Appears in the 1999 International Conference on Computer Design, Austin.

ActiveOS: Virtualizing Intelligent Memory

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

Abstract

Current trends in DRAM memory chip fabrication have led many researchers to propose "intelligent memory" architectures that integrate microprocessors or logic with memory. Such architectures offer a potential solution to the growing communication bottleneck between conventional microprocessors and memory. Previous studies, however, have focused upon single-chip systems and have largely neglected off-chip communication in larger systems.

We introduce ActiveOS, an operating system which demonstrates multi-process execution on Active Pages [OCS98], a page-based intelligent memory architecture. We present results from multiprogrammed workloads running on a prototype operating system implemented on top of the SimpleScalar processor simulator. Our results indicate that paging and inter-chip communication can be scheduled to achieve high performance for applications that use Active Pages with minimal adverse effects to applications that only use conventional pages. Overall, ActiveOS allows Active Pages to accelerate individual applications by up to 1000 percent and to accelerate total workloads from 20 to 60 percent as long as physical memory can contain the working set of each individual application.

1 Introduction

Microprocessor performance continues to follow phenomenal growth curves which drive the computing industry. Unfortunately, memory systems are falling behind when "feeding" data to these processors. Processor-centric optimizations to bridge this *processor-memory gap* [WM95] [Wil95] include prefetching, speculation, out-of-order execution, and multithreading. Unfortunately, many of these approaches can lead to memory-bandwidth problems [BGK96].

DRAM memory technology, however, is growing significantly denser. 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 32 million transistors. This density has led many researchers to propose intelligent memory systems that integrate processors or logic with DRAM. The increased bandwidth and lower latency of on-chip communication promises to address many aspects of the processor-memory gap [P+97] [KADP97].

These improvements, however, are limited to single-chip systems with limited memory requirements such as PDAs. We expect that the memory demands of most systems will scale with DRAM density, and multiple memory chips will be required. We introduce *Active Pages*, a page-based architecture for intelligent memory systems (see Figure 1). Active Pages consist of a superpage of data and a collection of functions which operate on that data. Implementations of Active Pages can execute functions for hundreds of pages simultaneously, providing substantial parallelism.

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. Then the processor, 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.

Previous work [OCS98] has shown that Active Pages are a flexible architecture that can lead to dramatic performance improvements (up to 1000X) in a single-process over conventional memory systems. This paper evaluates Active Pages in a multi-process environment. Active Pages fundamentally change memory systems from a uniform storage resource to a non-uniform collection of storage and computational elements that require both management and scheduling.

We introduce *ActiveOS*, a prototype operating system which illustrates the interaction between an OS and Active Pages. Our goal is to demonstrate that Active Pages can perform correctly and efficiently in a multi-process environment. In such an environment, our results show that while paging Active Pages to and from disk can be expensive, opportunities to overlap memory and processor computation increase dramatically. On mixed workloads of conventional and Active-Page applications, Active Page single-process performance translates well to a multi-process environment as long as memory pressure is moderate. Note that our current cycle-level simulation environment limits workload sizes in our experiments. Our applications are highly scalable, however, and we expect our results to generalize to larger workload sizes.

The remainder of this paper is organized as follows: Section 2 describes the three main services provided by ActiveOS to support Active Pages. Section 3 describes our experimental methodology. Section 5 describes the applications in our workload. Section 6 presents our results.

Acknowledgments: Thanks to André Dehon, Matt Farrens, Lance Halstead, Diana Keen, and our anonymous referees. This work is supported in part by an NSF CAREER award to Fred Chong, by NSF grant CCR-9812415, by grants from Altera, and by grants from the UC Davis Academic Senate. More info at http://arch.cs.ucdavis.edu/AP

*Now at University of California, San Diego

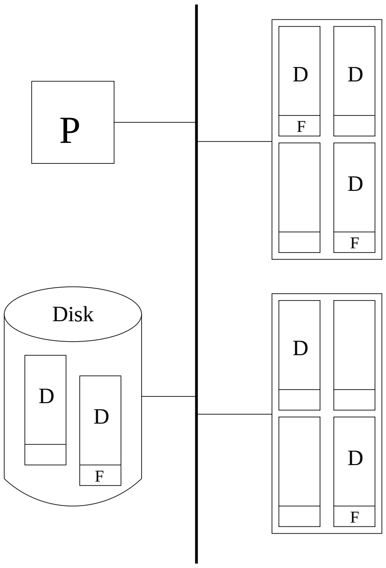

Figure 1: The Active Page architecture

Section 7 discusses related work. Finally, Sections 8 and 9 present future work and conclusions.

