Appears in the 1999 International Symposium on Microarchitecture. Haifa, Israel.

Exploiting ILP in Page-Based Intelligent Memory

Mark Oskin, Justin Hensley, Diana Keen, Frederic T. Chong, Matthew Farrens, and Aneet Chopra
Department of Computer Science
University of California at Davis

Abstract

This study compares the speed, area, and power of different implementations of Active Pages [OCS98], an intelligent memory system which helps bridge the growing gap between processor and memory performance by associating simple functions with each page of data. Previous investigations have shown up to 1000X speedups using a block of reconfigurable logic to implement these functions next to each subarray on a DRAM chip.

In this study, we show that instruction-level parallelism, not hardware specialization, is the key to the previous success with reconfigurable logic. In order to demonstrate this fact, an Active Page implementation based upon a simplified VLIW processor was developed. Unlike conventional VLIW processors, power and area constraints lead to a design which has a small number of pipeline stages. Our results demonstrate that a four-wide VLIW processor attains comparable performance to that of pure FPGA logic but requires significantly less area and power.

1 Introduction

Accessing and manipulating data has become increasingly expensive as the gap between microprocessor and memory system performance has widened. Rapid advances in DRAM density have led to several proposals to move computational logic into the memory system [P^+97] [GHI95] [MSM97]. We focus on Active Pages [OCS98], a page-based model of computation which associates simple functions with each page of memory. For example, an array data structure stored in memory may have functions bound to the Active Page memory to perform common manipulation functions such as *insert*, *delete* and *find*.

Active Page memory systems are intended to enhance microprocessor performance in a processor-memory architecture and use the same interface as conventional memory systems. Active Page data is modified with conventional memory reads and writes, while Active Page functions are invoked through memory-mapped writes. Synchronization is accomplished via user-defined memory locations.

Active Pages are also capable of exploiting a high degree of parallelism. A memory system typically contains hundreds to thousands of pages of physical memory; Active Page systems can potentially support simultaneous computations at each of these pages. This page-based computation supports data parallelism similar to supercomputers of the past, but in a ubiquitous technology aimed at commodity applications.

Previous work described an implementation of Active Pages which integrated a block of reconfigurable logic with each subarray in a DRAM chip. By partitioning tasks between the microprocessor and this Active Page memory system, speedups (compared to a conventional uniprocessor system running data-intensive applications) of over 1000X were achieved. Although this approach was shown to be extremely promising, several unanswered questions remain.

Foremost among those questions was whether reconfigurability is necessary to achieve high performance. This study compares the reconfigurable Active Page implementation with several designs based upon a simple processor core. The results reveal that the key to the performance gains achieved by Active Page memory systems is the support of fine-grained parallelism, not logic specialization.

To exploit this parallelism, we replaced the reconfigurable logic with several different scalar and VLIW processor designs and measured their performance using cycle-by-cycle simulation. We have also evaluated their various power and area requirements via design synthesis. In this paper, we show that the constraints of the DRAM environment and the needs of our data-intensive applications create design pressures which differ substantially from the conventional microprocessor environment. In particular, power constraints limit the clock speed of deeply pipelined designs with complex forwarding logic. Although somewhat counter-intuitive, short pipelines can actually be clocked faster than deep pipelines when the power budget is low.

The next section gives some background on Active Pages. Section 3 describes the Active Page implementations we explored. Section 4 describes the experimental methodology we used to evaluate those implementations. Section 5 discusses our results. Section 6 describes related work. Finally, Section 7 discusses future work and Section 8 presents our conclusions.

2 Background

The Active Page project builds upon several ground-breaking studies on intelligent memory. In particular, the Berkeley IRAM project has demonstrated many of the benefits of integrating processors with memory in upcoming DRAM technologies [P^+97] [F^+97]. However, the focus in IRAM is on replacing conventional architectures with single-chip systems. While such systems have great potential for portable personal devices, memory requirements for desktop applications are likely to stay ahead of single-chip capacities.

Active Pages, on the other hand, is designed to replace DRAM in conventional systems. Active Page chips will function as both conventional and intelligent memory. To exploit

Acknowledgments: Thanks to Neva Corrigan, André DeHon, Lance Halsted, Rich Lethin, Neil McKenzie, Tim Sherwood and Deborah Wallach. This work is supported in part by an NSF CAREER award to Fred Chong, by NSF grant CCR-9812415, by an NPSC fellowship to Diana Keen, by grants from Mitsubishi and Altera, and by grants from the UC Davis Academic Senate. Aneet Chopra is currently at Intel. More info at http://arch.cs.ucdavis.edu/AP

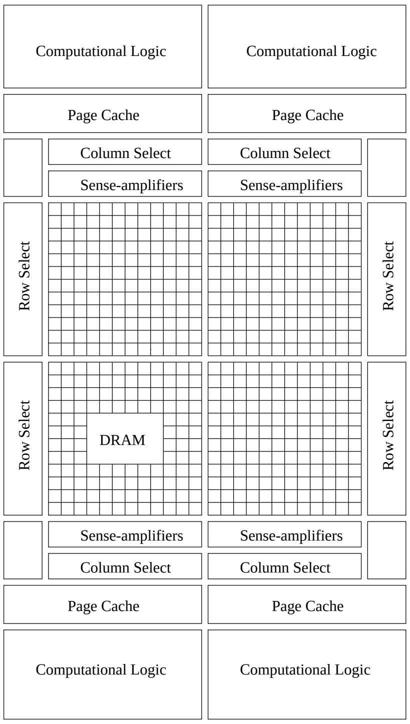

Figure 1: Active Page architecture (4 pages)

the intelligent memory, computation for an application must be divided, or *partitioned*, between the main processor and the memory system. For example, Active Page functions are used to gather operands for a sparse-matrix multiply and pass those operands on to the processor for multiplication. To perform such a computation, the matrix data and gathering functions must first be loaded into a memory system that supports Active Pages. The processor then, through a series of memory-mapped writes, starts the gather functions in the memory system. As the operands are gathered, the processor reads them from user-defined output areas in each page, multiplies them, and writes the results back to the array data structures in memory. To keep operands from being read before they are ready, user-defined synchronization variables are used.

If two Active Pages need to share data, the main processor reads the data from one and writes to the other. This *processor-mediated* approach to inter-page communication simplifies system design but assumes infrequent communication. The current prototype Active Page architectures assume a program-guided approach to communication. This means that the main thread of execution on the host processor is required to poll its own Active Pages and perform any inter-page communication requests explicitly. Future work will examine on-chip and off-chip hardware communication facilities.

