Active Pages: Memory-centric Computation

Diana Franklin
Department of Computer Science
California Polytechnic State University

Although field-programmable gate arrays (FPGAs) excel at tailoring the computation and interconnect to an application's needs, we can go one step further. In many applications, regardless of the speed of the computation, memory performance always will be the limiting factor. This problem, referred to as the *memory wall*, is broken up into two parts—memory latency and bandwidth. For large-scale data-parallel applications, the computation can be moved to memory. This allows for both parallel computation and increased bandwidth. The replication of small computation units provides parallelism, and the sum of their memory ports provides increased bandwidth. Because they are located in memory, there is no shared-bus resource to serialize communication.

One such system, Active Pages, places computation with each page of DRAM. It is unique in that it targets the commodity DRAM market. This decision has both advantages and disadvantages. One advantage is that it supports both data streaming and general-purpose computation, and the computational resources scale automatically with memory allocation. One disadvantage is that, to keep costs low, there is no additional interconnect, and parallelism is only at the page level.

Many of the characteristics of Active Pages are present in any memory-centric system. This case study explores several characteristics of the Active Pages design. It begins, in Section 35.1, with an overview of the Active Pages architecture and programming model. Section 35.2 shows the performance potential of a scalable, memory-centric design. Section 35.3 then looks at how this scaling of computational resources, but not the interconnect resources, affects the asymptotic properties of several algorithms. Finally, Sections 35.4 and 35.5, explore the parallelism properties and the defect tolerance provided by the Active Pages design. Active Pages is just one of many projects in this realm, and Section 35.6 presents related work, followed by some conclusions in Section 35.7.

35.1 ACTIVE PAGES

This section gives a brief description of the Active Pages system. We present three aspects of the design: the hardware design, the interface between Active

Pages and the Central Processor, and the programming model that arises naturally from the design and interface.

35.1.1 DRAM Hardware Design

High-density DRAMs are divided into subarrays, complete with row and column decoders, to minimize column capacitance and decrease power consumption [1]. The proposed Active Pages implementation exploits this natural structure, treating each subarray as an Active Page. As shown in Figure 35.1, a small computational unit and cache—a *Page Processor* and *Page Cache*—are embedded next to each subarray to implement Active Page functions [2]. Using commodity 1-Gb DRAM technology as a target [3], we expect subarray size to be 512 KB and the embedded processing to consume less than 31 percent of the chip area.

To minimize DRAM modification and reduce hardware overhead, the Active Pages implementations do not provide hardware support for communication between Active Pages. If two Active Pages need to share data, the Central Processor reads the data from one and writes to the other. The disadvantage of this *process-mediated approach* is that interpage communication must be infrequent to maintain performance with a single processor.

35.1.2 Hardware Interface

To interface with the Central Processor, Active Pages leverage conventional page-based memory mechanisms to "virtualize" hardware for memory-based computation. Computations for each page can be suspended, restarted, and even swapped to disk. Computations for several pages can be multiplexed on a single embedded processing element.

Further, Active Pages use the same interface as conventional memory systems. Active Pages data are modified with conventional memory reads and writes; Active Pages functions are invoked through memory-mapped writes. Synchronization is accomplished through user-defined memory locations.

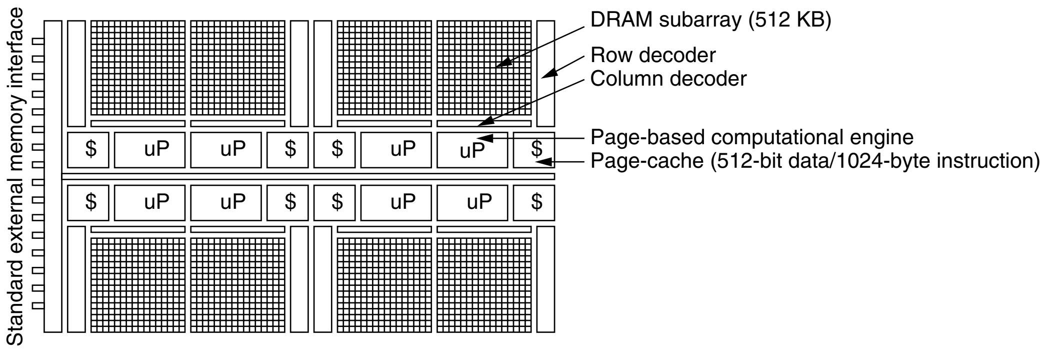
Standard external memory interface
DRAM subarray (512 KB)
Row decoder
Column decoder
Page-based computational engine
Page-cache (512-bit data/1024-byte instruction)
$
uP

FIGURE 35.1 ■ The Active Pages architecture (8 pages).

35.1.3 Programming Model

The programming model of Active Pages was determined by several design decisions. First, communication between Active Pages and the Central Processor is accomplished through traditional reads and writes, allowing the Central Processor to operate on Active Pages data just as it does on any other data. Second, Active Pages were intended for commodity DRAM systems, which may be running general-purpose applications. Thus, we could not assume a traditional data parallel, streaming model. Third, there is no interconnect between Active Pages processors. The model needs to limit the Pages to their own data, with no knowledge of neighboring cells. Finally, each Active Page has computation associated with it. This is a direct association of data with computation. For these two reasons, the model of computation here is object-oriented programming.

To program a Page Processor, the programmer creates an object in C++. The choice of C++ is not critical; it is used because it has no runtime system associated with it and has well-defined interfaces for object manipulation. The 512 KB allocated to each Page Processor is divided between code, stack, and data. These 512 KB, larger-than-typical operating systems' virtual pages are referred to as superpages. The code must fit within the code segment, and the data size of the object is padded appropriately.

The operating system (OS) is responsible for allocating Active Pages memory and loading the code into the correct region. The Page Processor begins on activation, first performing any initialization similarly to a C++ object constructor, and then polling a variable waiting for an invocation of a function. To maintain pin compatibility, all Active Pages functions are designed to use conventional reads and writes. The Central Processor invokes Active Pages functions by writing the parameters into appropriate places in the Active Pages memory. The Central Processor then changes the `Running` variable, on which the Page Processor is polling, indicating which function to execute next.

When the Page Processor has completed the function, it resets the looping variable (`Running`) and waits for the next invocation. Figure 35.2 shows the object declaration and implementation for execution on a Page Processor for LCS. More details on the LCS algorithm can be found in Section 35.3.3. In the LCS algorithm, the application requires only a single function, so the event loop is not actually necessary. It is shown, however, to illustrate how an application with many functions would use the Central Processor to invoke functions on the Page Processors. The main function run on the Central Processor is not shown. The Central Processor can poll the `Running` variable to determine whether a Page Processor has completed a particular function.