2 ActiveOS

ActiveOS is a prototype operating system for which the primary goal is to demonstrates mechanisms which correctly manage Active Pages in a multi-process environment. These mechanisms fall into three key categories: process services, interpage communication, and virtual memory.

2.1 Process Services

The first step in developing operating system support for Active Pages involves providing basic services for user processes to allocate pages and bind functions to them. Furthermore, a process must be able to efficiently call these functions.

The Active Page interface is designed to resemble the interface of a conventional memory system. Processes communicate with Active Pages through memory reads and writes. Specifically, the interface includes:

- Standard memory interface functions: *write(address, data)* and *read(address)*
- A set of functions available for computation on a particular Active Page: *AP_functions*
- An allocation function:

  AP_alloc(group_id)

  which allocates an Active Page in group *group_id* and returns the virtual address of the start of the superpage allocated. Pages operating on the same data will often belong to a *page group*, named by a *group_id*, in order to coordinate operations.
- A binding function:

  AP_bind(group_id, AP_functions)

  which associates a set of functions *AP_functions* with group *group_id* of Active Pages.
- A set of synchronization variables which the *AP_functions* and the processor use to coordinate activities between the processor and Active Pages within the page group.

There are two important implications of this interface. First, the synchronization variables are essentially memory-mapped registers that allow a process to efficiently invoke Active-Page functions without a system call. Second, the allocation of Active Pages, which are significantly larger than conventional pages, requires memory management of mixed superpages and pages. This memory management will become a significant issue as we discuss paging and our performance results.

2.2 Interpage Communication

An application will generally use more data than will fit on a single Active Page (each one is 512 Kbytes in our current implementation). Such data will often reside in *page groups*, a collection of Active Pages which work together to perform functions on the data. For example, inserting an element into the middle of an array that spans multiple pages would require elements to move between pages.

Such data movement is accomplished entirely via explicit interpage communication requests. Active-Page functions can reference any virtual address available to their allocating process. For performance, applications should keep references from functions to addresses on other pages to a minimum.

Our implementation uses a *processor mediated* approach to satisfying inter-page memory references. Each Active Page explicitly distinguishes between local and non-local references by checking if a virtual address is between its starting address, kept in a base register, and the starting address plus the Active-Page size. When an Active Page reaches a non-local reference it raises an interrupt and blocks. The interrupt causes the processor to obtain the memory request through a memory-mapped read and executes an OS handler. The handler consults a per-process table which will determine if the Active Page containing the reference is in physical memory. If so, the page is located and the request is satisfied. If not, a page fault occurs and the page is brought in from disk. In practice, a large number of non-local references from different pages can be satisfied upon a single interrupt, substantially reducing overhead. We assume that a single interrupt takes 3.5 ms.

Our processor-mediated approach makes inter-page communication expensive, but greatly simplifies page faults and reduces the complexity of Active Page chip implementations. We assume that our memory chips can trigger processor interrupts, but processor polling is also a possiblility.

2.3 Virtual Memory

When insufficent physical memory is available to execute all processes in the system, memory must be paged to swap space. We define the maximum required swap space to execute a set of processes as the *memory pressure*. As memory pressure increases, Active Pages will need to be paged out to disk. This implies that the data, the functions, and any active state must be written to disk. The choice of which pages to swap in and out is critical to both the correctness and performance of Active Pages. If the processor or an Active Page in memory references a page on disk, a page fault occurs. If no reference occurs, however, a page on disk may still need to be swapped in to complete a computation for a page group. Active Pages can only make computational progress if they are in physical memory. Managing Active Pages is more than traditional virtual memory management [Den70]. It is also scheduling of computation.

To help the OS schedule Active Pages, we identify three states in which an Active Page exists. These are initial (I), dormant (D), and active (A). When a page is allocated, prior to function binding, the state is initial (I). When functions are bound to a page, but no function is currently executing at the page, we identify the page as dormant (D). Finally, when a function is executing at the page, we identify the page as active (A). It is expected that an operating system will provide mechanisms to control the allocation and binding of functions to pages, but that userprocesses will transition a page from the dormant to active states.

