* of Active Pages. This set of functions may be redefined through repeated calls to *AP_bind*. Since implementations may limit the number or complexity of functions associated with each page, re-binding may be necessary to make room for new functions by eliminating old ones.

- Additionally, applications will commonly use several variables in each Active Page as *synchronization variables* to coordinate between *AP_functions* and a processor. These variables require no additional support beyond reads and writes. Memory accesses by *AP_functions* and a processor are atomic.

Active-Page functions use virtual addresses and can reference any virtual address available to the allocating process. In our sparse-matrix example, the code begins by calling *AP_alloc* to allocate a group of pages to store the matrices to be multiplied. Then *AP_functions* are defined to include a function for index comparison and data gathering. Next, *AP_bind* is called to associate this function to the pages. To start the page computations, the processor activates the pages with an ordinary memory write to an application-defined location. The *AP_functions* poll such synchronization variables as soon as *AP_bind* is called. Once the functions have computed their results and gathered the matrix data to be multiplied, they write to another set of synchronization variables to indicate the data is ready. The process polls these variables and begins reading and multiplying the data once it is ready.

Our global virtual address space implies that some Active-Page functions may reference data in other pages. Such references are meant to be used sparingly and the implementation of inter-page memory references will be discussed in Section 3. Active Page implementations are intended to function in any system that uses a conventional memory system. For example, pages may coordinate with multiple processors in a Symmetric Multiprocessor, using Active-Page synchronization variables to enforce atomicity.

Partitioning In our sparse-matrix example, the application was partitioned between work done at the memory system and work done at the processor. Such partitioning varies in emphasis between efficient use of processor computation and efficient use of Active-Page computation. We refer to these two extremes as *processor-centric* and *memory-centric* partitioning. Processor-centric partitioning is appropriate for algorithms with complex computations, such as floating point. Memory-centric partitioning is appropriate for data manipulation and integer arithmetic.

Sparse-matrix computations require substantial floating-point computation and suggest a processor-centric partitioning. Active Pages compute which operands must be multiplied with the goal of providing the processor with enough operands to keep it running at peak speeds. Our image processing application, on the other hand, uses integer arithmetic and can be performed almost entirely in Active Pages. Consequently, the goal is to exploit parallelism and use as many Active Pages as possible.

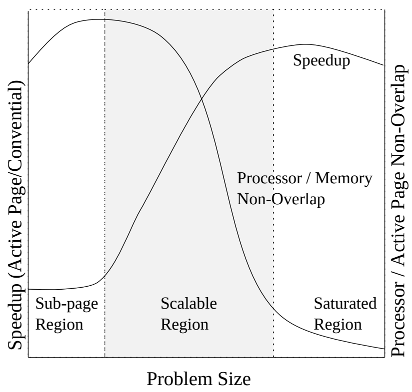

Figure 1: Expected computation scaling of Active Pages

Activation Time Intuitively, a processor working with a memory system that implements Active Pages is similar to a control processor working with a small data-parallel machine. Typically, an algorithm is partitioned by first dispatching a request for a computation to occur on the data within an Active Page. A well-structured application will have to move little, if any, additional data into the page in order for that function to complete. Thus, the majority of time in dispatching a work request is spent communicating to the Active Page the function to invoke and additional required parameters. We refer to the time it takes to dispatch this request as *activation time*. Activation time is generally constant for each page for a given function (measurements for each application will be given in Table 4).

Coordination Partitioning computations implies that Active Pages must coordinate with the processor and with each other. Processor-page coordination is accomplished via predefined synchronization variables. Inter-page coordination is accomplished with inter-page memory references.

Synchronization variables are used to coordinate activities between the Active Page functions and the processor. The structure and layout of these variables are implementation and application specific. The variables may serve as locks to indicate when inputs or outputs for an Active Page operation are valid. This interface is similar to memory-mapped registers used for network interfaces.

The Active Page model of computation does not define an explicit means for inter-page communication. Support for communication between pages can be accomplished in a variety of fashions. Abstractly, all forms of communication are viewed as non-local memory references issued by an Active Page. For performance reasons, an Active Page memory system may choose to combine several references into a contiguous inter-page memory copy. Our RADram implementation (Section 3) simulates such an approach.

Computation Scaling The computational power of Active Pages scales in an unusual way as application problem sizes grow. In this section, we develop some intuition about this scaling and we will verify these intuitions in Section 7.

Traditional multiprocessors generally operate with a fixed number of processing engines which must be applied to a variable problem size. With Active Pages, the number of processing engines is coupled to physical memory size. Since many systems are designed to scale memory size to contain the data of their intended applications, more Active Pages will be available for the computation.

Figure 1 shows how we expect Active-Page performance to scale as problem size grows. *Speedup* refers to the performance of a system using a conventional memory system divided the performance of a system using Active Pages. *Non-Overlap Time* is the time the processor spends waiting for Active Page computation which is not overlapped with processor computation. This is indicative of the quality of partitioning. As illustrated in Figure 1, we expect three regions of speedup as problem sizes scale:

The sub-page region: For very small problem sizes, applications use a small number of Active Pages and utilization of those pages is poor. Activation time dominates the computation and speedups do not scale until the Active Page function offloads sufficient work from the processor.

The scalable region: Once the problem is larger, the number of Active Pages involved increases linearly. The corresponding increase in computational power results in linear speed-ups.

The saturated region: Although the number of Active Pages grows with data size, the number of processors in a system does not. Consequently, we expect speedups to eventually level off as the processor-component of the application saturates constant processor resources. This leveling off can also produce a degradation in performance as an increased number of Active Pages can increase the synchronization and communication overhead.

Ideally, we want speedups which are in the rightmost portion of the scalable region. Fortunately, partitions can be tuned to shift this scalable region towards specific problem sizes.

Data Manipulation In addition to providing scalable computation, Active Pages allow programmers to optimize for density and indexing rather than data manipulation. Currently, programmers have a wealth of data structures they can choose to use for any given problem. However, these data structures each have advantages and disadvantages. For instance, doubly-linked lists provide fast insertion and deletion of elements, but poor random access. On the other hand, arrays provide fast random access, but poor performance on insertions and deletions.

To some extent, Active Pages remove the burden of compromise when choosing a data structure. For example, our implementation of the STL array class uses dense arrays, but exploits Active Page functions to provide fast insertion and deletion.

3 Implementation: RADram

In this section, we describe the Reconfigurable Architecture DRAM (RADram) system, shown in Figure 2. RADram is an architecture based upon the integration of the next generation of FPGA (Field-Programmable Gate Array) and DRAM technology. To minimize latency and reduce power consumption, large DRAMs are divided into subarrays, each with its own subset of address bits and decoders [I+97]. RADram exploits this structure by associating a block of reconfigurable logic with each subarray.