35.2 PERFORMANCE RESULTS

Now that we have an idea of what the Active Pages architecture looks like and how it is programmed, this section presents performance results for several applications using a simulated Active Pages system. A more detailed study can be found in Oskin et al. [4].

```
Class LCS{
  //int    CodeAndStack[8192];   // added by compiler
  public:
    int Running, Data[WIDTH-1][LENGTH-1];
    char X[WIDTH], Y[LENGTH];
    LCS(){ Running = AP_WAIT; }
    void Start();
    void DoLCS();
} ;
void LCS::DoLCS() {
  int i, j;
  for(i=1;i<LENGTH;i++) // row 0, column 0 initialized by Central Processor
     for(j=1;j<WIDTH;j++) {
        if (X[i] == Y[j])
             Data[i][j] = Data[i-1][j-1] + 1;
        else if (Data[i-1][j] > Data[i][j-1])
             Data[i][j] = Data[i-1][j];
        else
             Data[i][j] = Data[i][j-1];
  }
}
void LCS::Start() {
    volatile int *act = &(Running);
    while(*act != AP_STOP) {
       while(*act == AP_WAIT) ;   // wait for Central Processor
       switch (*act) {
        case(AP_LCS):
           DoLCS(Val);
           *act = AP_WAIT;  // it is done
           break;
      }
   }
}
```

FIGURE 35.2 ■ A code example of an Active Pages object. Each Page Processor initializes its own space on allocation using the constructor. The Central Processor starts the Page Processor by writing to the `Running` variable. When the call is finished, the Page Processor sets `Running` back to `AP_WAIT`.

To estimate the performance of Active Pages configurations, each Active Pages function was hand-coded in a high-level circuit-description language, such as VHDL (see Chapter 6 and [5]), and synthesized to an Altera 10K FPGA. The mapping was carried out all the way to placed and routed designs [6].

To demonstrate effective partitioning of applications between the Central Processor and Active Pages, we chose a range of applications representing both memory- and processor-centric partitioning. Table 35.1 summarizes the attributes of these applications.

35.2.1 Speedup over Conventional Systems

To evaluate performance of the Active Pages memory system, each application was executed on a range of problem sizes. The speedup of the applications

TABLE 35.1 ■ Summary of the partitioning of applications between the Central Processor and Active Pages

Name	Application	Central Processor computation	Active Pages computation
	Memory-centric applications		
Array	C++ standard template library array class	C++ code using array class cross-page moves	Array insert, delete, and find
Database	Address database	Initiates queries summarizes results	Searches unindexed data
Median	Median filter for images	Image I/O	Median of neighboring pixels
Dynamic program	Protein sequence matching	Backtracking	Compute MINs and fills table
	Processor-centric applications		
Matrix	Matrix multiply for Simplex and finite element	Floating-point multiplies	Index comparison and data gathering and scattering
MPEG-MMX	MPEG decoder using MMX instructions	MMX dispatch Discrete cosine transform	MMX instructions

running on an Active Pages memory system compared to a conventional memory system is shown in Figure 35.3. Each application was run on a range of problem sizes, given in terms of number of Active Pages (512-KB superpages). The following are two primary observations about this graph.

First, the performance results qualitatively scale as expected. This shows the advantage of memory-centric computation. We observe that most applications show little growth in speedup as data size grows within the subpage region (below one page). In this region, Active Pages applications have little parallelism to offset activation costs. When leaving this region, however, we enter the scalable region and see that performance on all applications grows as data size increases. Four applications—database, MMX, matrix-simplex, matrix-boeing, and median-filtering—also reach the saturated region. Here, Active Pages performance is limited by the progress of the Central Processor. This limitation may be because of either too much work for a given-speed Central Processor or too much data travelling between the Central Processor and Active Pages across the memory bus. Performance can actually decrease as coordination costs dominate performance. Given a large enough problem size, all applications would eventually reach the saturated region.

Second, we see that the array-delete primitive performs poorly in the subpage region. This is because of the difference between the FPGA implementation and the instruction set used to implement the Central Processor. The Central Processor's instruction set is especially well suited for the array-delete primitive. Thus, unless there is sufficient parallelism to justify using Active Pages, it is faster to use the Central Processor. So, for small deletes, we use only the Central Processor. This benchmark was a combination of small deletes and large deletes.

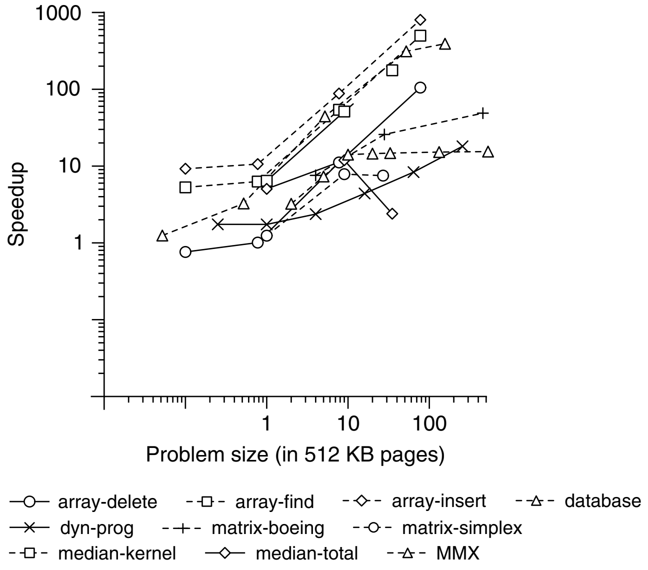

FIGURE 35.3 ■ Active Pages speedup as problem size varies.

As problem size grows, and the Central Processor is used for both the coordination of large deletes and the complete execution of small deletes, the Central Processor becomes the limiting factor in performance and the performance gets closer to that of the uniprocessor. This shows an interesting tradeoff between the FPGA and the Central Processor. Some computations, though not many, will perform better on the Central Processor. If this coincides with a part of the application that does not require parallelism, then the advantage of the memory-centric FPGA implementation will be reduced.

35.2.2 Processor–Memory Nonoverlap

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

As data size grows in an Active Pages application, so does the load on the Central Processor. We measure the remaining capacity of a Central Processor to handle this load with a metric, *processor–memory nonoverlap* time. Nonoverlap is the time the Central Processor spends waiting for the memory system and can be used to estimate the boundary between the scalable and saturated regions of application performance.

The relative percentage of time the Central Processor is stalled, waiting for memory system computation, is shown in Figure 35.4. As described in

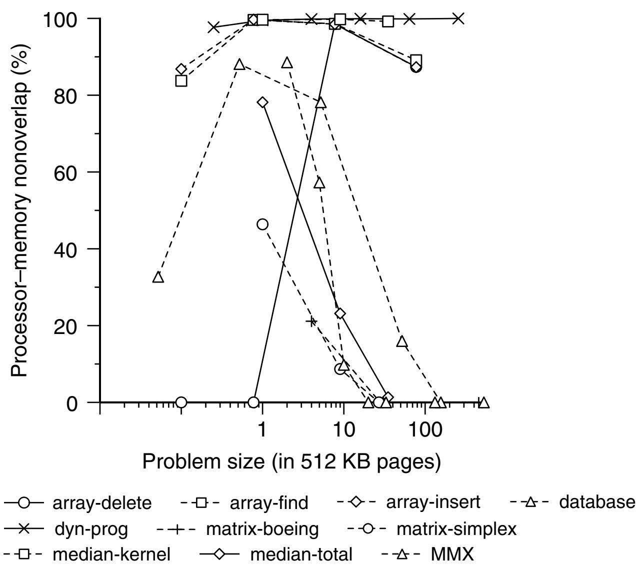

FIGURE 35.4 ■ The percent of cycles that the Central Processor is stalled on Active Pages as problem size varies.

the previous section, the applications that reached the saturated region of speedup were database, matrix-simplex, matrix-boeing, and median-filtering. As Figure 35.4 shows, these applications also reach a point of complete processor–memory overlap.