3 Architectures

The focus of this paper is on evaluating different implementations of the logic that performs Active Page computation. Our study compares and contrasts three approaches: the original FPGA implementation, one that features a scalar single-issue conventional MIPS-like RISC processor, and one that employs a multiple-issue VLIW processor.

All of our implementations are targeted for fabrication in commodity DRAM technology in a three- to five-year time frame. The Semiconductor Industry Association (SIA) predicts that 1G-bit DRAMs will be produced in commodity by the year 2001 [Sem97]. We assume up to half the area on such a chip will be required for the logic, reducing the memory to 0.5G bits. To reduce signal delay and power consumption, a 1G-bit DRAM is expected to be divided into 512K-byte subarrays [I+97]. Therefore, we assume each Active Page will consist of one of these subarrays, and we add our interfacing logic to each subarray. Given 128 subarrays per 1G-bit chip, we have an area budget of 256K transistors per Active Page for each associated computational logic element. Allowing for degraded logic in a DRAM process, this results in approximately 32M transistors for logic [Prz97]. Although we allocate up to half of the available chip for computational logic, clearly reducing this number will broaden the acceptance of Active Pages within the DRAM manufacturing market.

Each of these designs was also found to benefit from a small page cache that is integrated between the page based processing element and the local DRAM subarray. Within these design constraints we evaluate our three Active Page configurations.

3.1 Reconfigurable Logic (FPGA-RAM)

Our FPGA-based Active Page implementation was first introduced in [OCS98]. The use of reconfigurable logic is motivated by three factors. First, reconfigurable logic has been shown to perform extremely well for special-purpose applications [B+96] [A+96]. Second, Active-Page functions are simple and require relatively low logic resources. Third, the uniform nature of reconfigurable logic is expected to facilitate defect tolerance and keep chip yields high [OCS98]. With an area budget of roughly 256K transistors associated with each Active Page, approximately 256 reconfigurable Logic Elements (LEs) can be implemented in current Altera FPGA technology [Alt98]. This LE budget, while minimal, proved adequate for the applications studied.

The reconfigurable logic is configured to support the functions associated with each Active Page as the page is allocated or swapped in from disk. In current technologies, reconfiguration time will add about 50% to the allocation or swapping of a conventional super-page. Emerging FPGA technologies promise to reduce this overhead to less than 5% [G+99].

Exposing the reconfigurable nature of FPGA logic has its disadvantages. The programming model for reconfigurable logic differs substantially from traditional sequential process programming. Although tools exist that can map conventional programs onto reconfigurable logic, the density and performance of the resulting circuit is often poor. Designing circuits for execution in reconfigurable logic is a different sort of skill than writing traditional software, and this may hinder the acceptance of FPGA-based Active Page models.

In addition, power and area requirements are of great concern. Already, the FPGA based architecture requires nearly 50% of the chip area for computational logic. Adding a small page cache may require an additional 5-15% of chip area. Clearly, a design that consumes less area is desirable. While the programmability concern may be resolved by development of an efficient library of routines that a programmer may use for building applications, the area issue is more challenging. Future reconfigurable architectures, such as the

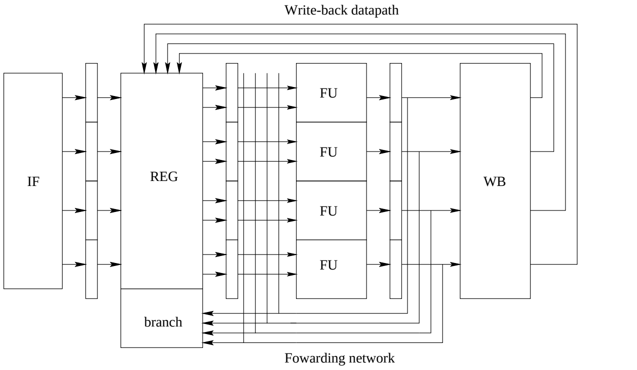

Figure 2: VLIW-4 Processor using a conventional four-stage pipeline

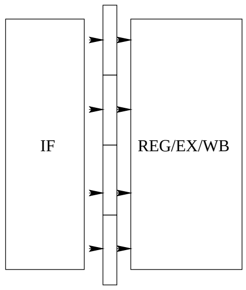

Figure 3: VLIW-4 Processor using a two-stage pipeline

PipeRench architecture from CMU [G+99], are currently being explored as potential low power, low area reconfigurable architecture candidates.

3.2 Scalar Processor (Scalar-RAM)

The simplest alternative to using FPGA logic is to replace it with a small RISC processor. In this study, we have chosen to replace it with a single-issue processor similar to the MIPS R3000 [KH92]. Since many of the features of the R3000 are not required for Active Page operation, the available instruction set was substantially reduced. Our current scalar implementation, Scalar-RAM, supports a minimal, but functionally complete, 22 instructions.

The Scalar-RAM processor was designed in VHDL. It uses a conventional five-stage pipeline with data forwarding. To eliminate branch delay, an aggressive dual-ported instruction cache was modeled that is capable of fetching down both paths of a conditional branch instruction. To conserve area, various features common to conventional processors were omitted, such as advanced memory management, co-processor support, and various comparison and branch instructions. These features were not found to be necessary for our applications. Instructions are 32 bits long, and there are 32 integer registers. Floating point support is not included.

Scalar-RAM simplifies Active Page allocation and swapping by avoiding complicated logic reconfiguration. Code to implement Active Page functions, however, must be stored in the DRAM subarray. This storage takes away from data storage space, but its effect is limited and generally requires less than 5% of the DRAM storage. Storage for logic configurations in the FPGA system is internal to the reconfigurable logic area. For Scalar Active Page memory systems, the address space is re-mapped so that all Active Page data and code appear in separate, contiguous address spaces.

3.3 Multiple-issue (VLIW-RAM)

Performance comparisons between the reconfigurable and scalar implementations reveal that the scalar architecture fails to achieve the same level of performance as the reconfigurable architecture. The scalar implementation did, however, require much less power and area than the FPGA. In order to isolate the source of the FPGA's higher performance (instruction level parallelism or logic specialization), a VLIW architecture (VLIW-RAM) was designed. The design goal of VLIW-RAM was to exploit the same level of ILP available to FPGA-RAM but have power and area characteristics closer to Scalar-RAM. Section 5 will show in more detail the benefits of parallelism versus specialization.