For performance reasons, we restrict the means by which an Active Page can transition from the dormant (D) to active (A) states. As shown in Figure 2, an operating system must periodically schedule Active Pages residing in swap space back into physical memory in order to ensure program correctness. This arises because an Active Page, X, may be waiting upon a result that another page on disk, Y, is about to write to the memory of X. Since there is no reference to Y, the page will not be rescheduled without some external mechanism.

If we restrict the mechanism by which an Active Page can transition from dormant to active states, then an operating system can periodically schedule only *active* Active Pages back from swap, thus avoiding having to schedule dormant pages. The restrictions are that an Active Page never transitions from the dormant to active states without processor or inter-Active Page communication intervention. That is, an Active Page never self-activates.

Our results indicate that performance is only marginally affected by the scheduling quantum for periodically swapping pages back in to physical memory, suggesting that as small of a quantum should be used as possible such that performance is not seriously degraded in order to improve user response time.

ActiveOS currently implements a round-robin memory page scheduler for Active Pages residing in swap space. For page replacement, we adopted a Least Recently Used (LRU) algorithm which was implemented by tracking a counter value for each physical page available in the system. The operating system maintains a counter that is incremented each time a memory page is referenced by the memory management subsystem. The memory management subsystem references pages when memory is allocated, upon direct kernel access to process memory, and when a TLB miss exception occurs. Special processing is performed on this counter to ensure it does not overflow and that the relative page access times are maintained. When memory is to be swapped to disk the LRU algorithm searches the entire page space and returns a reference to the page with the lowest access counter value.

3 Methodology

In order to demonstrate multi-process operation, a mixture of Active-Page enhanced and conventional applications was chosen. For the Active-Page applications, conventional-only counterparts were implemented. The applications where chosen to represent a mixed workload, as one might encounter in a multi-user system. A brief description of each application is presented in Section 5. Both the conventional and conventional/Active-Page mixed application workloads were executed under various page replacement policies and, in the case of the mixed Active-Page/conventional application suite, with various memory resource scheduling quanta. The results of these measurements is presented in Section 6.

Parameter	Reference
CPU Clock	500 MHz
L1 I-Cache	32K
L1 D-Cache	32K
L2 Cache	256K
Reconf Logic	50 MHz
Cache Miss	114 ns
Disk Access Time	10ms
Disk Transfer Rate	10mb/sec

Table 1: Summary of simulator parameters

These measurements were taken on a prototype operating system, ActiveOS, running on a processor and memory system simulator. ActiveOS runs on the SimpleScalar v2.0 [BA97] toolkit. 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 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 a RADram memory system, an FPGA-based implementation of Active Pages described in [OCS98]. Finally, the toolset was enhanced by extensive modifications required to support the simulation of an operating system and associated multi-program workloads. The simulated machine characteristics are shown in Table 1.

4 Limitations

A significant limitation to our methodology is the use of a cycle-level processor-memory simulator. While our results are extremely accurate, simulation speed limits the size of our workloads. Consequently, our results primarly demonstrate correctness of ActiveOS mechanisms. They also reveal some interesting qualitative issues involving Active Pages on disk and overlapping memory computations with multiple processes. We note that our applications are highly-scalable streaming applications, hence we expect our results to scale to larger workload sizes. Since our processor-mediated communication method has equal cost between on- and off-chip inter-page communication, communication costs are not dependent upon number or size of memory chips. Future work will increase simulation capacity through the use of a higher-level emulation system.

5 Multiprogrammed Workload

Our workload consists of three applications which use Active Pages and three conventional applications which do not. The Active-Page applications consist of: database inserts and deletes using a C++ array class library, matrix multiply of a large finite-element sparse matrix, and protein sequence matching using dynamic programming. The conventional applications consist of: the *gcc* compiler running on input from a source file generated by the yacc parser generator,