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

The dynamic programming example maintains a very high processor–memory nonoverlap; however, preliminary results indicate that processor-mediated communication required by the Active Pages memory system eventually dominates performance. This occurs for extremely large problems that are well beyond the range of problem sizes presented in this study. Dedicating more resources to the interconnect increases the range of problems that Active Pages can help solve.

35.2.3 Summary

Memory-centric computation provides a scalable source of performance for large-scale applications. Active Pages provides a large number of simple, reconfigurable computational elements that can achieve speedups up to 1000 times faster than conventional systems. Systems with rich interconnects have the potential for scalable gains on an even wider range of applications.

35.3 ALGORITHMIC COMPLEXITY

Although the simulated results show great promise, to truly understand how Active Pages improves runtimes as problem sizes grow, we need to explore asymptotic properties of algorithms in conventional systems as well as Active Pages systems [7]. For this study, we use a set of kernels whose asymptotic properties are well known in algorithmic literature.

While it is unrealistic to expect the number of processors in a conventional multiprocessor to scale arbitrarily, the amount of DRAM in a system is expected to scale with problem size for a majority of problems. With Active Pages DRAMs, computational hardware also scales. This scaling provides parallelism that can improve asymptotic performance. Table 35.2 gives a preview of such gains for a variety of algorithms. Note that Active Pages execution times rely on the optimal page size given in the table. In practice, we expect Active Pages hardware to support a small range of page sizes designed to support target applications and problem sizes.

The challenge in the analysis is to take communication costs into account. In any system, the interconnect will affect the asymptotic properties of the performance as the problem scales. Active Pages, in particular, requires careful consideration of the communication between Page Processors as well as between the Central Processor and the Page Processors. The partitioned computations and restricted communication model here differ substantially from traditional parallel models such as PRAM [8]. This section presents an analysis of each algorithm that considers these issues. These analyses are also validated with simulation results.

TABLE 35.2 ■ Algorithmic complexity (summary)

Application	Conventional	Execution time within Active Pages	Page size
Array insert	$O(n)$	$O(\sqrt{n})$	$O(\sqrt{n})$
2D LCS	$O(n^2)$	$O(n\sqrt{n})$	$O(n)$
3D LCS	$O(n^3)$	$O(n^{7/3})$	$O(n^2)$
All-pairs shortest path	$O(n^3)$	$O(n^{7/3})$	$O(n^{4/3})$
Sorting	$O(n \cdot \log_2(n))$	$O(n \cdot \log_2(\log_2(n)))$	$O(n/z)$ where $n = z \cdot e^z$
Volume rendering	$O(n^3)$	$O(n^{5/2})$	$O(n^{3/2})$

35.3.1 Algorithms

Active Pages can dramatically improve the performance of many algorithms. This section maps several common algorithms to an Active Pages system and analyzes performance gains. Figure 35.5 introduces the notation used here. With these conventions, we analyze the worst-case execution time of the algorithms: insertion of an element into a linear array of elements, longest common subsequence of two- and three-dimensional sequences using a dynamic programming formulation, all-pairs shortest path using a dynamic programming formulation, sorting of a linear array of elements, and volume rendering using ray-tracing and linear absorption coefficients [7, 9].

Each analysis is provided by first presenting a general model for the algorithm's execution time. Next, various model-specific parameters are assumed to be constants. After this simplification, the derivative of execution time with respect to page size is used to find an optimal page size. This page size is then substituted back into the model, and execution time is expressed again as a function of problem size.

These results are then validated with a high-level simulator. The simulator models Active Pages execution using parameters based on execution of the cycle-level simulator. The parameters used are given in Table 35.3. *Typical* parameters correspond to the target architecture studied here and often exhibit better performance than a purely asymptotic analysis would suggest. *Asymptotic* parameters emphasize the dominant terms in asymptotic performance while remaining within realistic problem sizes. These exaggerated parameters are used to validate the more conservative analyses.

Table 35.3 summarizes the parameters used in the high-level simulator. T_a is the amount of time required by the processor to invoke a function on a memory-based

n	is the size of the input.
p	is the number of data elements in an Active Page.
q	is a problem-specific function of p that is used for most algorithms to define p. For instance, for dynamic programming algorithms where a two-dimensional result set is generated, it is convenient to describe p as equal to $p=q^2$.
k	is a function of the number of Active Pages used for the problem—usually $k=n/q$.

FIGURE 35.5 ■ The notation used for algorithmic analysis.

TABLE 35.3 ■ Summary of simulation parameters

Parameter	APSP*	Sort	Array insert	LCS*	LCS3	Render
Activation time (T_a)	100/0	0	2058	100/100	100	100
Central Processor per-page processing time (T_p)	–	1	387	–	–	5
Page processing per-element processing time (T_c)	10/10	1	2	–	10	10
Fixed communication overhead (T_{sa})	1/1	–	–	10/10	1	–
Per-element communication cost (T_{sb})	1/1	–	–	1/100	1	–

∗ Typical/asymptotic.

processor. This includes setup, argument passing, and invocation. This constant is per page. T_p is the amount of time required by the processor to complete execution of an algorithm associated with a particular page. Generally, the "focus" of execution traverses from the Central Processor to the Active Pages and then back again. This may proceed many times and involve overlap throughout the execution of the algorithm. However, for the analysis presented here the focus is on a single set of transitions from host to memory and back. Hence, T_p is the time spent by the Central Processor per page when completing the Central Processor portion of the computation for that page. T_c is the amount of time required by the memory-based processing element to compute its portion of the algorithm for a single data item within the page. For instance, on a conventional processor and memory system, an $O(n)$ algorithm requires some time, T_c, to compute the solution for each element; hence, the execution time is described as $T = T_c \cdot n$. T_{sa} is the amount of time that corresponds to the "fixed overhead" associated with each interpage communication. Inter–Active Pages communication is a necessarily expensive process, and this constant quantifies the relatively large fixed overhead associated with each such communication request. T_{sb} is the amount of time, per data item, associated with an interpage communication. Not all algorithms use interpage communication, and some use portions of T_a or T_p to perform such communication as part of activation and postprocessing, respectively.

This short section can present detailed analysis only of the array and LCS applications. We refer the reader to a technical report by Oskin et al. for the full set of analyses and results [9].

35.3.2 Array-Insert

The analysis begins with a simple array library. Specifically, we examine an insertion operation performed on an array of elements arranged in a linear fashion. A conventional system requires $O(n)$ execution to complete this task. In an Active Pages memory system, we partition these n elements into k pages, with each Active Page managing n/k elements. To insert an element at position j within the array, each Active Page from the page containing j up to the last page of the array shifts the elements up by one to make room for the new element. These shifts proceed in parallel, however, since each Active Page operates independently. Note, though, that some form of communication between pages is required to migrate elements across page boundaries. This communication is grouped within the activation portion of each Active Page. The algorithm can be expressed as shown in Figure 35.6.

```
for j=1 to k
   communicate the last element of
     page j to page j+1
   activate page j informing it to
     shift elements upward
```

FIGURE 35.6 ■ The array-insert algorithm.

The analysis begins with $s(i)$, the nonoverlap (*stall*) time for page i. The nonoverlap time, discussed in Section 35.2.2, is the amount of time spent by the processor waiting for the Active Pages memory system to finish. Essentially, this algorithm (and many other Active Pages algorithms) proceeds by having the Central Processor set up and activate memory-based processing, then wait for a page to complete computing. After the memory-based computation section is complete, the processor can return to finish its section of the computation. It turns out that quantifying how much a processor stalls while waiting for memory-based computation to complete, for traditionally linear algorithms, is an important and measurable quantity that can be used to tune applications to achieve maximum performance. We use it to quantify execution time.