In this study, we took a conservative approach to designing the VLIW-RAM architecture. Since we are working within an embedded DRAM process with very confined power and area requirements, we examined three different VLIW processor designs with different instruction and functional unit widths (two-wide, four-wide, and eight-wide). In Section 5, we evaluate these designs and isolate the optimal width based upon performance, performance/area, and performance/watt.

Since area constraints limit FPGA-RAM to 256 reconfigurable logic elements per Active Page, Active Page applications can generally consume no more than two 32-bit data values from each page cache per cycle. Consequently, VLIW-RAM has two data ports per page cache. This artificial restriction was placed on the VLIW in order to clearly isolate the source of performance gains in the FPGA architecture. In Section 5, an application study is also presented that discusses which applications may benefit from additional data cache ports. Future work will explore this effect.

Branches can severely limit the amount of parallelism in a VLIW. To minimize extra control instructions, we borrow a common VLIW technique [Fis83] and allow several branches per instruction parcel. The VLIW traverses the first taken branch, thus permitting a form of precedence among the branch conditions. Like Scalar-RAM, VLIW-RAM fetches along all possible branch targets. While this is done in order to limit pipeline stalls, it does complicate the instruction cache design. Hence, we limit the number of conditional branches to two and direct branches to one. Consequently, there are no more than three possible destinations, which implies a three-ported instruction cache.

One drawback of VLIW designs is the super-linear increase in hardware as functional unit width increases. Although the number of register read and write ports scales linearly, a traditional four- or five-stage pipeline requires a super-linear growth in data forwarding busses. This is illustrated in Figure 2. Migrating from a one-wide to a four wide processor entails a corresponding growth in forwarding logic. Furthermore, the cross-bar interconnect between the second and third pipeline stages implies a quadratic growth in resources. We note that a two-stage VLIW processor design does not require this super-linear growth in forwarding logic. This is depicted in Figure 3.

Furthermore, power constraints prevent a deeply pipelined processor from taking advantage of high clock rates. This sit-

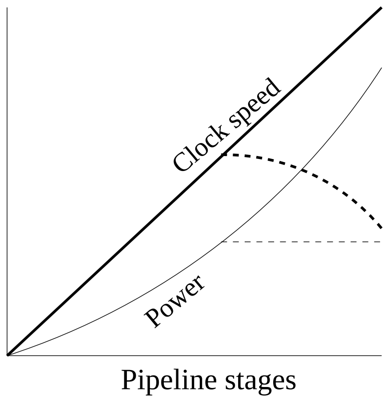

Figure 4: Pipeline stages versus clock speed and power consumption. The dotted line illustrates the effect of working under a power constraint.

uation is illustrated in Figure 4. The solid lines illustrate the conventional situation where no severe power constraint is placed upon the processor design. As the number of pipeline stages increases, the quadratic increase in forwarding logic results in a commensurate increase in power. Once we reach the limit of our power budget, then power must be held constant and clock speed must decrease. This power-limited situation is shown with the dotted lines. With the power constraint, it shows that the clock speed is faster with shorter pipelines. This is our motivation for exploring a short pipeline.

As power and area limitations become less restrictive, future designs can achieve higher clock rates. A deeper processor pipeline may be necessary to take advantage of these clock rates. One method to reduce the required hardware interconnect of the deeper pipeline is to limit the instruction set architecture [Ell86] [CNO+88] [LFK+93]. This would limit the required forwarding logic, trading off regularity in instruction set architecture for reduced hardware area. Current work has not focused on such design techniques and their potential impact on application performance.

Not shown in Figure 4 is processor performance. It should be noted that a deeper pipeline may also adversely affect performance, particularly if branch delay slots are introduced into the ISA, further mitigating the clock benefits from deeper pipelining.

Our DRAM design environment is substantially different from the conventional microprocessor arena. This leads us to explore the lower area and power of the two-stage pipeline processor design, which we will compare to the four-stage design. The two pipeline stages are: fetch and decode / execute / write-back. This pipeline was designed after timing the components from a four stage processor (fetch, decode, execute, and write-back) individually. It was observed that the slowest component was instruction fetch, and this was largely due to the cache access. The latter three stages, when combined, take approximately the same time as the instruction fetch. In Section 5, we demonstrate that the two-stage design has greater performance/watt than the four-stage design and takes a significantly smaller amount of chip area.

Power and area are just two of many factors influencing the decision to use a VLIW derivative. An obvious alternative to VLIW is a superscalar processor. However, Active Page computations are primarily simple, regular kernels. The regularity of these computations makes them ideal for static scheduling. Because our limited ISA supports only one-cycle operations, VLIW instructions do not stall waiting for an operand in a functional unit. This reduces the need for dynamic scheduling. Furthermore, with the addition of data prefetching, the VLIW gains the capability of masking load latencies for these regular computations. Another key benefit to VLIW is that it permits several branches to be executed in a single cycle. The superscalar would need to serialize these branches or use branch prediction and speculation to gain the same control performance. Finally, area and power requirements are higher for a superscalar processor than for a comparable VLIW processor. In this environment, the small performance advantages of a superscalar do not justify the substantial complexity required. In the DSP domain, several embedded processor designs have followed similar motivations [SRD96] [Ses98] [HYYS96].

Another alternative to the VLIW processor is a vector processor core. Vector processing, while attractive for data-streaming applications, is not suitable for the full range of Active Pages applications. Although several of the kernels in this study are vectorizable, future work will examine applications with irregular instruction and data patterns such as garbage collection, decision tree generation and artificial intelligence search algorithms.

4 Methodology

We simulate our Active Page implementations running a suite of applications from [OCS98]. The applications represent a range of data-intensive problems from both engineering and commodity domains. Not only are these applications well-suited for the Active Page model, they are also representative of a class of data streaming and processing tasks likely to drive future computing technologies.

Table 1 summarizes the attributes of these applications. Applications such as array, database, median, and MMX represent applications that will be important to PCs and the commodity DRAM market. Applications such as sparse matrix multiply and dynamic programming represent higher-end engineering environments.

In general, our applications fall into two categories: *processor centric* or *memory centric*. Processor-centric applications use Active Page memory to keep the processor fed with useful data and keep processor utilization high. Memory-centric applications take advantage of memory bandwidth and data parallelism within the Active Page memory system.