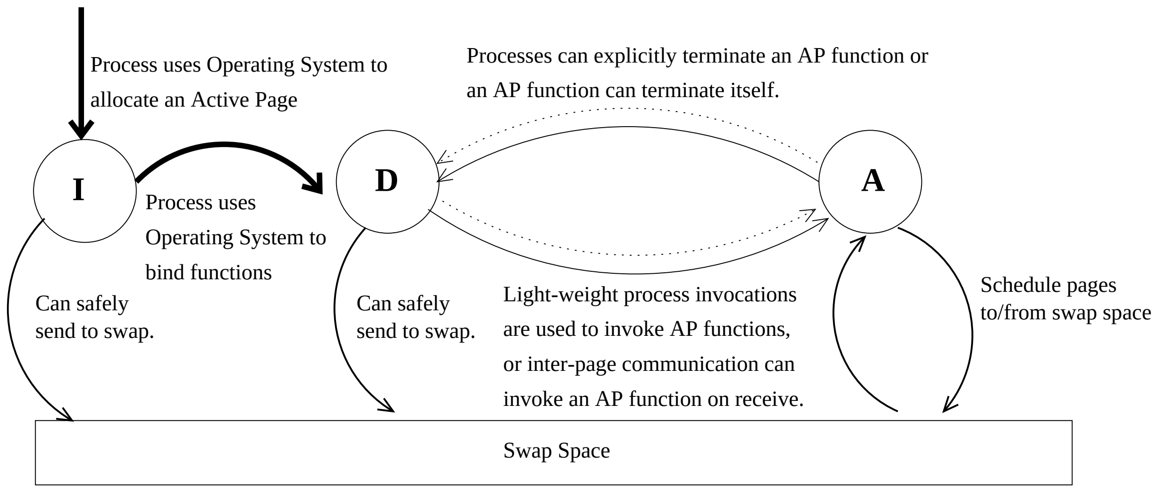

Figure 2: State transition diagram for Active Page memory

gzip compression of a 510 kilobyte executable file, and perl executing a prime-number finder. The conventional applications are largely taken from the SPEC 95 suite, with the exception of *gzip*, which we felt was more complete and up-to-date than *compress*. Each of the Active-Page applications was also implemented for a conventional memory system for comparison.

Database is a synthetic benchmark which consists of a sorted database of records, and requests on those records. The requests arrive sporadically from users and consist of inserts and deletes of individual records. The database was stored in an STL style C++ class array object which provides $\mathcal{O}(1)$ lookup. Active-Page functionality is used to shift data in parallel at the memory system.

Matrix is a matrix multiply application which multiplies a matrix by itself. The input matrices were chosen from the Harwell-Boeing benchmark suite [DGL92]. The normal and Active-Page versions of this application operate using sparse-matrices. Active-Page functionality is used to preprocess the input matrix against the columns of the matrix to be multiplied. The preprocessing consists of computing the intersecting elements and reordering the memory such that the values of the intersecting elements are compressed into individual cache lines.

DNA is a protein sequence matching application which takes in any number of input protein sequences of any length and computes all pair-wise sequence alignments [Gus97] . The result of the pair-wise alignments are values which indicate the relative likeness of each protein sequence to every other protein sequence in the input data-set. Active-Pages are used for computation of the largest common subsequence result matrices. This application makes use of inter-Active-Page communication facilities.

gcc: The *cc1* compiler is a standard component of the SPEC 95 benchmark suite [SPE95]. The input file was a 1859 line size preprocessed C file generated from the yacc parser generator.

perl: The characteristics of the *perl* interpreter are discussed in [RLV$^+$96] and [SPE95]. The input file was chosen to execute in roughly the same period of time as it took the *gcc* compiler to execute.

gzip: We include the *gzip* file compression utility to represent an updated version of the standard SPEC 95 *compress* utility. The input file was chosen to be a challenging 510 kilobyte size executable binary that compresses down to 133 kilobytes.

6 Results

In this section, we present our results from executing our multiprogrammed workload using several physical memory sizes. Our results indicate that Active-Page computations tolerate moderate memory pressure well, but performance degrades as physical memory size approaches the working-set size. This degradation tends to happen at slightly higher rates for Active-Page computations than conventional computations. Finally, although not discussed, performance was found to be relatively insensitive to the memory-resource scheduling quantum.

6.1 Application Characteristics

Table 2 lists the memory requirements, and access characteristics of each application in our workload. The data was acquired by operating the workloads in a simulated 256 Mbyte physical memory size environment to ensure that no memory paging was required. Note that the *DNA* application constitutes nearly half the working set of the workload. Also note that the Active-Page version *matrix* trades extra storage for parallelism, but *Database* actually saves space. Total memory consumption is one and a half megabytes larger for the mixed workload than for the conventional workload.