Three functions—$T\hat{a}, T\hat{p}$, and $T\hat{c}$—are used to quantify portions of the execution time. These are expressed as functions because several linear-based algorithms can be mapped to an execution time analysis similar to that presented here. The functions correspond to activation time, host processor postexecution time, and per-page memory-based computation time, respectively.

For array insertion, these are essentially constant functions; hence, $Tc(i) = T_c, Ta(i) = T_a$, and $Tp(i) = T_p$. Figure 35.7 shows the timing of the array-insert operation (or any other linear-based function) on the Active Pages system using Ta, Tc, and Tp. Next, note that $\sum_{i=1}^{k} s(i) \le T_c \cdot p$ allows us to simplify execution time and take the derivative of T with respect to p. This gives us a new expression for T given the optimal value for p:

$$T = \sum_{i=1}^{k} \left[T_a + T_p + s(i)\right] = k\left(T_a + T_p\right) + \sum_{i=1}^{k} s(i)$$

$$\le k\left(T_a + T_p\right) + T_c \cdot p = \frac{n}{p}\left(T_a + T_p\right) + T_c \cdot p$$

$$\frac{dT}{dp} = \frac{-n}{p^2}\left(T_a + T_p\right) + T_c \Rightarrow p = \sqrt{\frac{n\left(T_a + T_p\right)}{T_c}}$$

$$T_{opt} = \frac{n}{p}\left(T_a + T_p\right) + T_c \cdot p = 2 \cdot \sqrt{n \cdot (T_a + T_p) \cdot T_c} = O(\sqrt{n}) \tag{35.1}$$

$$T = \sum_{i=1}^{k} \left[Ta(i) + Tp(i) + s(i)\right]$$

$$s(i) = \max \begin{cases} 0 \\ s'(i) \end{cases}$$

$$s'(i) = Tc(i) - \left(\sum_{j=i+1}^{k} Ta(j) + \sum_{j=1}^{i-1} (Tp(j) + s(j)) \right)$$

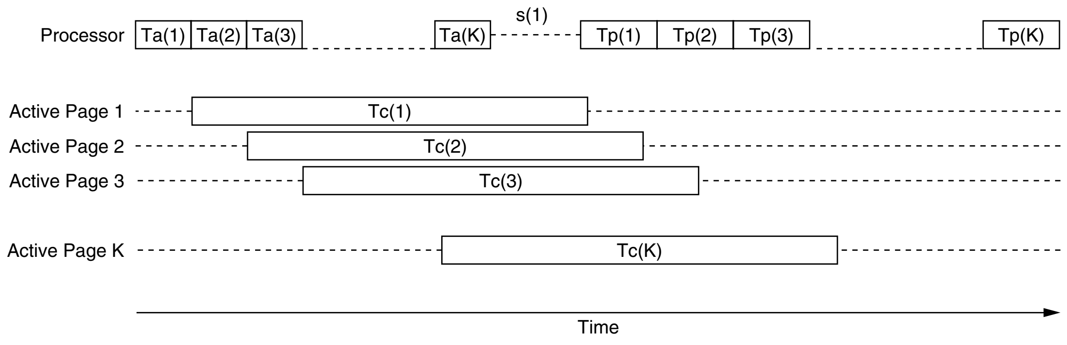

FIGURE 35.7 ■ An array-insert operation demonstrating processor and Active Page computations.

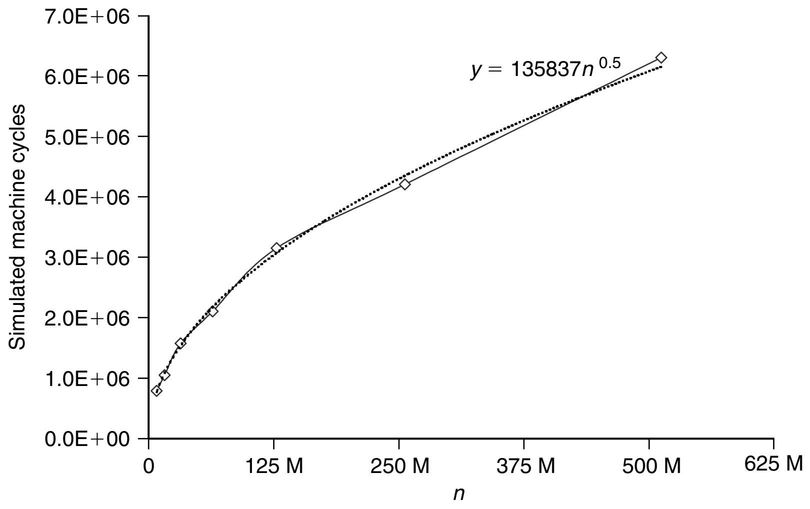

FIGURE 35.8 ■ Simulation results for the array-insert operation.

This analysis makes the conservative assumption that computation proceeds in serializable steps. First, all pages are activated; then all pages compute; finally, all pages finish and the processor performs some minimal postpage computation for each page. In reality, there is substantial overlap of these functions, and only during asymptotic performance is this serializing behavior observed. During practical application of this algorithm, the dominant term is $T_C \cdot p$, and execution time is held relatively constant. This behavior is observed until the point at which the number of pages times the activation and postpage processing per page starts to significantly approach $T_C \cdot p$. Figure 35.8 depicts simulated application performance versus problem size. As can be seen from the graph, simulated performance follows an $O(\sqrt{n})$ growth curve, as predicted by the analytical model here.

35.3.3 LCS (Two-dimensional Dynamic Programming)

Moving to a more complex algorithm, we examine a dynamic programming formulation for computing the longest common subsequence in a protein. The conventional execution time of this algorithm is $O(n^2)$. Figure 35.9 outlines the algorithm. For a more in-depth discussion of the LCS algorithm with fine-grained parallel execution in a systolic model, see Hoang [10].

Parallel execution of this algorithm proceeds in "wave-fronts," as depicted in Figure 35.10. Once the first subproblem is solved and the results have been dispatched, two other problems can immediately start computing, and when they are done, three other Active Pages can start their computation in parallel. The processor is responsible for activating a wave-front. When processor-mediated communication is used, the wave-front is uneven, with certain pages of the computation executing slightly ahead of other pages. This is because of the overlapping nature of Active Pages computation and processor activity. In the model of computation here, this overlap is very important to performance, and we take advantage of it to lower overall execution time. Also note that the subproblem solution that an Active Page will make available consists only of the items on two edges of the page.

For this problem we assume the following constants. T_c is the time required by the Active Pages processor to compute the result of a single item of the LCS computation. T_{sa} is the fixed overhead cost associated with an interpage communication. T_{sb} is the cost to transfer items between pages on a per-item basis.

```
partition x and y into k segments
divide the computation into x/q and y/q smaller computations
initialize page (i,j) with the corresponding component i of string x
           and with component j of string y.
let page (i, j) perform the conventional LCS algorithm after subproblems
           (i, j-1), (i-1, j), and (i-1, j-1) have been solved.
page (i,j) dispatches results to neighboring subproblems.
```

FIGURE 35.9 ■ The two-dimensional LCS algorithm.