RADram Architecture For gigabit DRAMs, a good sub-array size to minimize power and latency is 512 Kbytes [I+97]. The RADram system associates 256 LEs (Logic Elements, a standard block of logic in FPGAs which is based upon a 4-element Look Up Table or 4-LUT) to each of these sub-arrays. This allows efficient support for Active-Page sizes which are multiples of 512 kbytes.

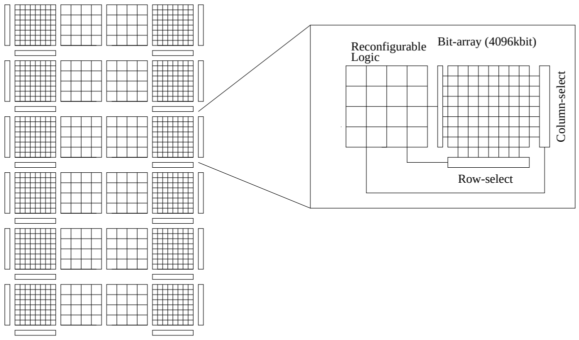

Figure 2: The RADram System

Parameter	Reference	Variation
CPU Clock	1 GHz	–
L1 I-Cache	64K	–
L1 D-Cache	64K	32K–256K
L2 Cache	1M	256K–4M
Reconf Logic	100 MHz	10–500 MHz
Cache Miss	50 ns	0–600 ns

Table 1: Summary of RADram parameters

Each LE requires about 1K transistors of area on a logic chip. The Semiconductor Industry Association (SIA) roadmap [Sem94] projects mass production of 1-gigabit DRAM chips by the year 2001. If we devote half of the area of such a chip to logic, we expect the DRAM process to support approximately 32M transistors, which is enough to provide 256 LEs to each 512K sub-array of the remaining 0.5-gigabits of memory on the chip. DeHon [DeH96b] gives several estimates of FPGA area.

We adopt a *processor-mediated* approach to inter-page communication which assumes infrequent communication. When an Active-Page function reaches a memory reference that can not be satisfied by its local page, it blocks and raises a processor interrupt. The processor satisfies the request by reading and writing to the appropriate pages. Once an interrupt is raised, the processor generally satisfies many requests from different pages in the system. Future work will evaluate hardware mechanisms for in-chip communication, increasing the number of outstanding references per page, and processor-polling for requests. The processor-mediated methodology, however, functions well for our applications and will greatly simplify future work in paging and virtual memory.

Table 1 lists the parameters of our reference RADram implementation. Several parameters were also individually varied in our experiments with respect to the reference implementation. The range of variation for these parameters is also given in Table 1. Additionally, a memory bus capable of transferring 32 bits of data between memory and cache every 10 ns is assumed.

Why Reconfigurable Logic? The potential of gigabit densities in DRAM has prompted research and development in a variety of implementation options for intelligent memory. IRAM [Pat95], an integration of processor core and DRAM, is a well-known option studied at Berkeley. RADram, however, is likely to have better yield, higher parallelism, and better integration with commodity processors than IRAM.

The primary advantage of RADram memory devices is that they will be inexpensive to fabricate. Processor chips cost ten times as much as memory chips because their complexity makes their yield, or percentage of working chips, much lower [Prz97]. DRAMs are fabricated with redundant memory cells that can replace defective cells through laser modification after chip production. The uniform nature of reconfigurable logic allows for similar measures in RADram chips. In contrast, IRAM chip designers will have to work hard to avoid yields similar to processor chips. If IRAM chips are fabricated at processor costs, systems will be limited to a few IRAM chips and to applications with smaller data. RADram is intended to fabricate at DRAM costs, which allows dozens of chips per system and much larger application data.

Our results will show that RADram can exploit extremely high parallelism by supporting simple, application-specific operations in memory. A multi-gigabit RADram can have more than 128 Active Pages, each of which can execute simultaneously. Processor-in-DRAM solutions can not support such high parallelism. The variety of custom operations used in our applications also suggests that fixed logic would severely limit the functionality of Active Page applications.

Finally, RADram is specifically designed to support commodity microprocessors. The RADram interface is compatible with standard memory busses. A primary goal of RADram is to supply microprocessors with enough data to keep them running at peak speeds. IRAM technology, however, is intended to compete with commodity processors. This competition may eventually be favorable for IRAM as the importance of single-chip systems increases, but ever-growing applications may always demand larger memories and multiple chips.

Fabrication Interest in the fabrication of Merged DRAM Logic (MDL) devices has grown dramatically in the past few years. Major manufacturers currently have the capability to fabricate DRAM cells (macrocells) in logic chips. Processors have also been fabricated in DRAM chips. Current DRAM in logic chips has poor density. Logic in DRAM chips has poor speed and density. Merged DRAM-logic processes, which can fabricate both kinds of structures well, are becoming available [Prz97]. Our study, however, is conservative and assumes a DRAM process with associated penalties in logic speed and density.

Power Power consumption is a major concern for DRAM chips because increased chip temperatures result in higher charge leakage from storage cells. This leakage increases the need for more frequent DRAM refresh. Fortunately, this higher refresh can be bundled into our logic added to each DRAM subarray.

Although a detailed study of power is beyond the scope of this paper, we have been conservative in our use of power in RADram. Our applications only use 32 bits of bandwidth between data and logic in RADram pages. This could easily be increased to 256 or 512 bits, but would result in higher power consumption. Increasing bandwidth would also require more reconfigurable logic, which is beyond our area constraints for some applications. Application performance, however, is high despite conservative bandwidth.

4 Methodology

To evaluate Active Pages, we conducted a detailed application study. The reference Active-Page platform used for this study was previously described in Section 3. This platform was studied using a three step approach. First, a simulator was implemented which modeled the RADram Active-Page memory system. Second, a set of applications were chosen which represented various algorithmic domains. Finally, these applications were written and optimized for both the RADram and conventional memory system architectures.

As a base for a simulation environment we started with the SimpleScalar v2.0 tool set [BA97]. This tool set provides the mechanisms to compile, debug and simulate applications compiled to a RISC architecture. The SimpleScalar RISC architecture is loosely based upon the MIPS R3000 instruction set architecture. The SimpleScalar environment was extended by replacing the simulated conventional memory hierarchy with an Active-Page memory system. The new simulated memory hierarchy provides mechanisms which simulate RADram application-specific circuits executing within the DRAM memory system. Further, the SimpleScalar instruction set was extended with Intel MMX multi-media instruction opcodes. Finally, the toolset was enhanced by updating the GNU C/C++ compiler version included to the latest v2.7.2.1 compiler suite. All applications in this study were compiled with the *-O3* optimization option.

