Cache Coherence in Intelligent Memory Systems

Diana Keen, Mark Oskin, Justin Hensley, Frederic T. Chong

Department of Computer Science

University of California at Davis

Abstract

The Active Pages model of intelligent memory can speed up data-intensive applications by up to two to three orders of magnitude over conventional systems. A fundamental problem with intelligent memory, however, arises when data cached by the processor is modified by logic in the memory.

We find that the Active Page model inherently limits sharing in a manner that keeps the coherence problem tractable, but exacerbates saturation problems. Two design decisions are fundamental to the performance of coherence protocol. First, the sharing allows for a hybrid snoopy/directory protocol with latencies similar to a snoopy protocol. Second, we introduce transparent remapping mechanisms which reconcile data scattering with efficient caching to maximize the benefit of coherence. We then investigate two optimizations, one-bit directory and asymmetric bus priorities. Finally, we examine SMP multiprocessor systems to mitigate saturation effects. We find that on smaller data sizes, flushing and hardware coherence get similar performance, but hardware coherence has a more attractive programming model and uses less bandwidth. As the number of threads increases, the bandwidth needs increase, making hardware coherence even more attractive as the system scales.

1 Introduction

As microprocessor clock speeds continue to climb, it has become increasingly difficult to keep computations supplied with data from memory. One promising solution is to move some of the

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

computation into memory [P+97] [GHI95] [MSM97]. We focus upon Active Pages, a model of computation that can be implemented in conventional DRAMs without losing pin compatibility [OCS98] [OHK+99]. Active Page memory systems are meant to accelerate computations on commodity workstations and PCs. A fundamental limitation to any intelligent memory system, however, is the primitive hardware mechanisms to keep data coherent between processor caches and intelligent memory.

Active Pages is 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 primitives such as *insert*, *delete* and *find*. By partitioning computations between a processor and Active Page memory, data intensive applications can execute up to 1000 times faster than on the same system using conventional memory. We implement Active Page systems by embedding small processors, *Page Processors*, with each page of memory in DRAM. We refer to the conventional processor as the *Central Processor*. Each Page Processor may only access the memory associated with it, whereas the Central Processor may access the complete address space.

The limited sharing present in this programming model is shaped by a desire to keep costs low by keeping hardware and operating system support as simple as possible. By disallowing sharing between Page Processors, we eliminate the need for virtual addressing (and its associated hardware and operating system support) in the Page Processors. Page Processors operate directly on the virtual addresses in the program, ignoring the upper bits. In addition, this limited sharing allows us to design a hybrid coherence protocol with the bandwidth of a directory protocol and latencies similar to a snoopy protocol.

With this simplicity comes a price. The current Active Pages model has two fundamental problems. Active Pages can face a fundamental scaling limitation as problems become large – the Central Processor becomes a serial bottleneck. This serial work can be broken into three categories - serial work that is independent of the Page Processors, the time spent communicating with the Page Processors, and the time spent accumulating results received from the Page Processors into a larger answer. We address the latter two limitations. First, we optimize the communication between the Central Processor and Page Processors. Then, we parallelize the accumulation of results by using multiple Central Processors.

Viewed as a shared-memory system, communication between the Central Processor and Page Processors is a cache consistency problem. Previous implementations used *explicit* software flushes to prevent cached data from becoming inconsistent with data modified by Active Pages in memory. At small problem sizes, this is a programmability issue - non-optimally placed flushes can degrade performance or net incorrect results. At large problem sizes, the bandwidth consumed by flushing may become prohibitive. Because they are designed to augment commodity systems and have potentially hundreds of Page Processors sharing a single memory bus, Active Page systems present several challenges in designing a hardware cache-coherence protocol. First, our goal is to integrate Active Pages with standard systems, so we can not change the protocol existing on the system. For this study, we assume the Pentium's MESI snoopy protocol running between the processors on the bus. Second, our system has potentially hundreds of processors running in parallel, and bus-based protocols have well-known performance drawbacks in such an environment. Finally, DRAMs are passive devices on systems, so they may not initiate coherence transactions. Changing the bus protocol is an option, but it would require extra development, standardization, and debugging expense.

Fortunately, the limited sharing that was created by a desire for simple hardware will allow for a simple, high-performance hardware coherence protocol. We can take advantage of this limited sharing to design a protocol that uses the bandwidth of a directory protocol, yet allows the Central Processor to maintain latencies closer to its original bus-based protocol.