Throughout our experiments, it was observed that the TLB misses remained relatively constant as physical memory size varied. Note that the total TLB misses for the Active Page mixed workload is lower than for the conventional workload. Furthermore, a conventional application, *gcc*, makes up the dominant factor in both the conventional and mixed workloads for TLB misses. Overall, Active Page applications require fewer TLB misses, thus lowering the TLB cache pressure.

Application	Conventional	Mixed (AP/Conv)	Conventional TLB	Mixed (AP/Conv) TLB
Database	2512k	1720k	11121	1766
DNA	8032k	8384k	31230	623
Matrix	664k	2680k	14048	8620
GCC	2820k	2820k	202867	204827
Perl	1276k	1276k	16155	16758
GZIP	628k	628k	10914	12318
Total:	15932k	17508k	286335	244912

Table 2: Memory requirements and number of TLB misses of conventional and Active-Page applications

Memory Size	15360k	13312k	11264k	9216k	7168k
Conventional/LRU	0k	2052k	4100k	6148k	8196k
Mixed/LRU	5768k	6640k	8400k	10392k	12472k

Table 3: Required swap space for conventional versus mixed workloads

Table 3 lists required swap space versus physical memory size for our conventional and mixed workloads under various paging policies. The mixed workload as a whole requires roughly four megabytes more physical memory than the conventional workload. Referring back to Table 2, one and a half megabytes of this difference comes from algorithmic restructuring in *database* and *matrix*. The remaining two and half megabytes, however, come from fragmentation and superpage effects during swaps. The current ActiveOS implementation does not compact partially used conventional pages to free up more super-pages. Furthermore, Active Pages must be swapped out as entire super-pages even when a single conventional page of storage is all that is needed. Future versions of ActiveOS will be optimized to reduce these effects and allow for better performance at higher memory pressures.

Our simulation results show that ActiveOS can run our mixed workload with high performance. Figure 3 plots application time versus physical memory size for each application. We note that Active Page applications take significant advantage of the Active Page memory system to enhance their performance. Since each application is only charged process time for when its process is scheduled in the processor, each Active Page application gets a potential six-fold performance advantage if its Active Pages execute while other processes are scheduled. Referring to the wall clock times in Figure 3, we can see that conventional applications in our mixed workload benefit from the reduced time that Active Page applications spend in the system.

The performance of the Active Page *DNA* application is most affected by moderate reductions in physical memory size. Also note a pathological case in which the interaction of LRU with the access patterns of *DNA* cause 13 Mbytes to perform better than 15 Mbytes of memory.

Further note that the Active-Page version of *matrix* performs extremely poorly when physical memory is very low. At 3 Mbytes, *matrix* performs an order of magnitude worse than at 7 Mbytes. This occurs because of frequent and repeated access to each Active Page. Although each page is touched repeatedly, the quantity of data used from them is very small. Under extreme memory pressure, intense swapping occurs with very little access to data by the processor.

Overall, we see that ActiveOS mechanisms can efficiently virtualize Active Pages for light and moderate memory pressures. Although our simulation infrastructure limits workloads to sizes smaller than those in future systems, the data-intensive nature of our applications will scale well to large sizes and we expect our results to be qualitatively similar for larger workloads.

7 Related Work

Active Disks, as proposed by Riedel and Gibson [RG97], provide a mechanism for doing application specific computation at the disks. This work is in a similar vein, in that it is offloading some of the work to the disk, with similar consequences. A small embedded processor is inserted into the disk controller which then executes portions of the user code. Reidel and Gibson argue that by moving this computation to the other side of the disk interconnect, more efficient use of disk and host resources can be achieved. It is then further argued that Filters, Real-Time, Batching, and High-level support are some of the applications which should benefit from Active Disks. Two applications, a database select and a parallel sort, are then looked at in greater depth.

The idea of using multiple page sizes is well established. Many commodity processors support the use of multiple page sizes, such as the DEC Alpha, SPARC, Intel, and HP PA-RISC [TH94]. A discussion of the use of multiple page sizes is done by Talluri *et al.* [TKHP92]. Further work is done by Tallurri and Hill to expand placement algorithms to support superpages [TH94]. Active Pages, however, involve both superpage management and computation scheduling.