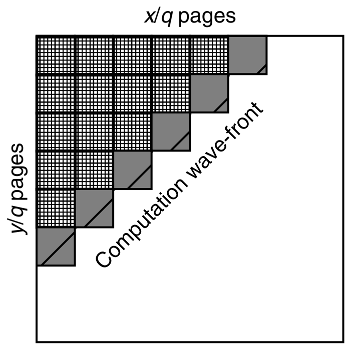

FIGURE 35.10 ■ Parallel execution of two-dimensional LCS on Active Pages.

Further, since the dynamic programming model dictates that the number of items in a page be quadratic in terms of the length of sequence x and the length of sequence y, we define the page size p to be equal to q^2, where q is a variable.

This makes the reasonable analytical assumption that x and y are of similar lengths. We can express application execution time as

$$T < 2 \cdot \sum_{i=1}^{j} \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] + 2 \cdot \sum_{i=j+1}^{n/q} i \cdot \left[3 \cdot T_{sa} + (2 \cdot q + 1) \cdot T_{sb}\right] \tag{35.2}$$

where j represents the particular wave-front in which the overall execution switches from being bounded by computation to being bounded by communication. Focusing on the first half of the computation-bound area, each wave-front has an ever-increasing cost of communication. This is because more Active Pages are involved in each wave-front.

At first, the communication is hidden by computation, but eventually the cost of communicating the required data between wave-fronts exceeds the cost of computation for the wave-front. At this point, the algorithm crosses over from being bounded by computation to being bounded by communication; thus, computation completely overlaps with communication. We denote the wave-front where this occurs as j. This chapter presents an analysis that achieves a better theoretical upper-bound than the conventional sequential solution. Based on particular protein sequence sizes, computer-assisted analysis can reveal the ideal j and q, which minimize the execution time of this algorithm, thus tailoring the behavior of Active Pages in terms of the given problem size. The simulation results show that computer-calculated ideal page sizes entail even a slightly better performance than the theoretical analysis. As will be seen, this is because of a simplification in the analysis.

Suppose we force $j \geq n/q$. This implies that the algorithm will never become bounded by communication resources. We can do this by carefully selecting q and then demonstrating that this q does indeed force $j \geq n/q$. To find a q that satisfies these conditions, we require that the communication always weighs less than computation:

$$\frac{n}{q} \cdot \left[3 \cdot T_{sa} + (2 \cdot q + 1) \cdot T_{sb}\right] \leq \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] \tag{35.3}$$

Then simplify this inequality by:

$$\begin{aligned} \frac{n}{q} \cdot \left[3 \cdot T_{sa} + (2 \cdot q + 1) \cdot T_{sb}\right] &\leq \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] \\ \frac{n}{q} \cdot \left[3 \cdot q \cdot (T_{sa} + T_{sb} + 1)\right] &\leq \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] \\ T_c \cdot q^2 &\leq \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] \end{aligned} \tag{35.4}$$

This simplification will not lead to an absolute lower-bound on execution time, but it does present a tractable alternative that can be used to find an "ideal" q:

$$q \geq \sqrt{n} \cdot \sqrt{\frac{3 \cdot (T_{sa} + T_{sb} + 1)}{T_c}} = \alpha \cdot \sqrt{n} \tag{35.5}$$

Then use this q to drop j from the equation, since the algorithm will never be bound by communication:

$$\begin{aligned} T &< 2 \cdot \sum_{i=1}^{n/q} \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb} \right] = 2 \cdot \frac{n}{q} \cdot \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb} \right] \\ &= 2 \cdot \frac{\sqrt{n}}{\alpha} \cdot \left[T_c \cdot n \cdot \alpha^2 + T_{sa} + \sqrt{n} \cdot T_{sb} + \alpha \right] = O(n\sqrt{n}) \end{aligned} \tag{35.6}$$

While $O(n\sqrt{n})$ is a loose upper-bound, it is faster than the conventional runtime of $O(n^2)$. The simulation results concurred with the findings and suggested a slightly better than $O(n\sqrt{n})$ lower worst-case execution bound.

Figure 35.11 depicts simulated performance of the LCS algorithm; two curves are shown. The first curve depicts the predicted performance of $O(n\sqrt{n})$ (using *asymptotic* parameters from Table 35.3). The second curve predicts a more realistic performance of $O(n^{4/3})$ (using *typical* parameters). The discrepancy is because of communication performance. If communication were more expensive, then the ideal page size would shift away from communication requirements and toward increased computational requirements, amplifying that term in the execution time expression. This in turn would reveal the asymptotic order of the LCS algorithm.

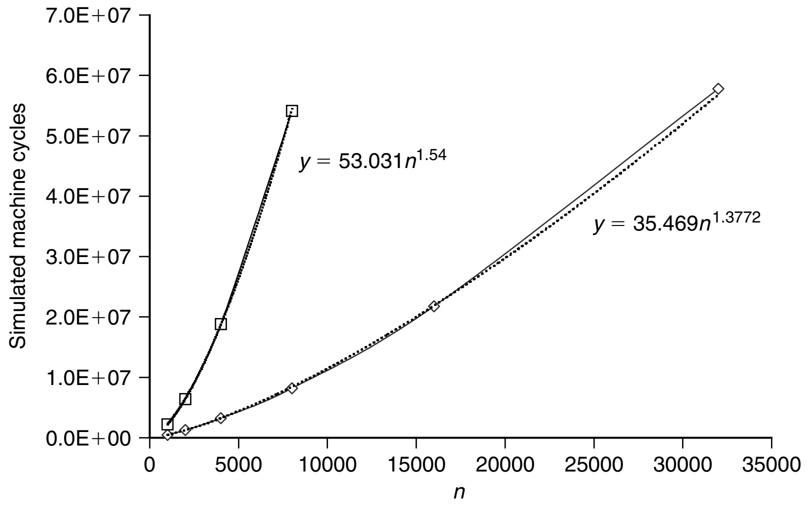

FIGURE 35.11 ■ Simulation results for the two-dimensional LCS.

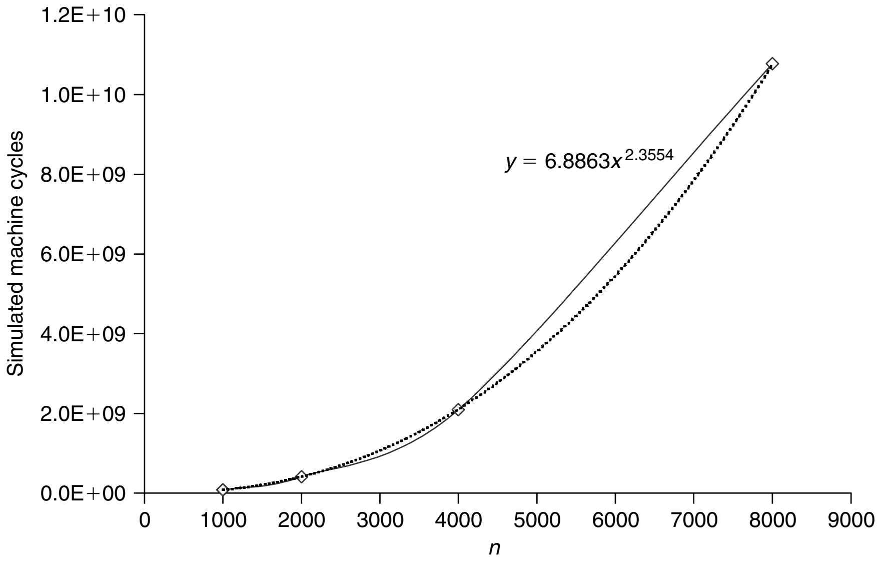

FIGURE 35.12 ■ Simulation results for the three-dimensional LCS.