The three different configurations compared in this study (FPGA, Scalar, and VLIW) were evaluated by running our benchmark programs on a cycle-by-cycle simulator on the input problem sizes shown in Table 1. A large, realistic data set is used for each application. The simulator was created by adding an Active Page memory system module to the SimpleScalar v2.0 tool set [BA97]. The benchmark programs were hand-designed for the FPGA architecture in VHDL. Furthermore, for each of the processor-based architectures, the applications were hand optimized. Optimization for the applications on the processor variants began by constructing C code for each application and recompiling them with the GNU gcc compiler using the *-O3* flag. Hand optimization began by first cleaning up the assembly code generated by the compiler. Next, a combination of loop unrolling and software pipelining was applied. Finally, each application was statically scheduled for the various VLIW widths. Code optimiza-

Memory-Centric Applications					
Name	**Application**	**Processor Computation**	**Active Page Computation**	**Core Computation**	**Problem Size**
Array	C++ standard template library array class	C++ code using array class Cross-page moves	Array insert, delete, and find	a[i] = a[i+1] if (a[i] == 'c') x++	20MB
Database	Address Database	Initiates queries Summarizes results	Searches unindexed data	if (*n == ','), break L1; else if (!*m or (*m !=*n)) break L2	25MB
Median	Median filter for images	Image I/O	Median of neighboring pixels	find median of nine numbers	16MB
Dynamic Prog	Protein sequence matching	Backtracking	Compute MINs and fills table	if (x == y) B = 1; C = ul + 1 elseif (u > l) {B=2;C=u;} else {B=3;C=l;}	50MB
Processor-Centric Applications					
Name	**Application**	**Processor Computation**	**Active Page Computation**	**Core Computation**	**Problem Size**

Matrix	Matrix multiply for Simplex and finite element	Floating point multiplies	Index comparison and gather/scatter of data	if (X[m] < Y[v]) m++;
				else if (X[m] > Y[v]) v++;
				else {set of ld's and st's}
					15MB and 15MB

MPEG-MMX	MPEG decoder using MMX instructions	MMX dispatch Discrete cosine transform	MMX instructions	c[i] = min(a[i]+b[i],255)	64MB

Table 1: Summary of partitioning of applications between processor and Active Pages

tion and VLIW width scheduling were performed iteratively in order to achieve optimum performance.

We first performed the data cache simulations first on the FPGA configuration. Since the FPGA has no instruction stream, it suffers no instruction fetch misses, and it will generally exhibit the highest data request rate of all the configurations. Since the applications used are largely data-streaming applications, little non-critical algorithmic data is introduced by the Scalar and VLIW-RAM architectures, so the results from the FPGA simulations can be used for all architectures.

Once the size and characteristics of the data cache were known, simulations were performed to calculate the appropriate size of the instruction cache for Scalar and VLIW implementations. These simulations were followed by a set of simulations that measured the performance of the FPGA, Scalar and a number of VLIW processor variants. The best VLIW configuration was chosen, and more simulations were run to measure the performance, performance per watt, and performance per area of the remaining three Active Page architectures. Finally, the effect of clock-cycle scaling on the three architectures was evaluated.

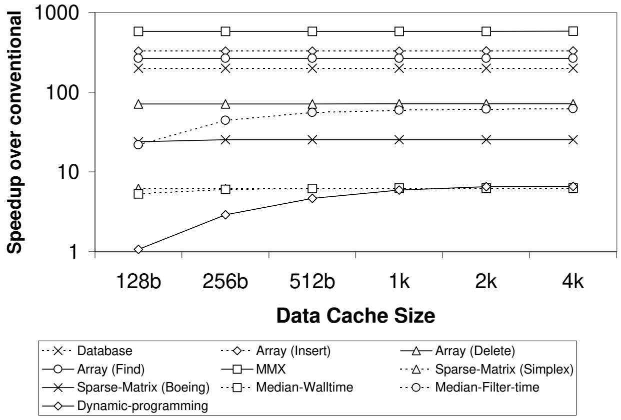

Figure 5: Data cache size

5 Results

In this section, we compare our three Active Page architectures: FPGA, Scalar, and VLIW. Our results reveal that a VLIW architecture is the best choice, saving substantial area and power without sacrificing performance.

We begin by profiling applications to optimize cache parameters for each architecture. Next, we identify an optimum instruction width for the VLIW architecture. We continue by demonstrating substantial gains through prefetching, an option not available to our FPGA architecture due to area limitations. Finally, we present comparisons of performance per watt and performance per area for the three architectures.

5.1 Data and Instruction Caches

Although Active Page logic is adjacent to the DRAM subarray of each page on chip, memory latencies are still a concern. Furthermore, cycling the DRAM on every processor cycle would require a substantial power budget. Consequently, a cache is placed between the computational logic and DRAM sub-array. In this section, we present simulation results that were used to determine the size and organization of this cache.

We utilize a 256-bit datapath between logic and DRAM for each Active Page to match a 32-byte cache block size. This provides sufficient block size for our applications without excessive area and power requirements. Figure 5 depicts application performance on our FPGA architecture versus data cache size. Data cache associativity is fixed at four. Results show that a 512-byte cache is sufficient for our working sets.

Figure 6 depicts application performance versus data cache associativity. We see that a 4-way set associativity is required in order to avoid conflict misses. An 8-way set associative cache eliminates slightly more conflict misses than a four-way but is not worth the added complexity.

Since the application data-access characteristics are similar for all three architectures, this data cache size and configuration is used throughout. The Scalar and VLIW architectures use split instruction/data caches. For these architectures, a similar process to that described for data cache sizing was performed in order to determine the size and nature of the required instruction caches. The results are summarized in Table 3.

5.2 Architecture Comparison

Using the cache configurations described in the previous section, we compare the three Active Page architectures. The result of this comparison is displayed in Figure 7, where we

Parameter	Single-Issue (Scalar)		Multi-Issue (VLIW)		Reconfigurable (FPGA)	
	Reference	Variation	Reference	Variation	Reference	Variation
CPU Clock	1 GHz	–	1 GHz	–	1 GHz	–
L1 I-Cache	64K bytes	–	64K bytes	–	64K bytes	–
L1 D-Cache	64K bytes	–	64K bytes	–	64K bytes	–
L2 Cache	1M byte	–	1M byte	–	1M byte	–
External Memory Speed	50 ns	–	50 ns	–	50 ns	–
Memory / Processor Bus	PC-100	–	PC-100	–	PC-100	–
L2 Line size	128 bytes	–	128 bytes	–	128 bytes	–
AP Line size / Bandwidth	256 bits	–	256 bits	–	256 bits	–
AP Cache Ports	2 Data, 2 Inst	–	2 Data, 3 Inst	–	2 Data	–
AP Cache Organization	Split I/D	–	Split I/D	–	Data	
AP I-Cache size	512b	64b - 4KB	1KB	64B - 4KB	-	-
AP D-Cache size	512b	128b - 4KB	512b	128B - 4KB	512b	128b - 4KB
AP Cache Associativity	I: 2,D: 4	1 – 8	I: 2,D: 4	1 – 8	D: 4	1 – 8
AP Logic Element	Mini-RISC	–	Mini-VLIW-4	2, 4, 8 width	FPGA 256 LEs	–
AP Pipeline stages	5	–	2	2, 4	–	–
AP Logic Clock	100 MHz	50–500 MHz	100 MHz	50–500 MHz	100 MHz	50–500 MHz