There has been a fair amount of work in the design of active memory systems, most notably are the SWIM project [ACK94] and the PAM project [VBR+96]. Asthana *et al.* proposed and built an active memory system for network applications known as SWIM. SWIM is a memory system built out of SRAM with small processing elements on each chip. The memory system then has a back end on it which is interfaced to communication lines and disk subsystems. The Programmable Active Memories Project (PAM) at Digital's Paris Research Laboratory focused on the performance of a memory mapped FPGA coprocessor on compute intensive applications. The FPGA coprocessor was linked to a high bandwidth SRAM bank in order to keep up with the increased demand for data in the FPGA. Other groups have also proposed computation in memory [SS95], but none, to the authors' knowledge, have examined the operating system ramifications of doing such.

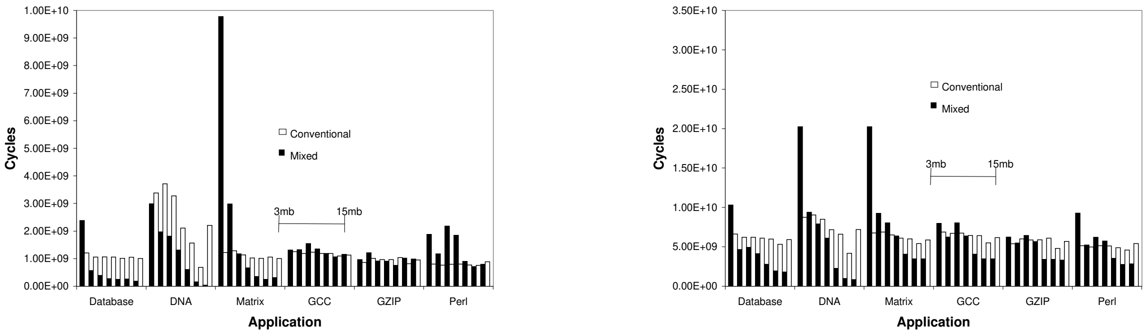

Figure 3: Conventional vs. mixed workload times for each application for LRU page replacement. Process time is given on the left and wall-clock time on the right (bars are for 3, 5, 7, 9, 11, 13, and 15 Mbytes of physical memory).

8 Future Work

Although ActiveOS has shown promising results in virtualizing Active Pages for a multi-process environment, many issues remain. A major effort is underway to automate the application partitioning process. The number of applications in the multi-program workload needs to be increased, and the working set size of each application also needs to be expanded. Furthermore, virtual memory paging and memory resource scheduling techniques need to be investigated in more detail. Operating system support for Active Page memories with hardware communication facilities will need to be addressed. Finally, support for multiple sizes of Active Pages may prove useful.

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 [VG95] 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 [HAA+96] to produce data layouts and schedule computation within the memory system.

The current application workload operates within a confined memory range. The required memory core size for the normal application mix requires 16 megabytes of memory, not counting the operating system. Extending the memory size of this workload naturally implies extending the length of the execution time for each application. Future work will study applications with larger memory sizes. Furthermore, more Active-Page applications will be implemented providing an opportunity for a better understanding of Active-Page application and OS interaction.

With more applications available, a better understanding of their paging characteristics can be achieved. Future work will focus on characterizing Active-Page program behavior in more detail, and then designing efficient paging policies to support both Active-Page and conventional applications. Furthermore, work will address active Active-Page priorities and memory resource priority scheduling. All current paging policies rely upon access information generated by the processor. However, Active-Pages degrade the quality of this information because activity can be occurring solely inside the memory system. This activity is currently not tracked by the current processor access statistics.

Future work will expand on this notion of an information gap and provide solutions for it. Currently, we suggest a solution of Active-Page addressable per-page priorities. This extends the notion of the tri-states for an Active-Page. Thus an Active-Page exists in either the initial (I), dormant (D), or active(A) with some priority (K) states. The priority based mechanisms would ensure a contract between the operating system and a user process. The contract is that if a process properly varies the priorities of its associated Active-Pages, then the operating system can more efficiently execute the application under low-memory conditions. This new information of Active-Page priorities can then be used to offset the lack of information the operating system currently has about the Active-Page activity. Fairness between Active Page and conventional processes, however, will be an important issue.