Optimized communication, however, is only part of the solution. The Central Processor must perform work to accumulate each Active Page result into a larger result. When this limits the scalability of the application, we refer to the application as *processor-saturated*. We show that adding Central Processors (as in an SMP) alleviates this problem, allowing the application to scale again.

To summarize, we make four primary contributions. First, we present a protocol and system design which efficiently supports coherence with minimal modification to commodity systems. The coherence protocol provides up to a three times speedup over existing hardware-only mechanisms. Second, we present performance simulations of applications enabled by coherence, including: Java garbage collection, decision support, game search, and online game kernel. We also introduce a transparent, cache-conscious remapping mechanism which allows data to be easily scattered to multiple Active Pages. Finally, we show that applications which are *Processor-saturated* benefit from additional Central Processors.

The remainder of this paper is organized as follows: Section 2 gives some background on Active Pages, Section 5 introduces the applications studied, and Section 3 discusses the effect the Active Page architecture has on the implementation of cache coherence protocols. Section 4 presents the coherence protocols we studied. Section 6 describes the experimental methodology

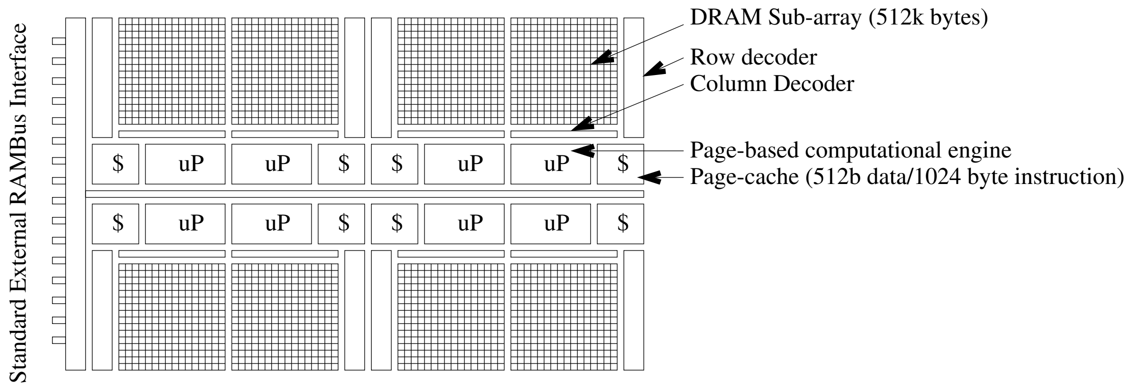

Figure 1: Active Page architecture (8 pages)

we used to evaluate those implementations. Section 7 discusses our results. Section 8 describes related and future work. Finally, Section 9 presents our conclusions.

2 Active Pages

2.1 DRAM Hardware Design

High-density DRAMs are divided into sub-arrays, complete with row and column decoders, to minimize column capacitance and decrease power consumption [I+97]. Our proposed Active Page implementation exploits this natural structure, treating each sub-array as an Active Page. As shown in Figure 1, a small VLIW processor and cache, a *Page Processor* and *Page Cache*, are embedded next to each sub-array to implement Active Page functions [OHK+99]. Using commodity 1G-bit DRAM technology as a target [Sem97], we expect sub-array size to be 512K-bytes and the embedded processors to consume less than 31% of chip area.

Active Page memories are pin-compatible with conventional DRAM and can satisfy memory accesses with negligible added delay. A conventional DRAM access occurs as follows: When the main processor, or *Central Processor*, makes a conventional memory access, the DRAM must arbitrate internally between requests from the Central Processor and the local Page Processor. Two gate delays are incurred for this delay. If no local access is in progress, the access is issued in parallel to both the DRAM sub-array and the local Page Cache. An additional demultiplexor delay is incurred leaving the array in order to select between the DRAM data or Page Cache data. In addition, a larger delay occurs if the Page Processor is in the middle of a DRAM access. This delay, however, only occurs when an application is using Active Pages and is offset by the overall performance benefits gained.

To minimize DRAM modification and reduce hardware overhead, our Active Page implementations do not currently 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. This *processor-mediated* approach to inter-page communication has one disadvantage and three advantages. The disadvantage is that inter-page communication must be infrequent to maintain performance with a single processor. Fortunately, past and present work demonstrate that a broad range of applications can perform extremely well under this constraint.