After implementation of this simulation environment, a set of applications was chosen for architectural evaluation. Each application is briefly described in Section 5. Here we explore the methodology used in choosing, partitioning and evaluating these applications.

Applications were chosen with three motives in mind. First, the algorithms to be implemented in the application were representative of a broad class of algorithms used in a range of applications. Second, the algorithm or application illustrated a certain kind of partitioning as described in Section 2. Finally, an MMX-instruction-set compatible application was chosen to explore Active-Page implementations other than RADram. For instance, future work may investigate the possibility of identifying a small key set of data manipulation primitives which should be implemented in fixed logic in the Active-Page model.

The first step in studying each application or algorithm described in Section 5 is to implement and optimize it on a conventional memory system. The application is then hand-partitioned for an Active-Page memory system. Next, Active-Page functions are coded in VHDL and synthesized to FPGA logic. The results of this are discussed in Section 6. State transition characteristics of these synthesized circuits is used to simulate the functions with our SimpleScalar simulator.

5 Applications

In order to demonstrate effective partitioning of applications between processor and Active Pages, we chose a range of applications representing both memory- and processor-centric partitioning. Table 2 summarizes the attributes of these applications. This section describes each application and divides those descriptions into each partitioning class.

Memory-Centric Applications			
Name	Application	Processor Computation	Active Page Computation
Array	C++ standard template library array class	C++ code using array class Cross-page moves	Array insert, delete, and find
Database	Address Database	Initiates queries Summarizes results	Searches unindexed data
Median	Median filter for images	Image I/O	Median of neighboring pixels
Dynamic Prog	Protein sequence matching	Backtracking	Compute MINs and fills table
Processor-Centric Applications			
Name	Application	Processor Computation	Active Page Computation
Matrix	Matrix multiply for Simplex and finite element	Floating point multiplies	Index comparison and gather/scatter of data
MPEG-MMX	MPEG decoder using MMX instructions	MMX dispatch Discrete cosine transform	MMX instructions

Table 2: Summary of partitioning of applications between processor and active pages

5.1 Memory-Centric Partitioning

As discussed in Section 2, Active Pages can exploit the parallelism in applications through memory-centric partitioning. Our array, database, median filtering, and dynamic programming applications are good examples of such partitioning.

STL Array Template The STL array template is a general purpose C++ template which permits the storage, access, and retrieval of objects based upon a linear integer index. The template class supports the usual array access operators, as well as insert, delete and binary-find/count operations. All of the applications implemented hide the layout of data and partitioning of algorithmic operations from the application via a simple C++ interface. However, the STL array best demonstrates this principle. Library calls, derived from a common subclass, allow single source files to work with either the Active-Page or conventional-system implementation of the array template.

The implementation uses reconfigurable logic to speedup the following operations: array insert, delete, and count operations. The insert and delete operations involve moving portions of the array in parallel to accommodate the change in array size. The count operation is implemented by a binary comparison circuit.

These three operations are indicative of a broad range of array operations which the RADram system can effectively compute. Further examples from the STL library include: accumulate, partial sum, random shuffle, rotate, and adjacent difference.

Database Query Several methods [SKS97] exist to speed up database searches, if the searches involve indexed fields. Indexing produces a second table within the database which permits the database engine to quickly locate fields in logarithmic or constant time. However, indexing is often not practical for highly-varied queries or under tight storage constraints. Unindexed queries can take time proportionally linear to the number of records. Our database benchmark uses a synthetically generated address book. Custom Active Page functions were written to search for exact matches on any of the string fields contained in the address records.

The RADram system time complexity of the unindexed database query is $\mathcal{O}(1)$, however the constant bounding it is quite large. The performance gained by the RADram system comes from the parallelism available in the database search. In theory, all records can be searched simultaneously. In practice, the records are grouped into blocks, which are roughly the size of a RADram memory page. These blocks are then distributed among the pages in the RADram memory system. Each page is then custom programmed with the search engine's application specific circuit. To demonstrate the performance of the RADram system on this application a count of exact matches for the last name of an individual in the address book is performed. The count is run on the same database in both the RADram system and on a conventional implementation.

Image Processing Image processing and signal processing have been traditional strengths of FPGA's and custom processor technologies [R+93] [AA95] [K+96]. We implemented an image median filtering [RW92] application on RADram. Median filtering is a non-linear method which reduces the noise contained in an image without blurring the high-frequency components of the image signal. The RADram implementation divides the image by row blocks among various Active Pages. Each row block contains two additional rows, one above the current row block, and one below it, in order to perform the median filtering kernel computation. The Active Pages are then programmed with a custom circuit designed to find the median of nine short integer values. For comparison, the conventional system uses a hand-coded algorithm which takes a minimal number of comparisons to find the median of nine values.

Because the computational work involved is small in terms of circuit area, the bulk of the median filtering application runs inside the RADram memory system. Not surprisingly, this application allows RADram to exploit high parallelism and memory bandwidth. RADram also uses a custom circuit which is designed for sorting nine short integer values. The conventional implementation requires several conditional instructions, as well as memory I/O operations, in order to find the median value.

Largest Common Subsequence This algorithm is representative of a broad class of string algorithms which form the basis for modern biological research. At the heart of the computer algorithm to reconstruct DNA sequences are string algorithms such as largest common subsequence, global alignment, and local alignment [Gus97]. The largest common subsequence (LCS) computation is typically done using a dynamic programming construction. This construction runs in $\mathcal{O}(n^2)$ time and space for sequences of length n. One can view the construction as a set of computations over a plane. For the LCS algorithm, the computation can proceed in parallel as a wave-front starting at the upper left corner and ending in the lower right corner of this plane. This wave-front computation runs in $\mathcal{O}(n\log(n))$ time on the RADram system.

The RADram system implements the LCS computation by dividing the algorithm into two steps. The first step is the computation of the LCS result matrix itself. The second step is the backtracking [CLR96] required to find the largest common subsequence. The RADram system executes the first step entirely within the reconfigurable logic inside the memory system. Backtracking executes entirely within the processor.

5.2 Processor-Centric Partitioning

Active Pages are intended for simple, application-specific operations, leaving more complex computations to general-purpose microprocessors. Our MMX and matrix applications are good examples of processor-centric partitioning.

MMX Primitives The MMX multimedia instruction primitives were chosen for implementation within the RADram system for two reasons. First, they represent a well known "commodity" set of architecture primitives. Second, they are simple primitive operations designed for parallel execution.