A more realistic depiction of application performance follows an $O(n^{4/3})$ trend. A similar analysis predicts performance of $O(n^{7/3})$ for three-dimensional LCS. Figure 35.12 shows that the simulated performance for three-dimensional LCS closely matches this prediction.

35.3.4 Summary

We can see that with a memory-centric architecture such as Active Pages, in which the computation scales with the communication, the asymptotic complexity can be reduced. We also see that it is a much more complex equation than one might think. The overhead of the Active Pages, the delay of any communication, and the page size need to be taken into account. Two algorithms, along with validated simulations, have been presented to show their new asymptotic properties. We have found that the inexpensive parallelism provided by page-based intelligent memories can have a significant affect on asymptotic performance. We have also found the optimal page sizes that are required to maximize performance.

35.4 EXPLORING PARALLELISM

In any memory-centric system, we must decide the proper balance between memory resources and computation power. To save money, we could share a single computational element with twice as much memory. Allowing sharing can potentially even out the computational requirements of two processing elements because their needs may not always be identical.

This section looks at virtualizing the computational logic across superpages in the Active Pages chip. Virtualization is accomplished by time-slicing a VLIW processor (see VLIW datapath control subsection of Section 5.2.2) across one to eight Active Pages. We refer to this time-slicing as the *multiplexing* of the computational logic. This study presents an analysis of multiplexing and its effects on performance in a multiprocess environment. In addition, it looks at how varying individual processor widths affects performance. By combining these approaches, we demonstrate that multiplexing is a more effective technique for reducing logic area requirements than reducing individual Page Processor performance.

In this study, we chose to use VLIW computational elements rather than an FPGA so that we could explore the trade-off between instruction-level parallelism and task-level parallelism. The results hold for FPGAs as well. From a high level, it is merely the trade-off between smaller dedicated resources per memory segment and shared resources between memory segments. The study is cleaner when using processor width rather than FPGA area.

35.4.1 Speedup over Conventional

We begin with the raw speedups of a commodity workload that is used for this study. Because the focus is on multi-programmed systems, we are using a slightly different workload than before.

Figure 35.13 depicts application speedup when applications use an Active Pages memory system. Speedup is measured in terms of wall-clock time for the application in a conventional memory system divided by its wall-clock time

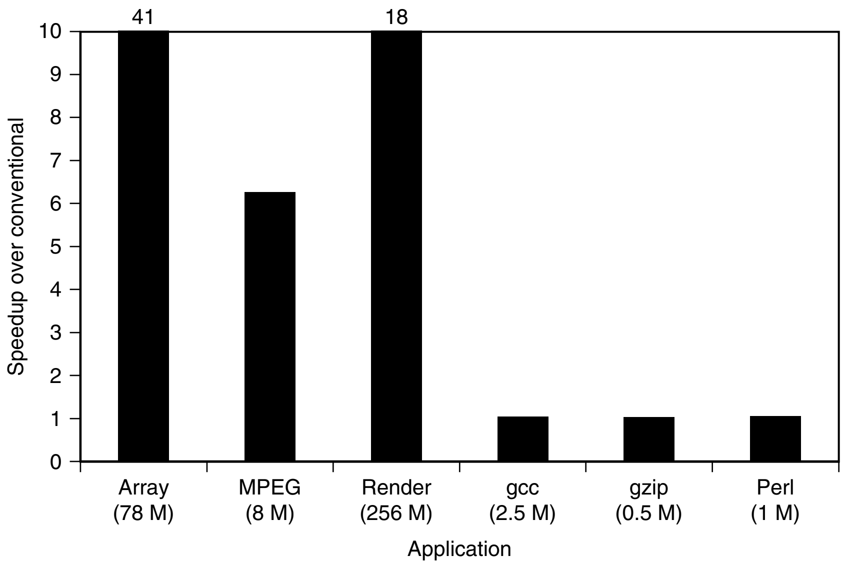

FIGURE 35.13 ■ Speedup over conventional.

using an Active Pages memory system. We observe that Active Pages applications continue to show substantial speedups when executed in a multiprocess environment. That is, even when many independent applications are executed at once, the applications experience speedup.

35.4.2 Multiplexing Performance

We continue by exploring how much performance degradation occurs as resources are shared between Active Pages. Figure 35.14 depicts relative application performance as the degree of multiplexing is increased. We normalize the results to a configuration with no multiplexing, where a one-to-one relationship exists between 4-wide VLIW processors and DRAM subarrays. Multiplexing factors of two, four, and eight make up the remaining data points. Note that hardware multiplexing of eight incurs no more than a 17 percent performance penalty, and a multiplexing factor of four incurs no more than a 6 percent performance penalty for all Active Page applications in the workload.

35.4.3 Processor Width Performance

It is promising that with a 4-wide VLIW, performance does not degrade substantially, as it is shared between Active Pages. Is this because the VLIW processor is not being used efficiently? We now examine the inherent instruction-level parallelism (ILP) in our applications. Figure 35.15 depicts relative application performance as VLIW processor width is varied. Here, processor widths of one, two, four, and eight were evaluated. We observe that half of the applications show a 20 to 80 percent increase in performance from increasing processor width, but the other half do not. It should be noted that MPEG suffers adverse

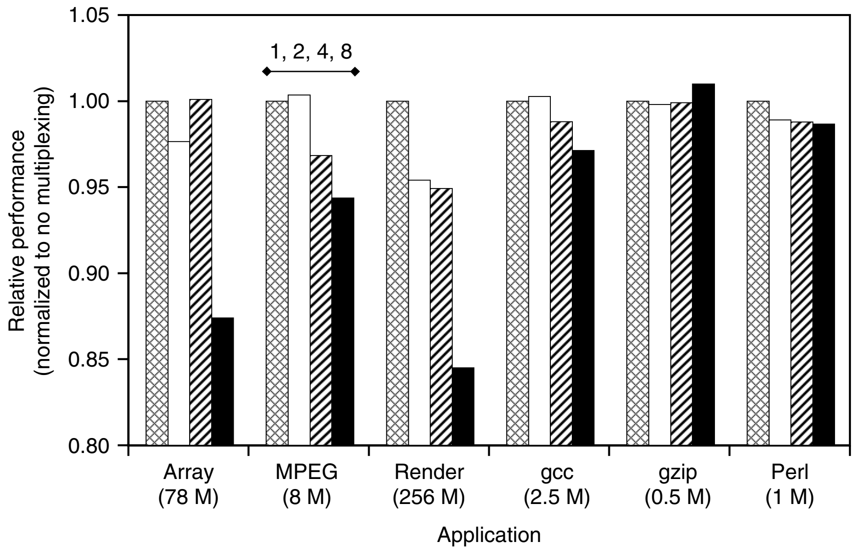

FIGURE 35.14 ■ Performance versus hardware multiplexing.