Table 2: Active Page reference parameters

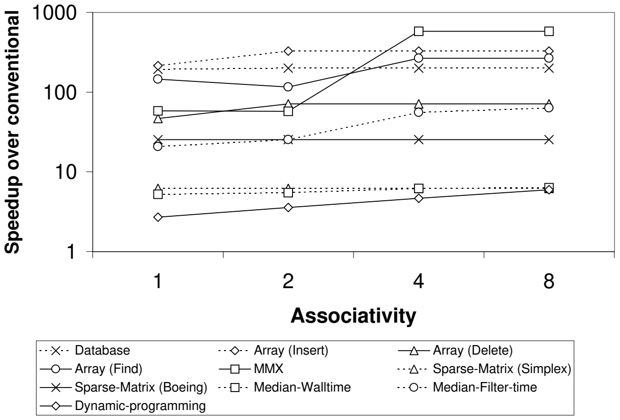

Figure 6: Data cache associativity

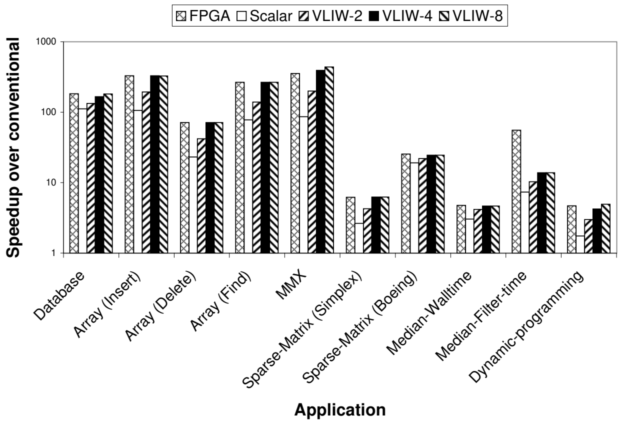

Figure 7: Baseline performance relative to conventional memory

Architecture	Instruction Cache	Data Cache
FPGA	-	512b/4-way
Scalar	512b/2-way	512b/4-way
VLIW 2 wide	512b/2-way	512b/4-way
VLIW 4 wide	1024b/2-way	512b/4-way
VLIW 8 wide	2048b/2-way	512b/4-way

Table 3: Active Page architecture cache configurations

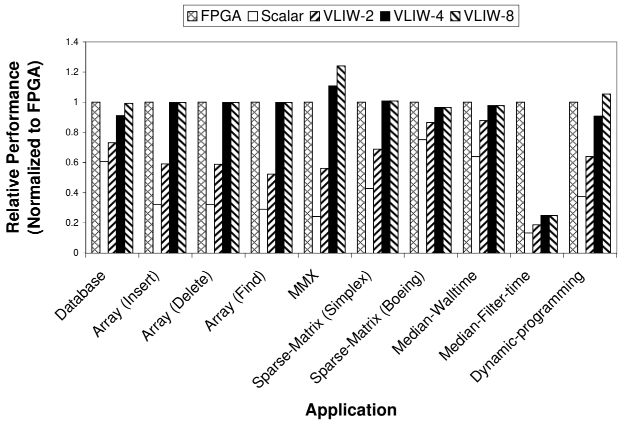

Figure 8: Baseline performance normalized to FPGA-RAM

plot the speedup over conventional memory for each architecture. We can see that all configurations exhibit substantial performance gains over a conventional system. To highlight the differences between architectures, we normalize the same data to FPGA performance in Figure 8. While the Scalar processor implementation does not achieve the same level of performance as the FPGA implementation, the VLIW processor does. In fact, with a VLIW width of four, performance is within 10% of the FPGA for the majority of applications. As the VLIW width is extended to eight, most applications do not see additional improvement. The exceptions are MMX and dynamic-programming. On the other hand, neither the Scalar nor VLIW Active Page architectures perform well on the median filtering application. We identify four principal architectural attributes that affect performance on the various applications.

- *Limited data ports in cache*: For most applications, as the VLIW width expands to four or eight, the application achieves performance equal to the FPGA implementation. This is primarily due to the fact that both architectures efficiently utilize the available data cache ports. Several of the applications can benefit from more data ports.

- *Limited processor branching capabilities*: Our VLIW hardware model permits up to one unconditional and two conditional branches to be executed in the same instruction. This restriction comes into play on both the eight-wide and four-wide VLIW processors. The database search application could benefit from the addition of one conditional branch instruction, whereas the median-filtering application could achieve better performance with more advanced control instructions.

- *Limited area*: The FPGA circuits were severely limited by the number of logic elements and routing resources available. Power and area restrict the number of logic elements to 256 per page. This means that simpler state machines must be used where perhaps a more complex one could have achieved higher performance. This is the reason that the MMX application performs better with a four-wide VLIW processor than the FPGA logic. With infinite area available, such limitations on the FPGA may be removed. We observe that as long as a data-flow graph within the FPGA can be described succinctly within the VLIW's ISA, the VLIW can implement more complex data pipelining of that data-flow graph than the FPGA.

- *Specialization*: The FPGA benefits from being able to identify an application-specific ISA. This is most evident in the median-filtering application. Here, a specialized sorting network is implemented in the FPGA that performs a custom data-flow sort. Such an instruction would not be practical in a general purpose processor functional unit. Here, the FPGA achieves a clear performance benefit over a generalized processing element.