The simulator was extended to support SimpleScalar MMX instructions, and RADram MMX instruction equivalents. The MMX instructions themselves are highly parallel, simple, and generally complete in a single processor cycle. To improve upon the base SimpleScalar MMX instructions, the RADram equivalents operate on larger data widths. While an MMX instruction in SimpleScalar is restricted to producing only 32 bits of data per instruction, a RADram MMX instruction can produce up to 256 kbytes of data per instruction.

While implementation of the complete MMX instruction set is still underway, enough is implemented to carry out key portions of the MPEG encoding and decoding processes. While future work will explore more MPEG routines, current work has focused upon application of the correction matrices within the P and B frames [M$^+$96]. Future implementation of the MPEG algorithm will partition additional components between the processor and RADram memory system. The processor will be responsible for the Discrete Cosine Transform (DCT), while the RADram system will handle motion detection, application of motion correction matrices, run length encoding and decoding (RLE), and Huffman encoding and decoding.

Sparse-Matrix Multiply A wide range of real-world problems can be represented as sparse matrices. We examine both a common scientific benchmark and a more challenging compiler optimization problem. Our scientific benchmark involves the multiplication of matrices representing finite-element computations taken from the Harwell-Boeing benchmark suite [D$^+$92]. Our compiler optimization problem involves using the Simplex method [NM65] to perform optimal register allocation [GW96].

A key computation in both these applications is sparse vector-vector dot-product. Conventional implementations of this operation are severely limited by processor-memory bandwidth. Sparse vector FLOPS on a conventional system are often an order of magnitude lower than those for dense vectors. The processor must fetch the indices of each nonzero in both vectors of the dot product, determine which indices match, fetch the data corresponding to those indices, multiply the data, and write the data back to its appropriate location.

In contrast, the RADram system implements a *compare-gather-compute* approach. Active Page functions fetch and *compare* vector indices, fetch the data values for the indices that match, and *gather* the data into cache-line size blocks. Vectors are co-located on pages. The processor then reads the packed data, *computes* the multiplies, and writes back cache-line size blocks of results. Note that only "useful" data travels between the processor and memory, greatly conserving bandwidth. With large matrices, the RADram system has enough Active Pages executing to keep the processor computing at peak floating-point speeds.

6 Synthesized Logic

In order to estimate performance and area of RADram logic configurations, each function of an application's Active Pages was hand-coded in a high-level circuit-description language, VHDL [Ash90], and circuits synthesized to completely routed designs in contemporary FPGA technology. This provided a means to verify the timing of the simulated circuit implementation, as well as information on circuit area, which helped guide the RADram design.

The results of our implementations of the application specific circuits for the simulated applications are summarized in Table 3. These results were obtained by implementing the circuit design in behavioral VHDL and synthesizing them with the Synopsys FPGA design tools. After synthesis to a technology independent logic description, the designs were placed and routed to an Altera FLEX-10K10-3 part. This allowed us to study the post-routed designs on real FPGA technology. The count of logic block usage reported in Table 3 includes both completely used and partially used LEs. The speed and code size were directly reported by the Synopsys tools.

The results obtained from implementation of application-specific circuits indicate that the RADram Active-Page system can execute the application kernel's circuits. The RADram implementation can implement designs with approximately 256 LEs per Active Page, and all of our designs are below this amount. Our designs can also be further optimized by implementing common memory interfaces in fixed logic. Our system simulation assumes a 100 MHz clock for our circuits. Given modest advances in FPGA technology, this should be achievable for our circuits by 2001. Finally, the code size is an indication of the potential "code-bloat" which will happen when transitioning an application to the RADram system. Code size is also indicative of the page-replacement cost for Active Pages, which we anticipate to be 2-4 times larger than for conventional pages due to reconfiguration time. However, pages which do not use Active-Page functions do not incur this cost, and future reconfigurable technologies may significantly reduce this cost (see Section 10).

7 Results

In this section, we compare our RADram simulation results of each application kernel described in Section 5 to our expectations from the Active-Page application characteristics discussed in Section 2. First, we discuss performance of RADram versus a conventional memory system executing optimized versions of the same applications. Then we explore the memory hierarchy of both memory systems by studying the effects of cache parameters. Finally, we develop an analytical model to

Application	LEs	Speed	Code
Array-delete	109	29.0 ns	2.7 KB
Array-insert	115	26.2 ns	2.9 KB
Array-find	141	32.1 ns	3.5 KB
Database	142	35.4 ns	3.5 KB
Dynamic Prog	179	39.2 ns	4.5 KB
Matrix	205	45.3 ns	5.6 KB
MPEG-MMX	131	34.6 ns	3.3 KB

Table 3: Active-Page functions synthesized for RADram.

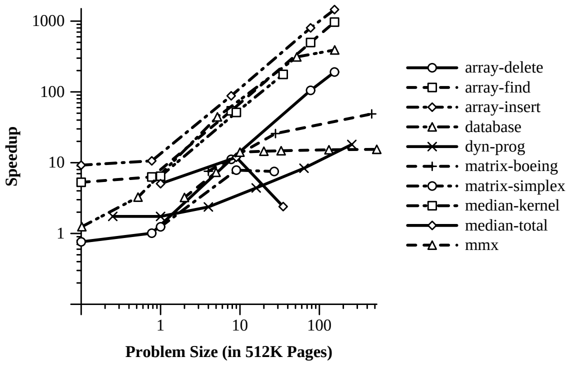

Figure 3: RADram speedup as problem size varies.

describe partitioned application performance, and then compute the correlation between this model and our experimental results.

7.1 Performance

To evaluate performance of the RADram Active Page memory system, each application described in Section 5 was executed on a range of problem sizes using a fixed set of machine characteristics listed in Table 1. The speedup of our applications running on a RADram memory system compared to a conventional memory system are shown in Figure 3. Each application was run on a range of problem sizes, given in terms of number of Active Pages (512 Kbyte superpages). We make two primary observations about this graph.

First, application kernels execute significantly faster on a RADram memory system than a conventional memory system. The one exception from our application mix is the array-delete primitive in the sub-page region. The SimpleScalar processor instruction set actually favors array-delete over array-insert. To take advantage of this fast delete, the RADram version of array-delete uses an adaptive algorithm that uses the processor more for arrays that are smaller than one Active Page.