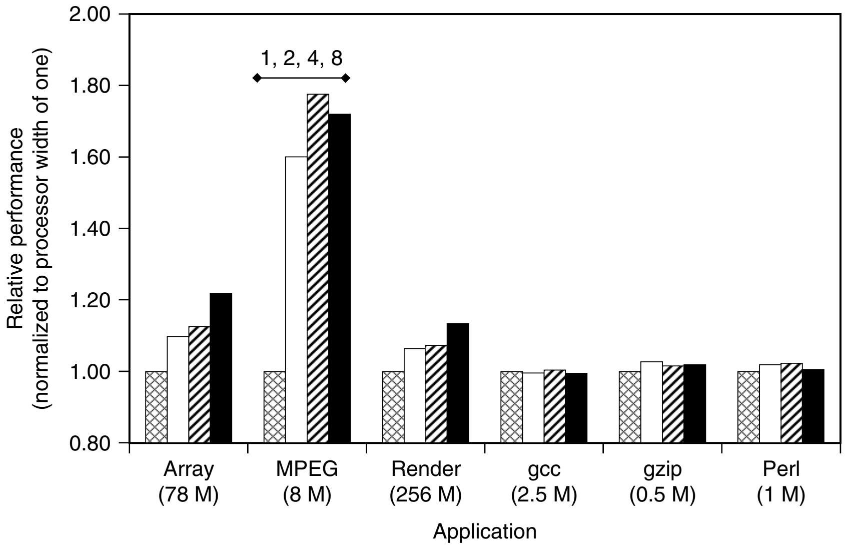

FIGURE 35.15 ■ Performance of multiplexing versus VLIW processor width.

cache effects with a VLIW width of eight, thus lowering performance relative to a 4-wide VLIW. We note that the largest performance gains because of VLIW processor width are achieved with processor widths of two and four, and not with eight.

35.4.4 Processor Width versus Multiplexing

Taking another look at Figure 35.15, we find that the Active Pages applications do not have the static instruction-level parallelism to use much beyond a 4-wide VLIW processor. In addition, Figure 35.14 shows that degradation because of multiplexing is superlinear, suggesting that too much coarse-grained parallelism exists within the application workloads to substantially multiplex processor resources.

An experiment designed to compare these two forms of parallelism is depicted in Figure 35.16. Here we compare an Active Pages device using a single-issue processor with no multiplexing against a device using a 2-wide VLIW with two-way multiplexing, a 4-wide VLIW with four-way multiplexing, and an 8-wide VLIW with eight-way multiplexing.

In the Active Pages applications, a 2-wide VLIW with two-way multiplexing shows a performance gain. This implies that the gain from the increased ILP outweighs the reduced coarse-grained parallelism. Because several conventional applications are active in the workloads, this makes sense because many of the pages do not need the page processors. A 4-wide VLIW with four-way multiplexing is the best configuration studied. Hence, we use this configuration in the remainder of this study.

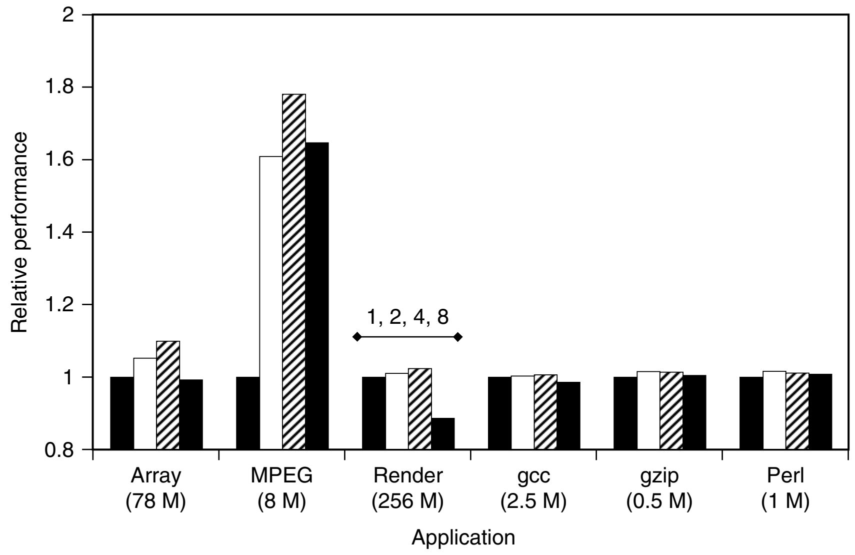

FIGURE 35.16 ■ Performance versus processor width.

To describe why multiplexing performs well in a multiprocess environment, we identify three key factors: nonactive memory, Active Pages processing time, and partitioning.

Nonactive memory
This helps mask the performance degradation because of multiplexing. By definition, all pages of memory in a conventional application require no computation in memory. Some pages in an Active Pages application also require no memory computation.

Active Pages processing time
This is the amount of time spent by the Active Pages computing without main processor intervention. The time varies with Page Processor performance. Simple data manipulations are easily offloaded to the memory system. This leads to longer per-page computation times, most notably MPEG, with Active Pages processing time on the order of seconds.

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

In an environment with Active Pages processing times longer than a Central Processor time slice, such as those observed in MPEG, we would expect multiplexing to degrade performance. Within this study, however, degradation is minimal due to the relatively low memory requirements of MPEG and the effects of conventional memory (without computational capability).

Partitioning

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

We find that multiprocess environments change the position at which an application transitions from scalable to saturated. Multiprocessing time slices the Central Processor, which may be viewed as artificially slowing down the processor from the perspective of a single process. This will shift the scalable-saturated point toward smaller problem sizes. We may use multiplexing to reverse this shift. Essentially, multiplexing slows down the Active Pages computation, shifting the scalable-saturated point back toward larger problem sizes.

Because of the preceding properties of multi-programming environments, we observe that multiplexing is an efficient mechanism for reducing logic area requirements in an Active Pages memory device. A four-way multiplexed 4-wide VLIW Active Pages device is estimated to require 12 percent of the available chip area for computational logic while still providing substantial performance gains. This estimate is based on the reduced logic area coupled with a 20 percent logic area increase because of additional interconnect requirements.

35.4.5 Summary

This study has looked at a promising method for reducing the computational logic area requirements of an Active Pages memory device. Such an approach could be exploited by any memory-centric device. By multiplexing the computational logic among one to four Active Pages, hardware cost can be reduced by four times with little performance impact in a multiprogrammed environment. Further, we find that it is more important to have fewer, faster computational logic elements that are time-shared across pages than more abundant, slower ones available for direct computation at each page. With a 4-wide VLIW processor multiplexed with every four Active Pages, computational logic area can be reduced to 12 percent of total chip area in a gigabit DRAM.

35.5 DEFECT TOLERANCE

The previous section explored the parallelism trade-offs gained by sharing computational units between pages. This section focuses on another major factor in cost: manufacturing defects. DRAM architectures use redundant cells to tolerate defects, dramatically increasing chip yields and reducing cost. Embedded processors, however, do not have an analogous unit of redundancy. While multiplexing several Active Pages with one embedded processor reduces chip area, multiplexing each group of pages with two processors allows each group to tolerate a processor defect. This *associativity* requires some additional interconnect, but tolerance to randomly distributed processor defects increases from 33 percent to more than 50 percent.

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

Instead of four Active Pages sharing one 4-wide VLIW processor, we allow eight pages to share two processors. We study the effect of randomly distributed processor defects on this associative system. If a group suffers two defects, the operating system will only map conventional pages to that group (pages with no computation).

The performance degradation because of randomly distributed processor defects is depicted in Figure 35.17. We note that up to a 50-percent defect rate is tolerated. Increasing the defect rate to 60 percent decreased the number of functional Active Pages below that required by the workload without page swapping. Virtualizing Active Pages to disk was studied by Oskin et al. [11], and a similar mechanism can be used to further increase defect tolerance.