We found that applications became equally bounded by available control-flow *and* data cache port facilities. Hence, a more optimized architecture for an eight-wide VLIW processor will require both additional branching and load / store capabilities in order to extract the available performance provided by the added functional unit width. It is not clear

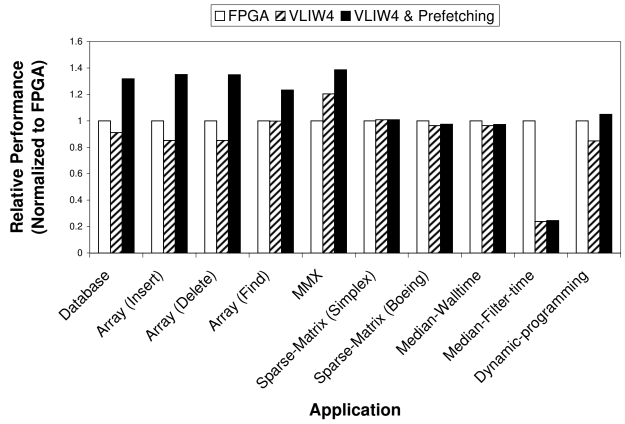

Figure 9: Speedup of FPGA compared to VLIW4 with and without data prefetching

whether these facilities can be utilized by all applications without more advanced compiler techniques such as trace scheduling.

Figure 8 shows that a four-wide VLIW processor achieves 90-100% of the performance of an eight-wide VLIW processor. For this reason, we now focus our study on the four-wide VLIW, and consider the eight-wide too area inefficient for practical use.

5.3 Data Prefetching

Prefetching is a common processor technique used to hide the adverse effects of long memory-latency operations. Within an intelligent memory system, we find that a similar approach improves performance. An explicit data-prefetch instruction was implemented in the processor. Prefetches are non-binding requests for the local data cache to prefetch cache lines, i.e., a non-blocking, non-binding load. Well-placed prefetching requests will increase the data cache hit rate. Poorly placed prefetches may degrade performance because the DRAM subarray is busy when a processor cache miss occurs. Although processor loads by-pass prefetching requests, if a prefetch request is in progress it is not terminated early, and thus the processor may have to wait for the current prefetch to complete before the load is issued. Prefetching data too early or prefetching unused data pollutes the cache, expelling data that may be used before the newly prefetched data is used.

The performance benefits due to explicit data prefetching are shown in Figure 9. Here, application performance is normalized to that of the FPGA. Due to area constraints, the FPGA can not provide prefetching. We observe that for certain highly regular streaming applications such as database, array, and MMX, prefetching can improve performance dramatically. Prefetching is also useful for more complex control flow oriented applications such as median filtering and dynamic programming, but performance gains are less significant. Within these applications, destructive data cache interference occurred periodically due to the prefetch requests interfering with ordinary cache misses. Although not explored in this paper, larger data caches may help alleviate part of this performance problem.

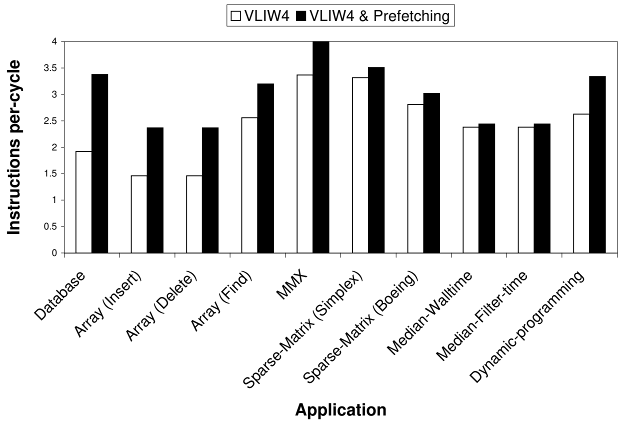

Figure 10: IPC of VLIW 4 wide with and without prefetching

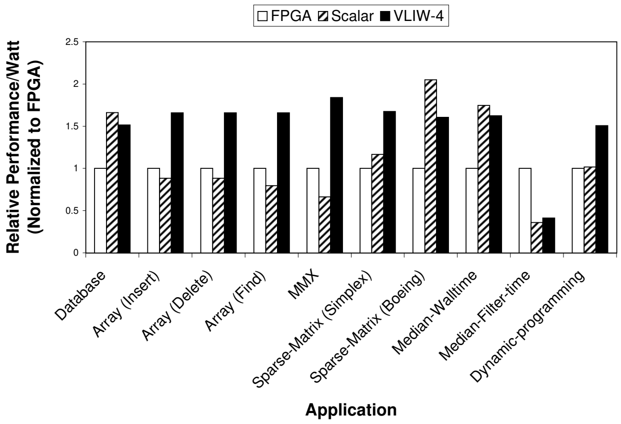

Figure 11: Performance/Power

We observe that the four-wide VLIW processor with data-prefetching is actually faster for most applications than the FPGA. Furthermore, with the addition of prefetching and a compiler that can effectively schedule the prefetches, the performance of the VLIW should reach or exceed a comparable superscalar implementation. These codes have very regular data access patterns, so prefetches are not difficult to place. This provides the slip that superscalars use to mask load latencies. In applications that use multiple branches per line, the VLIW could exceed the superscalar performance. The superscalar would need to perform the branches in order, whereas the VLIW can place up to three branches in a single instruction. Branch prediction and speculation might help, but the resulting superscalar would be prohibitively large in its area and power requirements.

5.3.1 Performance Limits of VLIW

Throughout this study, we have placed two artificial limitations on the VLIW architecture. First, we have limited the control-flow capabilities to one unconditional and two conditional branch instructions per cycle. Second, we have limited the data cache access ports for all Active Page implementations to two. We limited ports to provide a uniform means of measuring the ability of each architecture to extract instruction level parallelism from the underlying algorithm. However, here we would like to explore the potential of the VLIW architecture to utilize additional cache ports. Area constraints do not permit the FPGA to utilize these additional resources.

Figure 10 depicts instructions per cycle for each application executing on a four-wide VLIW Active Page architecture with and without prefetching enabled. We see that prefetching does contribute to increases in IPC from 5-70%, depending upon the application.

If we focus on the IPC of the prefetching implementation of each application and look back at that application's source, we can isolate specific application components that may benefit from added VLIW hardware features. Of principal concern is the number of data cache ports and branching capabilities of the processor. Highly regular data-driven applications will benefit from additional cache ports, whereas their non-uniform data-dependent counterparts may benefit from additional branching capabilities. To isolate these potential future performance gains, we examine each application individually.

- *MPEG-MMX* focuses on the packed-add MMX instruction. During execution of this instruction on a data-set, each iteration of its inner loop loads two values, stores one, and contains one conditional branch. Although compiler generated code is unlikely to utilize additional data cache ports and branching capabilities, efficient hand-optimized implementations can be constructed.