Second, our performance results qualitatively scale as we expected in Figure 1. We observe that most applications show little growth in speedup as data size grows within the sub-page region (below one page for most applications). In this region, RADram applications have little parallelism to offset activation costs. As we leave this region, we enter the scalable region and see that performance on all of our applications grows nicely as data size increases. Four applications–database, mmx, matrix-simplex, matrix-boeing, and median-filtering–also reach the saturated region. Here, RADram performance is limited by the progress of the processor. This limitation may be due to either too much work for a given

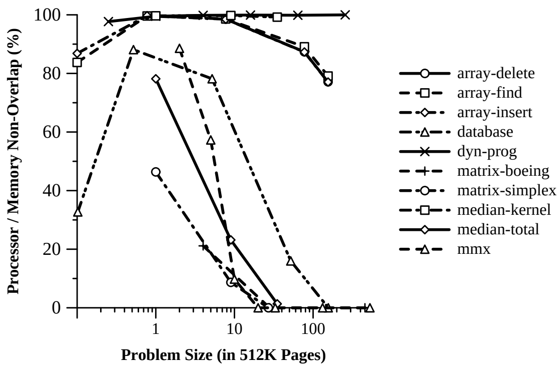

Figure 4: Percent cycles the processor is stalled on RADram as problem size varies.

speed processor or too much data traveling between the processor and RADram across the memory bus. Performance can actually decrease as coordination costs dominate performance. Given a large enough problem size, all our applications would eventually reach the saturated region.

7.2 Processor-Memory Non-overlap

The saturated region of Active-Page performance emphasizes the importance of partitioning applications to efficiently use the processor in a system. For processor-centric applications, this dependence is obvious. The goal is to keep the processor computing by providing a steady stream of useful data from the memory system. For memory-centric partitions, however, the processor is still a vital resource. Active Pages can not compute without activation and inter-page communication, both provided by the processor.

As data size grows in an Active-Page application, so does the load upon the processor. We measure the remaining capacity of a processor to handle this load with a metric we call *processor-memory non-overlap* time. Non-overlap is the time the processor spends waiting for the memory system and can be used to estimate the boundary between the scalable and saturated regions of application performance.

The relative percentage of time the processor is stalled, waiting for memory system computation is shown in Figure 4. As described earlier in Section 7.1, the applications which reached the saturated region of speedup were: database, matrix-simplex, matrix-boeing, and median-filtering. As is shown in Figure 4 these applications also reach a point of complete processor-memory overlap. The effect of this is described in Section 2.

We also observe that for the array primitives and the dynamic programming application the non-overlap percentage remains relatively high. These applications are largely memory-centric, with very little processor activity. In fact, the array primitives operate asynchronously to the end of the application, and are artificially forced in synchronous operation for this study. This means that an application can use the insert an delete array primitives with only the cost of RADram function invocation. Modulo dependencies on the array, the time spent by the memory system shifting data can be overlapped with operations outside of the STL array class. This overlap occurs in a natural way with no additional effort required by the programmer who uses the RADram STL array class.

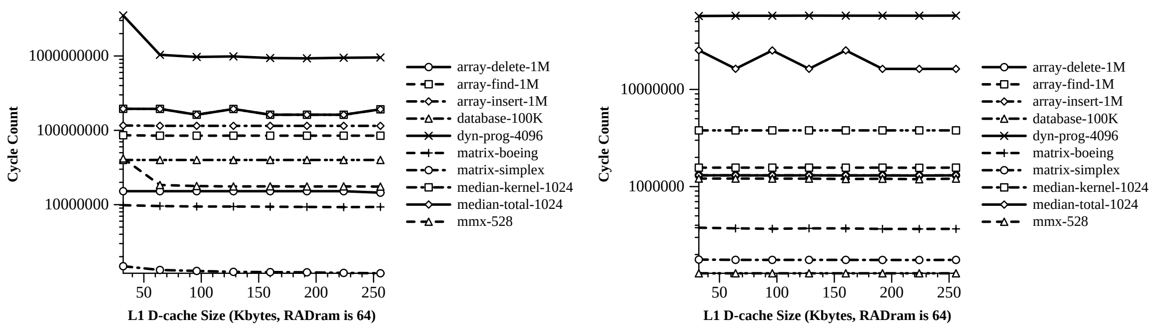

Figure 5: Conventional **(left)** and RADram **(right)** Execution Time vs. L1 Data Cache Size

Opportunities for overlapping execution of data structure operations with data-structure usage is intriguing, and is being investigated further.

The dynamic programming example maintains a very high processor / memory non-overlap, however preliminary results indicate that processor-mediated communication required by the RADram memory system eventually dominates performance. This occurs for extremely large problems that are well beyond the range of problem sizes presented in this study.

7.3 Cache Effects

The simulated processor used for this study has a default split instruction-data level-one cache. Each level-one cache is 64 kilobytes, and is 2-way associative. The processor also has a combined level-two cache of 1 megabyte and is 4-way associative. For this study the level-one data cache size was varied from 32 to 256 kilobytes. The level-two cache size was varied from 256 kilobytes to 4 megabytes.

Figure 5(left) plots total conventional application kernel execution time versus the size of the level-one data cache. As illustrated, within the range of cache sizes explored most conventional applications where unaffected. However, at the left edge of Figure 5(left) we note that some conventional applications are affected by the size of the level one cache when it fell below 64 kilobytes.

Figure 5(right) plots total RADram application kernel time versus level-one data cache size. As illustrated, all but one application was unaffected by the size of the level one cache. The median-total application shows various stride effects. The application consists of two phases. The first reads data into an array and transforms it into a special data layout required by the Active-Page memory system. The size of the level-one cache plays a role in enhancing the performance of this operation. The second phase simply dispatches the request for median filtering to the Active Page memory system and waits for the result. As evident from the performance of median-kernel, the second phase is unaffected by the size of the level one cache.

All applications were also executed with a range of level-two cache sizes. Throughout this range no significant performance differences occurred. This, combined with the level-one cache results indicates that our applications are sensitive to extremely small caches sizes, but small to reasonable size caches achieve all of the performance of large caches. Active-Page applications tend to work with large datasets. Although their primary working set may fit in a small cache, secondary working sets will not fit in realistic cache sizes. Consequently, without migrating to a cache-only architecture, our application performance is bounded by other architectural characteristics such as DRAM memory latency and bandwidth.

7.4 Analysis

To achieve a deeper understanding of the performance of application partitions, we introduce an analytic model. This model is based upon an abstract application. From this abstract application a formula is developed which models performance under various problem sizes. Additionally, total application performance is bounded by Amdahl's Law. We present this model by first developing an intuitive understanding of a partitioned application. Then we characterize processor performance with an Active-Page memory system. Finally, we compute the correlation of this analytical model with the results obtained from our RADram simulator.