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

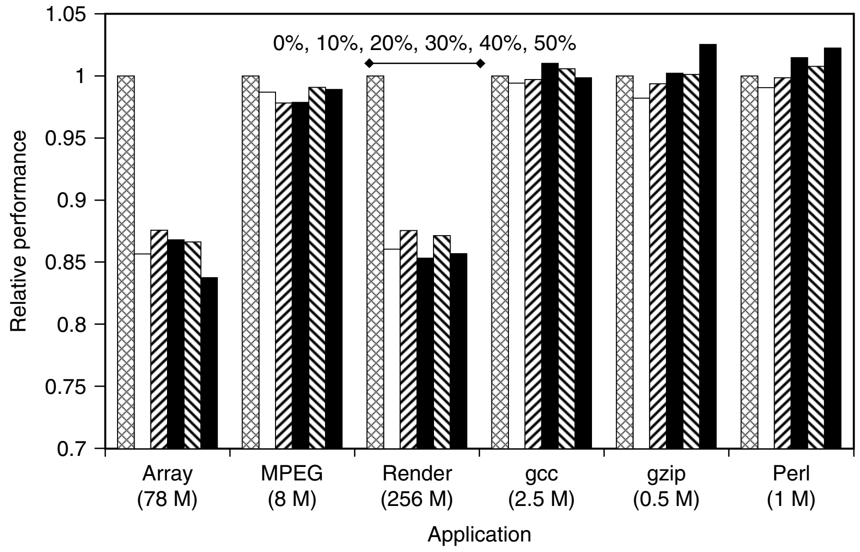

FIGURE 35.17 ■ Performance versus random processor defects.

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

As noted in this section, multiplexing, associativity, and clever OS resource allocation can map around manufacturing defects with only a 20 percent performance penalty with 50 percent random logic defects. An Active Pages–aware OS can be defect tolerant and allow a lower-cost system to be developed by increasing manufacturing chip yield. These incremental costs make Active Pages an attractive memory-based computation model, though the same principles would hold for FPGA-based systems (see Chapter 37).

35.6 RELATED WORK

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

The project that is most similar to Active Pages is FlexRAM [18], which targeted general-purpose computation. The goal was to find computation that could take advantage of the bandwidth provided within a DRAM chip. FlexRAM proposed a hierarchical solution with simple computational elements within each page and a more complex processor for each DRAM. This allowed communication to be handled by an on-chip processor rather than the Central Processor. This had the disadvantage of adding pins to commodity DRAM packaging.

Several other projects explored placing processors in DRAM for more massively parallel computation. IRAM [19] solved this problem by placing a single-vector processor in DRAM. For applications amenable to vectorization, this is an excellent match between a high, bandwidth memory and a processing element. Notre Dame's PIM [20] project uses SIMD functional units to consume the extra bandwidth. DIVA [21] has the most sophisticated design, allowing for a kernel to run on the PIM processors. It also features a dedicated PIM communication network, allowing for communication between PIM processors without host processor intervention. Currently, there is a single computational element in each DRAM.

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

35.7 SUMMARY

This chapter presented the enormous potential for memory-centric computation, along with several issues specific to the Active Pages DRAM environment. The potential for all memory-centric designs is the bandwidth between memory and the nearest computational unit. The challenge, just as in Active Pages, is how to communicate between units. As the ratio of memory to processing units decreases, the total bandwidth increases, but the communication needs increase. This different balance between computation and communication can affect the asymptotic properties of algorithms.

The barriers for intelligent memory, in particular, are the need for explicit parallel programming and the buy-in by manufacturers to put it in commodity production to lower the price. DIVA is working on a migration path for this technology. The advent of multicore commodity processors pushes the field in two directions. First, it provides performance improvements in multi-programmed environments without the need for parallel programming. This hurts the case for intelligent memory. The prevalence of parallel processors on the market, however, increases the utility of parallel programming so that this may not be such a rare skill in the future. If parallel programming becomes commonplace, then intelligent memory will be poised for success in the commodity market.

Acknowledgments Like any large-scale project, Active Pages was the work of several people over several years. Fred Chong and Mark Oskin were the driving force behind the project. Matt Farrens provided valuable advice. Several graduate and undergraduate students contributed to the project, including Justin Hensley, Lucian Vlad-Lita, Tim Sherwood, Ravishankar Rao, Aneet Chopra, Paul Sultana, and Jennifer Hollfelder.

References

[1] K. Itoh et al. Limitations and challenges of multigigabit DRAM chip design. *IEEE Journal of Solid-State Circuits* 32(5), 1997.

[2] M. Oskin, J. Hensley, D. Keen, F. T. Chong, M. K. Farrens, A. Chopra. Exploiting ILP in page-based intelligent memory. *International Symposium on Microarchitecture*, 1999.

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

[4] M. Oskin, F. T. Chong, T. Sherwood. Active pages: A computation model for intelligent memory. *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[5] P. Ashenden. *The Designer's Guide to VHDL*, 2nd ed., Morgan Kaufmann, 2002.

[6] Altera Corporation. *FLEX 10K Embedded Programmable Logic Family*, May 1998.

[7] M. Oskin, L. V. Lita, F. T. Chong, J. Hensley, D. K. Franklin. Algorithmic complexity with page-based intelligent memory. *Parallel Processing Letters* 10(1), 2000.

[8] A. Kautonen, V. Leppnen, M. Penttonen. PRAM model. *http//www.cs.joensuu.fi/pages/penttonen/parallel/pram.pram.html.*

[9] M. Oskin, L.-V. Lita, F. T. Chong, J. Hensley, D. K. Franklin. Algorithmic Complexity with Page-Based Intelligent Memory. Technical Report CS-01-00, Department of Computer Science, University of California, Davis, February 2000.

[10] D. T. Hoang. Searching genetic database on Splash 2. In D. Buell, J. Arnold, W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, 1996.

[11] M. Oskin, F. T. Chong, T. Sherwood. ActiveOS: Virtualizing intelligent memory. *Proceedings of the IEEE International Conference on Computer Design*, 1999.

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

[13] M. Noakes, D. Wallach, W. Dally. The J-Machine multicomputer: An architectural evaluation. *Proceedings of the 20th Annual ACM International Symposium on Computer Architecture*, May 1993.

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

[15] A. A. Chien, R. K. Gupta. MORPH: A system architecture for robust high performance using customization. *Frontiers*, 1996.

[16] C. Ebeling et al. Mapping applications to the RaPiD configurable architecture. *Symposium on FPGAs for Custom Computing Machines*, April 1997.

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

[18] Y. Kang, M. Huang, S. Yoon, Z. Ge, D. K. Franklin, V. Lam, P. Pattnaik, J. Torrellas. FlexRAM: An advanced intelligent memory system. *International Conference on Computer Design*, October 1999.

[19] D. Patterson. Microprocessors in 2020. *Scientific American*, September 1995.

[20] P. M. Kogge, T. Sunaga, E. A. E. Retter. Combined DRAM and logic chip for massively parallel applications. *16th IEEE Conference on Advanced Research in VLSI*, 1995.

[21] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, G. Daglikoca. Architecture: The architecture of the DIVA processing-in-memory chip. *International Conference on Supercomputing*, 2002.

[22] J. Carter, et al. Impulse: Building a smarter memory controller. *Proceedings of the International Symposium on High-Performance Computer Architecture*, January 1999.