The advantages of the processor-mediated approach are significant. First, Active Page DRAMs avoid extensive modification to support inter-page communication. To support inter-page communication, we must provide the communication network and arbiters to transfer the messages. In addition, because Page Processors have a limited amount of memory, we must provide handlers that place overflow into system or the process' memory. Also, because Active Page systems use virtual memory, we must provide translation tables so that the destination Page Processor can be determined. Lastly, we must provide support in the memory controller to facilitate inter-page communication between Page Processor that are in separate DRAMs. Second, the operating system maintains control of virtual memory because all accesses and translations go through the Central Processor. Additional synchronization is required if the Page Processors are executing at the same time that a super-page is being placed in a different physical memory location (or swapped out to disk). Finally, sharing is only between the Central Processor and the Active Pages, never directly between Active Pages. If Active Pages could address each other's memory, then one Active Page could hold a line that belongs in a different Active Page DRAM. We shall see that this last advantage is critical to our inexpensive designs for maintaining coherence.

This proposed design for Active Pages leads to three sources of sharing between the Central Processor and the Active Pages. First, there is the data that both the Central Processor and the Page Processors access in their respective computations. Second, there are the software-defined locations used for coordination between the Central Processor and the Page Processors. Finally, there are the locations that the Central Processor must read and write to facilitate communication between Active Pages.

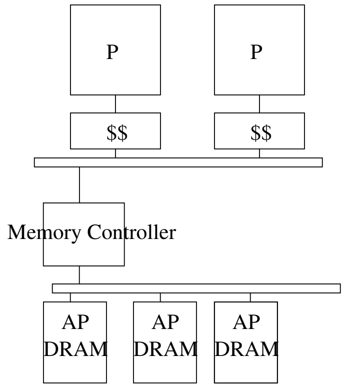

Figure 2: System view with modified Memory Controller

2.2 Active Pages Software System

Each Active Page contains a code space and data space, limited in size to the physical DRAM subarray associated with the Active Page Processor. The block of pages associated with each Page Processor is referred to as a super-page. To program an Active Page Processor, the programmer creates an object in C++. The 512k allocated to each Active Page Processor is divided between code, stack, and data. The code must fit within the code segment, and the data size of the object is padded appropriately.

When an Active Page object is allocated, the operating system gets involved. The operating system allocates the memory and loads the code into the correct region. The Active Page Processor begins upon activation, first performing any initialization (constructor) and then polling a variable waiting for an invocation of a method. To maintain pin compatibility, all Active Page functions are implemented using conventional reads and writes. The Central Processor invokes Active Page functions by writing the parameters into appropriate places in the Active Page memory. The Central Processor then changes the variable on which the Page Processor is polling, indicating what function to execute next. When the Page Processor has completed the method, it resets the looping variable and waits for the next invocation. The Central Processor can poll this variable to determine whether an Active Page has completed a particular function. An example of communication between the Central Processor and Page Processor during execution is given in Section 4.

3 Hardware Coherence Design Issues

The hardware coherence protocol design is influenced by two main factors. First, we want to change as few components as possible. There are several components in this system: DRAM, Memory Bus, Memory Controller, System Bus, and Central Processor. Each element we modify is extra cost in terms of design, verification, and loss of commodity. Because the DRAM will be redesigned to accommodate Active Pages, the extra cost of coherence support can be substantially amortized. Second, our programming model is an asymmetric model in which there is one Central Processor and many Page Processors.

We first present our basic hybrid protocol. We then present two optimizations. The asymmetric programming model creates access patterns which produce poor cache performance with our basic protocol, so we add support to the memory controller to address this problem. We then look at how to place coherence requests from the passive DRAM through the memory controller and onto the system bus.

3.1 Hybrid Coherence Protocol

Active Page systems are asymmetric in their sharing patterns. All sharing is between the Central Processor and individual Active Pages. Active Pages can only cache data local to their own page, which prevents sharing between Active Pages. In fact, sharing can only occur with data contained in the L2 cache.

We design our protocol knowing that it is expensive to change the protocol run on the Central Processor, and our protocol must sustain hundreds of Page Processors. From the Central Processor's point of view, then, we must use a bus-based snoopy protocol. From the Page Processors' points of view, we should use a directory protocol to reduce unnecessary bus traffic. In light of those constraints, we must look at two fundamental design questions. The first is how to limit coherence messages to sharers of a relevant line, and the second is how to serialize the operations. The result is a hybrid protocol that exploits both the asymmetry of our system and the limits on shared data.