7.4.1 Model

Section 2 described partitioning, and the role it plays in application performance on an Active Page memory system. To investigate partitioning in more detail, an abstract application is depicted in Figure 6. As illustrated in Figure 6 a partitioned algorithm undergoes two phases from the perspective of the processor: *activation* and post-processing. The activation phase is characterized by increased Active Page activity. The post-processing phase is characterized by decreasing Active Page activity but potential processor-memory non-overlap stalls mixed with processor computation.

The abstract application depicted in Figure 6 uses K pages of Active Page memory. The processor spends $T_A(i)$ time activating Active Page i. Initially, the processor activates all pages in sequence, thus requiring $\sum_{i=1}^{K} T_A(i)$ time to activate all pages. Immediately after activation, an Active Page begins to execute. The time required to complete execution for Active Page i is $T_C(i)$. After dispatching the activation request to all K pages, the application returns to the first page to perform any follow-up processor computation. Before the processor can perform this computation, however, the processor may be forced to stall and wait for the Active Page in memory location 1 to finish execution. At this point in Figure 6, the processor is *stalled*, waiting in non-overlap time. We account for this as $NO(1)$, or non-overlap time waiting for Active Page 1. The processor, after waiting for $NO(1)$ time for the Active

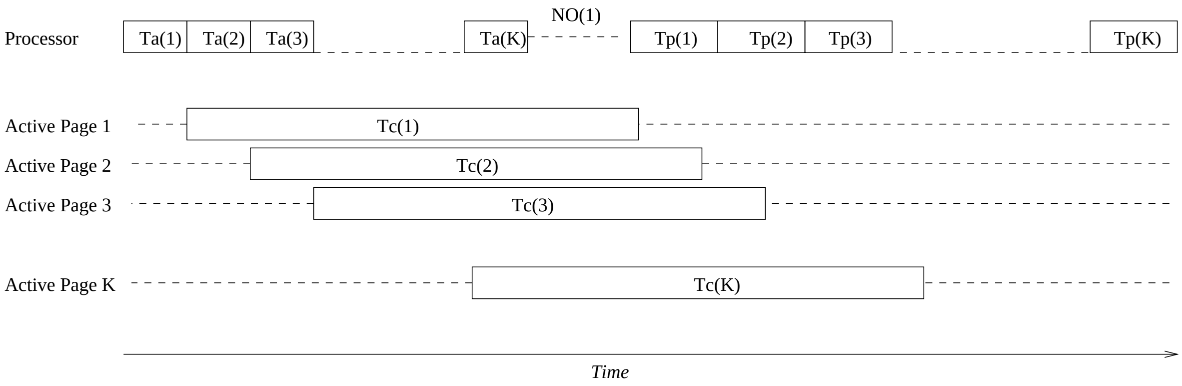

Figure 6: Abstract view of processor and Active-Page memory activity.

$$NO(i) = \max \begin{cases} 0 \\ T_C(i) - (\sum_{n=i+1}^{K} T_A(n) + \sum_{n=1}^{i-1} T_P(n) + \sum_{n=1}^{i-1} NO(n)) \end{cases}$$

$$Speedup_{partitioned} = \frac{T_{conv} \cdot \alpha \cdot K}{\sum_{i=1}^{K}(T_A(i) + T_P(i) + NO(i))}$$

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{partitioned}) + \frac{Fraction_{partitioned}}{Speedup_{partition}}}$$

Figure 7: Simplified performance model for Active Pages

Page to complete execution can then perform the follow up computation $T_P(1)$.

The abstract application shows constant per-page activation time T_A, constant per-page post-computation time T_P, and $T_P > T_A$. This means that no other stalls or processor-memory non-overlaps occur. In the general case, however, an application transitions between post-computation on page $T_P(i)$ to non-overlap time $NO(i+1)$ for the next page. This occurs for all pages within the computation.

Using this abstract application we observe that the all processor time for a single partitioned algorithm is accounted for in three distinct sets of variables: $T_A(i)$, $T_P(i)$ and $NO(i)$. Thus total kernel execution time for a partitioned application is the summation $\sum_{i=1}^{K} T_A(i) + T_P(i) + NO(i)$.

Figure 7 formalizes this model. Note that an application need not have constant per-page activation and post-activated computation time. Furthermore, an application need not have constant per-Active Page computation time. From the processors perspective, each application executes three general phases: dispatch, wait for result, and post-compute.

Figure 7 models conventional application performance in terms of $T_{conv} \cdot \alpha \cdot K$. That is the time spent by a conventional application working with a particular data set of size $\alpha \cdot K$. T_{conv} is time per item.

We note that within the non-overlap time the processor spends before post-processing of page i is a maximum of zero, or the computation time of the Active Page minus the time spent by the processor between finishing activation of page i and the current time. This intermediate time is spent either activating subsequent pages, stalled, or post-computing on previous pages.

Application	T_A (us)	T_P (us)	T_C (ms)	Pgs for overlap	Speedup correl.
Array-insert	2.058	0.387	1.250	3225	0.999
Array-delete	1.927	0.512	1.250	2438	0.999
Array-find	1.776	0.923	1.500	1624	0.999
Database	1.263	0.798	60.430	76	0.999
Matrix-simplex	2.033	4.418	13.422	8	0.968
Matrix-boeing	1.722	11.486	12.814	9	0.830
Median-kernel	0.381	0.580	3.502	9185	0.997
*MPEG-MMX	8.484	0.438	0.1423	9	0.997

Table 4: Activation time (T_A), computation time (T_C), post-activated processor time (T_P), and minimum problem size for complete overlap.

7.4.2 Correlation

In general, an average activation time (T_A) and average post-page computation time (T_P) can be measured using a small to medium data-set. Furthermore, an average Active-Page computation time (T_C) can be measured from this small data-set. Using these averages, and the model in Figure 7 a rough estimate of the non-overlap time for a particular problem size can be found. Using this estimate, it is possible to predict performance of a partitioned application for a range of problem sizes. This prediction provides insight into the particular characteristics of a partitioned application. By modeling performance as activation, post-page computation, per-page Active-Page computation, and processor-memory non-overlap

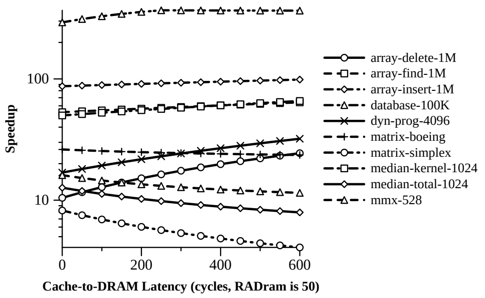

Figure 8: RADram speedup as cache-to-memory latency varies.