- *Median* sorts sets of numbers and finds the median. The VLIW is control dependent when sorting the elements. If one constructs a decision tree for the sorting, the height is log(n). This means that log(n) serial comparisons must occur in the worst case. In the FPGA, several of these comparisons can be performed during the same cycle, whereas the VLIW must have separate instructions depending on the outcomes of the previous comparisons. Additional performance gains will only be achieved by additional deep loop-level software pipelining, and this will introduce strain on both the data cache ports and processor control flow capabilities.

- *Dynamic Programming* has several factors which inhibit its ability to utilize all instructions within the VLIW program word. Chief among these is the fact that the data comes from four separate data streams, increasing the likelihood that prefetching will not occur in time for all of the data values. The amount of data that has to be fetched and compared to determine the next block is also a limiting factor. Thus additional data ports will only bring marginal performance improvements.

- *Matrix* is control bound. It is limited by the number of branches allowed on a line. Additional branching capabilities within the processor would reduce the number of instructions inside the inner core loop. Currently, the inner loop must branch to one of four locations depending upon the input data. This multi-way branch could be collapsed into a single instruction with additional VLIW branching capabilities.

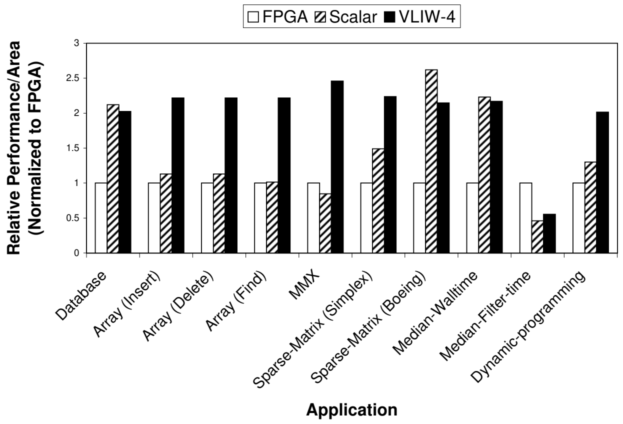

Figure 12: Performance/Area

- *DataBase* is also limited by the number of branches allowed per instruction. In this application a last name is being searched for within a simulated database of records. This comparison requires at least three conditional branches per comparison, plus additional branches for the loop conditions. Although software techniques coupled with prefetching can be used to remove data stalls, the application remains control-bound. Additional data ports would not significantly increase performance.

- *Array insert/delete* shifts binary data within memory. The application is clearly data cache port bound and would benefit from additional cache access ports. Additional control-flow capabilities within the processor are not required.

- *Array find* is both data and control-bound. Application performance here is data dependent. Data with few matches to the search key will benefit from additional data ports and control-flow instructions, while data with many matches will see little, if any, benefit.

5.4 Performance vs. Power and Area

The advantages of our VLIW Active Page implementation become more pronounced as we examine power and area requirements. Table 4 compares several of our implementations. Estimates for our FPGA design are based upon published specifications for Altera's FLEX architecture [Alt] in 0.25 micron technology. Power and area analysis of the various processor designs were performed using the Synopsys tools. Each design was modeled in VHDL and synthesized using LSI Logic's G10-p 0.25 micron process. Constraint driven synthesis with high map effort was used to optimize the designs. Once the designs were synthesized, back annotation was used to gather toggle information for the power analysis. Our figures are reasonably consistent with recent processors in 0.25 micron technology [Cho98] [Shi98]. Note that synthesis makes our results conservative. Custom design would give our processor designs a larger advantage over state-of-the-art FPGA technologies.

Architecture	Area (mm^2)	Power (mW)
FPGA	1.86	375
Scalar 5 - stage	0.534	137.5
VLIW-4 4 - stage	1.295	306.5
VLIW-4 2 - stage	0.838	225.8

Table 4: Active Page architecture power and area comparison (per-Active page). All data is from a 3.3v, 0.25 micron standard-cell process

Figure 11 depicts application performance versus power use for three of our Active Page architectures. These are the FPGA, the single-issue five-stage pipeline processor, and the four-wide (two stage) VLIW processor. Figure 12 depicts application performance versus relative computational logic area. Clearly, the FPGA has poor performance per-watt and performance per-area when compared to the Scalar and VLIW implementations. Furthermore, Table 4 shows that the FPGA has a higher absolute power and area cost.

It is clear that for most applications, the VLIW implementation is the most efficent implementation in terms of performance per watt and per area. For highly control-bound applications, the Scalar attains comparable efficiency. This makes intuitive sense, since control-bound applications cannot make as efficient use of the available VLIW width.

Finally, we also note that the two-stage pipeline implementations of the VLIW processor is more efficient in terms of power and area than the four stage pipeline versions. This saving is more pronounced for wider VLIW processors due to the increasingly large forwarding networks needed to prevent data hazards.

5.5 Technology Scaling

Intelligent memory technology is a moving target. The key to this study is the *relative* performance, area, and power of our Active Page alternatives. One issue not yet discussed is *processor saturation*. Since Active Page applications depend upon both the main processor and the Active Pages, the memory computation may complete before the processor is ready. Figure 13 illustrates the effect of speeding up our Active Page logic while holding the rest of the system parameters constant. We can see that the sparse matrix application is saturated; Performance is limited by the main processor as we increase Active Page logic speed. This graph also allows us to compare architectures with differing clock speeds. For example, we assume power constraints which prevent the scalar and VLIW implementations from being clocked much faster than our FPGA implementations in equivalent technologies. If these constraints were relaxed in the future, the scalar and VLIW clocks could scale well beyond the FPGA clock – a cost of reconfigurability.

Our power and area figures assume a .25 micron technology. Power will decrease as the square of future process shrinks. We expect future process technologies to reduce Active Page power consumption dramatically and make commodity packaging of Active Page memory practical.