Nothing restricts an Active-Page memory from containing a hardware-based communication network. However, without hardware mechanisms in place to restrict inter-page communication, potential security and safety holes are created. For this reason, hardware assisted communication in Active-Page memory devices must have mechanisms to identify Active-Pages as belonging to a particular process. The user-level function must be restricted from changing this process identification without sufficient operating system intervention. Furthermore, all communications must be confined to within the same process or to processes defined in a restricted fashion. Inter-process communication occurring within a hardware assisted communication network must be restrictable by the operating system. Finally, errant and malicious application functions bound to an Active-Page must be restricted from affecting other super-pages in the memory system.

Current work has focused on the RADram [OCS98] Active Page memory system. It is anticipated that future Active Page memories will include hardware support for inter-Active Page communication. The techniques developed in this paper to provide virtual memory support for Active Page memories are still applicable. However, hardware communication networks for inter-Active Page memory accesses will pose additional operating system challenges in order to minimize communication delay, ensure process protection, and maximize overall application performance.

Finally, an investigation into multi-grain Active-Page implementations and associated operating system support is required. Research into application partitioning suggests that one method of tuning an application partition to a particular problem size is to vary the size of the Active Page. Rather than choose a fixed overall page size, it is proposed that by providing two to four sizes of Active Pages, the maximum possible speedup can be extended and maximized across a broader range of problem sizes. Multi-grain Active-Page support extends the Active-Page memory model but poses new hardware and software design challenges.

9 Conclusion

ActiveOS demonstrates that intelligent memory systems can be virtualized to achieve high multiprogrammed performance. Active Pages provide a page-based intelligent memory architecture that makes this virtualization possible. In fact, multiprogramming increases the opportunities for overlapping processor and memory computation. The fact that memory performs computation, however, means that some reasonable amount of physical memory must be available. Consequently, our results show high performance from zero to moderate memory pressures, but performance degrades significantly as available memory falls below the working set size of any one Active-Page application.

References

[ACK94] Abhaya Asthana, Mark Cravatts, and Paul Krzyzanowski. Design of an active memory system for network applications. In *International Workshop on Memory Technology, Design and Testing*, pages 58–63. IEEE Computer Society Press, 1994.

[BA97] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3), June 1997.

[BGK96] D. Burger, J. Goodman, and A. Kagi. Quantifying memory bandwidth limitations in future microprocessors. In *International Symposium on Computer Architecture*, Philadelphia, Pennsylvania, May 1996. ACM.

[Den70] Peter J. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, September 1970.

[DGL92] Ian S. Duff, Roger G. Grimes, and John G. Lewis. 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.

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

[HAA+96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, December 1996.

[KADP97] Kimberly Keeton, Remzi Arpaci-Dusseau, and David A. Patterson. IRAM and SmartSIMM: Overcoming the I/O bus bottleneck,. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, Denver, Colorado, June 1997.

[OCS98] Mark Oskin, Frederic T. Chong, and Timothy 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. To Appear.

[P+97] D. Patterson et al. The case for intelligent RAM: IRAM. *IEEE Micro*, April 1997.

[RG97] Erik Riedal and Garth Gibson. Active disks - remote execution for network-attached storage. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213-3890, December 1997.

[RLV+96] Theodore H. Romer, Denis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VII*, pages 150–159. ACM Press, October 1996.

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

[SPE95] SPEC. Spec benchmark specifications. SPEC95 Benchmark Release, 1995.

[SS95] Steven Van Singel and Nandit Soparkar. Logic-enhanced memories for data-intensive processing (extended abstract). In *International Workshop on Memory Technology, Design and Testing*, pages 88–94. IEEE Computer Society Press, 1995.

[TH94] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB perfomance of superpages with less operating system support. In *Architectural Support for Programming Languages and Operating Systems V*, pages 171–182. ACM Press, 1994.

[TKHP92] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *19th annual International Symposium on Computer Architecture*, pages 415–424. ACM Press, 1992.

[VBR+96] J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories : Reconfigurable systems come of age. *IEEE Transactions on VLSI systems*, 4(1), March 1996.

[VG95] F. Vahid and D. Gajski. SLIF: A specification-level intermediate format for system design. In *Proceedings of the European Design and Test Conference*, pages 185–189, Washington, March 6–9 1995. IEEE Computer Society Press.

[Wil95] M. Wilkes. The memory wall and the CMOS end-point. *Computer Architecture News*, 23(4), September 1995.

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