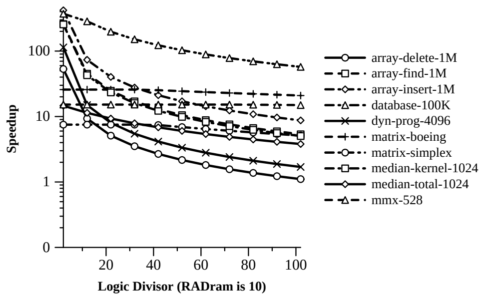

Figure 9: RADram speedup as logic speed varies.

time, it is possible to gauge performance at a variety of problem sizes and adjust the balance of work between the memory system and processor according to the expected workload of the application.

To illustrate, Table 4 lists the activation time, post-page processor time, and per-page Active Page computation time for a number of application kernels in our workload. Using a simplified version of the formulas in Figure 7 which assume constant values for these metrics, pages for complete overlap is computed. Furthermore, for each application, and for each data-point used to construct Figure 3 a predicted speedup is computed using these constant activation and computation times, and a measured non-overlap time taken from Figured 4. The correlation between the predicted speedup from using the analytical model and the actual speedup observed is shown in the rightmost column of Table 4. Most applications are well-correlated to the analytical model. A notable exception is the matrix-boeing application. This application violates the assumption of constant activation and computation times per Active Page. The times are inherently data-specific for this application and using constant values proved to be less useful than for the other applications studied.

8 Sensitivity to Technology

Our results for the RADram system demonstrate that Active Pages can be implemented with substantial success on a variety of applications. RADram technology, however, is a long-term goal which is several years in the future. Shorter-term and alternative long-term technologies can also be used to implement Active Pages. This section describes such technologies and analyzes the sensitivity of our results to some of the key parameters in the RADram system.

Current technologies exist to implement Active Pages at significantly higher cost than RADram. Such costs would limit the amount of memory available to support Active Pages, and consequently, the problem sizes of the applications. These technologies include: small merged FPGA-DRAM or SRAM chips, DRAM/SRAM macrocells in ASICs, and small processor-in-DRAM/SRAM chips. In general, logic speeds in these technologies are either equal to or better than RADram assumptions. Chip cost, however, will limit most near-term technologies to substantially smaller problem sizes. SRAM or multi-chip solutions will also have an effect on memory latencies.

We vary two technological parameters in our RADram simulations: memory latency and logic speed. First, Figure 8 plots the sensitivity of RADram speedups to memory latency in terms of cache-miss penalty. In general, the performance advantage of RADram comes from in-DRAM computation which is unaffected by cache-miss penalty. Cache effects, however, account for slight changes in both RADram and conventional system performance. These changes can result in *either* increases or decreases in speedup as cache-miss penalties increase. The sign of the slope depends upon the relative ratio of instruction cycles to memory stall cycles for the conventional versus the partitioned application. If one splits the total application runtime into two components: processor time, and memory stall time, then computes the ratio of these two values for both the conventional and partitioned applications, then the slope of application speedup versus memory latency depicted in Figure 8 will depend upon the relative ratio of these two ratios.

Second, Figure 9 plots speedup versus the speed of the application-specific circuit. The speed of application-specific circuits in the simulated RADram system is measured in relative clock divisions of the processor clock. In Figure 9 a higher logic divisor corresponds to a *slower* reconfigurable logic clock.

To generalize across applications, those operating on problems in the *scalable* region of their partitioning domain are sensitive to the speed of the Active Page computation, whereas those applications operating on problems in the saturated regions of their partitioning domain are generally insensitive to the speed of the Active Page computation.

9 Related Work

The IRAM philosophy goes to the extreme by shifting all computation to the memory system through integration of a processor onto a DRAM chip. This results in dramatically improved DRAM bandwidth and latency to the processor core, but conventional processors are not designed to exploit these improvements [B+97a]. An interesting alternative is to integrate specialized logic into DRAM to perform operations such as Read-Modify-Write [B+97b]. This alternative is promising, but we have seen that different applications can exploit significantly different computations in the memory system. Our results have shown that integrating reconfigurable logic is highly effective.

Reconfigurable computing has shown considerable success at special-purpose applications [A+96] [B+96], but has had difficulty competing with microprocessors on more general-purpose tasks such as floating-point arithmetic. Some groups focus upon building reconfigurable processors [HW97] [WH96] [RS94] [WC96], but face an even more difficult competition with commodity microprocessors. Our approach avoids these difficulties by exploiting the strengths of *both* microprocessors and reconfigurable logic. We focus upon data manipulation to make the memory system perform better for the processor. DeHon described limited integration of reconfigurable logic and DRAM in an early memo [DeH95], but did not evaluate it further.

Our philosophy is reminiscent of scatter-gather engines from a long line of supercomputers [HT72] [SH90] [CG86] [Bat74] [EJ73] [HS86] [L+92]. Hockney and Jesshope [HJ88] give a good history of such machines. Our approach, however, supports a much wider variety of data manipulations and computations than these machines. Additionally, our emphasis on commodity technologies results in a focus on different applications and design tradeoffs.

10 Future Work

Active Pages and our RADram implementation have shown great potential in our study. Unlocking this potential involves many interesting issues, including: compiler support for automatic application partitioning, operating system integration, multi-threaded application support, complete application runtimes, application-specific circuits vs. data-primitives, hierarchical computation structures, inter-page and inter-chip communication. In addition, a detailed power, yield and hardware implementation study of RADram is required.

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 [GVNG94] 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 [H+96] to produce data layouts and schedule computation within the memory system.

Integration of Active Pages with a real operating system poses new challenges. Active Pages are similar to both memory pages and parallel processors. Several open operating system issues exist such as allocation policies, paging mechanisms, scheduling, and security. Of particular concern is the high cost of swapping Active Pages to and from disk. Current FPGA technologies take 100s of milliseconds to reconfigure. New technologies, however, promise to reduce these times by several orders of magnitude [DeH96a]. Our future work will address these issues both formally and practically by clarifying the policy of interaction between an operating system and the Active Page memory system, and by simulation of a modified operating system kernel such as Linux [Bee96]. In addition to operating system studies, multi-threaded application support will be investigated.

Future work shall address inter-page and inter-chip communication issues. Before mechanisms are formalized for inter-page communication, a detailed evaluation of inter-page communication requirements is required. This evaluation must study whether inter-page communication is required by a broad class of application domains, and if so, if it should it be simulated via processor intervention or implemented with dedicated hardware support. Along with inter-page and inter-chip communication, a study of inter-page synchronization primitives is required. Such primitives, if implemented in hardware, pose additional challenges.