6 Related Work

DRAM densities have made intelligent memory designs attractive as commodity components. Intelligent memory, how-

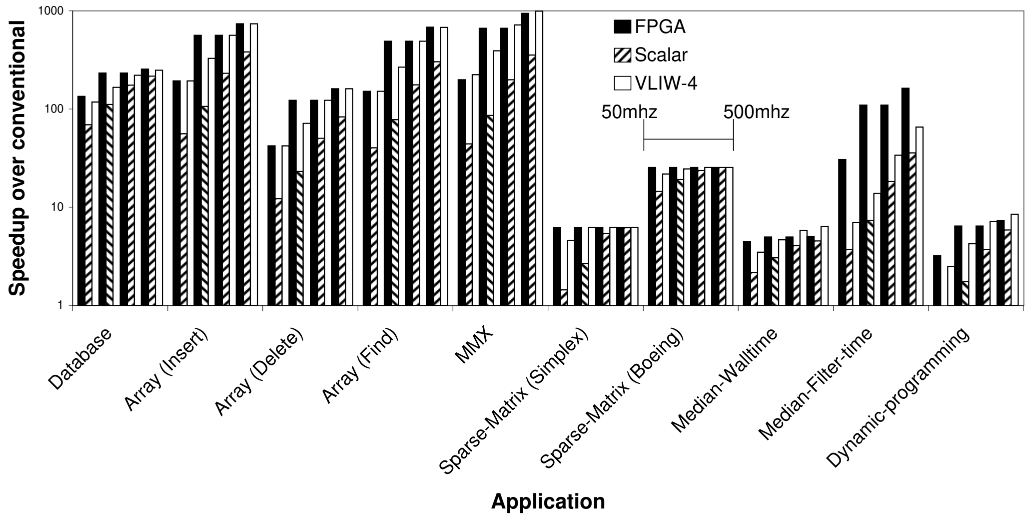

Figure 13: Performance scaling with respect to Active Page computational logic clock

ever, was proposed well before the current commodity thrust. The SWIM project [ACK94] combined reconfigurable logic and memory to perform fast protocol computations. The J-Machine integrated processor, memory, and network router in a single chip to form building blocks for a fine-grained multiprocessor [NWD93]. The RAW [L+98], MORPH [CG96], and RaPiD [E+97] projects continue to explore the use of reconfigurable technology to exploit parallelism. The HPAM project [MEFT96] takes a hierarchical approach to intelligent memory.

The Impulse project [C+99] has similar goals to Active Pages but focuses on adding address manipulation functions to the memory controller. Their applications, such as gather-scatter for sparse matrix multiplied by dense vector, are also enhanced by more efficiently feeding the microprocessor with data. All of our applications, however, require some small computations which can not be supported without more generalized computation in the memory system than provided by Impulse.

Intelligent disks [AUS98] [G+98] have also been proposed. These systems are meant for streaming applications that do not fit in main memory. After processing by an intelligent disk, data may still benefit from computations in an intelligent memory. A unified view of intelligence at all levels of the memory hierarchy may eventually prove useful.

7 Future Work

For Active Pages to become a successful commodity architecture, the application partitioning process must be automated. Current work uses hand-coded libraries which can be called from conventional code. Ideally, a compiler would take high-level source code and divide the computation into processor code and Active Page functions, optimizing for memory bandwidth, synchronization, and parallelism to reduce execution time. This partitioning problem is very similar to that encountered in hardware-software co-design systems [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.

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

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

8 Conclusion

Active Pages is a page-based computational model for intelligent memory which can lead to substantial performance benefits. This study has explored several alternative implementations of Active Pages. We found that reconfigurable implementations perform well due to fine-grained parallelism in the memory system rather than logic specialization. This finding led us to design a 4-wide VLIW processor that exploits this parallelism without the high cost of reconfiguration. Unlike conventional VLIW processors, our Active Page VLIW processor uses a short pipeline to optimize performance under the strict power and area constraints of the DRAM environment. Our simulations demonstrate that the VLIW Active Pages perform as well as their reconfigurable counterparts but with substantial additional benefits in power, area, and programmability.

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.

[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.

[Alt] Altera. FLEX 6000. http://www.altera.com/html/products/f6k.html.

[Alt98] Altera Corporation. *FLEX 6000 Programmable Logic Device Family Datasheet (Ver 3.02)*, March 1998.

[AUS98] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

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

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

[C+99] J. Carter et al. Impulse: Building a smarter memory controller. In *HPCA*, January 1999.

[CG96] Andrew A. Chien and Rajesh K. Gupta. Morph: A system architecture for robust high performance using customization. In *Frontiers '96*, 1996.

[Cho98] J. Choquette. Genesis microprocessor. In *Hot Chips 10*, pages 49–58, August 1998.

[CNO+88] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *IEEE Transactions on Computers*, volume C-37, pages 967–979. 1988.

[E+97] C. Ebeling et al. Mapping applications to the rapid configurable architecture. In *Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, April 1997.

[Ell86] J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, MIT, 1986.

[F+97] Richard Fromm et al. The energy efficiency of IRAM architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, 1997.

[Fis83] J. Fisher. Very long instruction word architectures and the ELI-512. In *10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.

[G+98] G. Gibson et al. A cost-effective high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

[G+99] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In *26th Annual International Symposium on Computer Architecture*, pages 28–39, 1999.

[GHI95] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: the Terasys massively parallel PIM array. *Computer*, 28(4):23–31, April 1995.

[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.

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

[HYYS96] Edgar Holmann, Toyohiko Yoshida, Akira Yamada, and Yukihiko Shimazu. VLIW processor for multimedia applications. In *Hot Chips 8*, August 1996.

[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.

[KH92] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[L+98] W. Lee et al. Space-time scheduling of instruction-level parallelism on a Raw machine. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

[LFK+93] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[MEFT96] Z. Miled, R. Eigenmann, J. Fortes, and V. Taylor. Hierarchical processors-and-memory architecture for high performance computing. In *Sixth Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

[MSM97] K. Murakami, S. Shirakawa, and H. Miyajima. Parallel processing RAM chip with 256Mb DRAM and quad processors. In *ISSCC Digest of Technical Papers*, 1997.

[NWD93] M. Noakes, D. Wallach, and W. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture 1993*, pages 224–235, San Diego, CA, May 1993. ACM.

[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.

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

[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.

[Sem97] Semiconductor Industry Association. The national technology roadmap for semiconductors. http://www.sematech.org/public/roadmap/, 1997.

[Ses98] N. Seshan. High VelociTI processing. *IEEE Signal Processing Magazine*, pages 86–101, March 1998.

[Shi98] Toru Shimizu. M32Rx/D – a single chip microcontroller with a high capacity 4MB internal DRAM. In *Hot Chips 10*, August 1998.

[SRD96] Gerrit Slavenburg, Selliah Rathnam, and Henk Dijkstra. The Trimedia TM-1 PCI VLIW mediaprocessor. In *Hot Chips 8*, August 1996.