Finally, further evaluation of application kernels is required. Instruction sets such as MMX codify a set of data-manipulation primitives for a certain application domain. Further study of data-manipulation primitives could distill a common base set of primitives for a broad set of application domains. If such primitives exist, hybrids of the RADram implementation should be investigated.

11 Conclusion

Active Pages provide a general model of computation to exploit the coming wave of technologies for intelligent memory. Active Pages are designed to leverage existing memory interfaces and integrate well with commodity microprocessors. In fact, a primary goal of Active Pages is to provide microprocessors with enough useful data to run at peak speeds.

Our RADram implementation of Active Pages achieves substantial speedups when compared to conventional memory systems. RADram provides a large number of simple, reconfigurable computational elements which can achieve speedups up to 1000 times faster than conventional systems. This high performance, coupled with low cost through high chip yield, makes RADram a highly promising architecture for future memory systems.

References

[A+96] R. Amerson et al. Teramac – configurable custom computing. In *Symp on FPGAs for Custom Computing Machines*, pages 32–38, Napa Valley, CA, April 1996.

[AA95] P. M. Athanas and A. L. Abbott. Real-time image processing on a custom computing platform. *IEEE Computer*, 28(2):16–24, February 1995.

[Ash90] Peter J. Ashenden. The VHDL cookbook, 1st ed. Dept of CS, U of Adelaide, S Australia, July 1990.

[B+96] D. Buell et al. *Splash 2: FPGAs in a Custom Computing Machine.* IEEE Computer Society, 1996.

[B+97a] N. Bowman et al. Evaluation of existing architectures in IRAM systems. In *Workshop on Mixing Logic and DRAM*, Denver, CO, June 1997.

[B+97b] A. Brown et al. Using MML to simulate multiple dual-ported SRAMs: Parallel routing lookups in an ATM switch controller. In *Workshop on Mixing Logic and DRAM*, Denver, CO, June 1997.

[BA97] D. Burger and T. Austin. The SimpleScalar tool set, v2.0. *Comp Arch News*, 25(3), June 1997.

[Bat74] K. E. Batcher. STARAN parallel processor system hardware. *AFIPS Conf Proceedings*, pages 405–410, 1974.

[Bee96] Nelson H. F. Beebe. A bibliography of publications about the *Linux* operating system. Technical report, Ctr for Scientific Comp, Dept of Math, U of Utah, Salt Lake City, UT, May 1996.

[BGK96] D. Burger, J. Goodman, and A. Kagi. Quantifying memory bandwidth limitations in future microprocessors. In *ISCA*, Philadelphia, PA, May 1996.

[CG86] A. Charlesworth and J. Gustafson. Introducing replicated VLSI to supercomputing: The FPS-164/MAX scientific computer. *IEEE Computer*, March 1986.

[CLR96] T. Cormen, C. Leiserson, and R. Rivest. *Introduction To Algorithms.* MIT Press, Cambridge MA, 1996.

[D+92] I. Duff et al. User's guide for the Harwell-Boeing sparse matrix collection. Technical Report TR/PA/92/86, CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France, October 1992.

[DeH95] A. DeHon. Notes on integrating reconfigurable logic with DRAM arrays. Transit Note 120, MIT, AI Lab. 545 Tech Sq. Cambridge MA 02139, March 1995.

[DeH96a] André DeHon. DPGA utilization and application. In *Proc of the Int Symp on Field Programmable Gate Arrays*. ACM/SIGDA, February 1996.

[DeH96b] André DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, MIT, 1996.

[EJ73] A. Evensen and J.Troy. Introduction to the architecture of a 288-element PEPE. In *Proc. 1973 Sagamore Conf. on Par Processing*, pages 162–169, 1973.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[GVNG94] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, Inc, Englewood Cliffs, New Jersey 07632, 1994.

[GW96] D. Goodwin and K. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software–Practice & Experience*, 26(8):929–965, 1996.

[H+96] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, December 1996.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers: Architecture, Programming, and Algorithms*. Adam Hilger Ltd., Bristol, UK, second edition, 1988.

[HS86] W. D. Hillis and G. L. Steele. *The Connection Machine*. M.I.T. Press, 1986.

[HT72] R. Hintz and D. Tate. Control data STAR-100 processor design. In *COMPCON*, pages 1–4, 1972.

[HW97] J. Hauser and J. Wawrzynek. Garp – a MIPS processor with a reconfigurable coprocessor. In *Symp on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 1997.

[I+97] K. Itoh et al. Limitations and challenges of multigigabit DRAM chip design. *IEEE Journal of Solid-State Circuits*, 32(5):624–634, 1997.

[K+96] W. King et al. Using MORPH in an industrial machine vision system. In K. L. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 18–26, Napa, CA, april 1996.

[L+92] Charles E. Leiserson et al. The network architecture of the connection machine CM-5. In *Symposium on Parallel Architectures and Algorithms*, pages 272–285, San Diego, California, June 1992. ACM.

[M+96] J. Mitchell et al. *MPEG Video Compression Standard*. Chapman & Hall, New York, 1996.

[NM65] J. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[Pat95] David Patterson. Microprocessors in 2020. *Scientific American*, September 1995.

[Prz97] Steven Przybylski. Embedded DRAMs: Today and toward system-level integration. Technical report, Verdande Group, Inc., 3281 Lynn Oaks Drive, San Jose, CA, September 1997.

[R+93] D. Ross et al. An FPGA-based hardware accelerator for image processing. In W. Moore and W. Luk, editors, *More FPGAs: Proc of the 1993 Int workshop on field-programmable logic and applications*, pages 299–306, Oxford, England, 1993.

[RS94] R. Razdan and M. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Int Symp on Microarchitecture*, pages 172–180, San Jose, CA, November 1994.

[RW92] G. Rafael and R. Woods. *Digital Image Processing*. Addison-Wesley, 1992.

[Sem94] Semiconductor Industry Association. The national technology roadmap for semiconductors. http://www.sematech.org/public/roadmap/, 1994.

[SH90] N. Sammur and M. Hagan. Mapping signal processing algorithms on parallel architectures. *J of Par and Distr Comp*, 8(2):180–185, February 1990.

[SKS97] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 1997.

[WC96] R. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In *Symposium on FPGAs for Custom Computing Machines*, pages 126–135, Napa Valley, California, April 1996.

[WH96] M. Wirthlin and B. Hutchings. A dynamic instruction set computer. In *Symposium on FPGAs for Custom Computing Machines*, pages 99–107, Napa Valley, California, April 1996.

[WM95] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1), March 1995.