First, we limit coherence messages to the appropriate sharers. For Central Processor requests, the possible sharers are other Central Processors and the single Page Processor associated with the request's physical address. The other Central Processor runs its MESI protocol. The Memory Controller acts as an agent for the Page Processor. The Memory Controller sets the signal on the coherence line to indicate it is not ready to respond. It directs the request to the appropriate

Page Processor (determined by the physical address), and begins a read operation in parallel (if the coherence operation included a miss). The Page Processor checks to see if that cache line is in its cache. If the system allowed sharing between Page Processors, then we would need to either broadcast every request or keep a large directory. Because of the limited sharing, we do neither. The Memory Controller responds when it receives the sharing information from the DRAM. Because this is done in parallel with the access, it does not slow down the total latency of miss operations. For invalidate operations, there is no response expected from the Memory Controller, so there also is no slowdown. If another Central Processor can satisfy the request, we may see a slowdown because the Memory Controller indicated it was not ready to respond. We can sense this and have the Memory Controller respond the next cycle that it does not have the line, and it can delay or cancel its request to the DRAM.

On the DRAM side, we wish to limit the number of Page Processor requests that reach the bus because there are potentially hundreds of Page Processors running at once. For Page Processor coherence operations, we keep track of whether the data may be in the Central Processors' caches. This implies a directory bit per line in memory (128 bytes) to keep track of whether data has been read by a Central Processor. A second bit may be used to determine whether the line is shared or exclusive in a Central Processor cache. With only one bit per line, every read request by a Page Processor to a cache line that is in a Central Processor cache must go on the bus even if the line is in the shared state in the Central Processor (implying that a consistent line is now in the DRAM). With hundreds of Active Pages in a system, this could quickly saturate the bus. We show in Section 7.4 that even though the worst-case scenario is much worse with a single bit than two bits, our applications do not benefit much from the second bit.

Finally, we must serialize operations. If a protocol has no serialization point, then some writes can be lost in the system. In a traditional bus-based multi-processor system, the serialization point is the bus, and each processor has the necessary hardware to detect when there is an outstanding request for the same line as a new request. In a traditional directory protocol, the serialization point is the directory. The serialization point is not as obvious because the Page Processors are not connected directly to the system bus. Instead, coherence requests are placed in a hardware queue for the memory controller to place on the bus. This can be a problem. For example, if the Central Processor places an invalidate on the bus at the same time the Page Processor places an invalidate into the internal DRAM queue for the memory controller to pick up (and place on the bus), they are perceived in different orders, and both processors can lose track of the line (and their changes). The first must be an invalidate, and the Page Processor

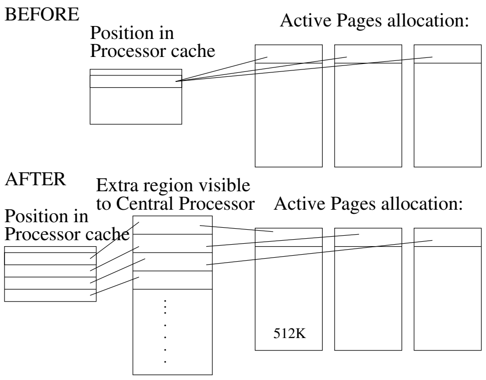

Figure 3: Mapping of variables into cache before and after remapping.

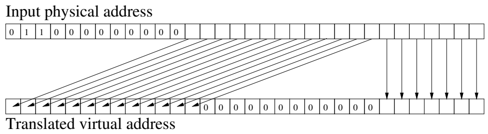

Figure 4: Remapping operation

request must change to a read-invalidate so that the Page Processor can receive the updated line from the Central Processor. The bus must still be the serialization point, since that is what the Central Processor expects. Page Processor operations can proceed if they do not require a bus transaction. For operations that require a bus transaction, the Page Processor stalls until its request reaches the bus (and for some it must wait for completion). This is why the question of priority on the bus arises.

The Central Processor already has the hardware necessary to reconcile requests for the same address by bus snooping. For the Page Processors, we will perform our serialization in the hardware coherence request queue. As Central Processor requests enter the DRAM, they are checked against the requests currently in the queue, and any changes to the type of request are made. From the previous example, when the DRAM receives the invalidate, it will find the invalidate to the matching address in its queue, and change the waiting Page Processor invalidate into a read-invalidate. Thus, even though the Page Processor may have been initiated first, it becomes the "later" invalidate because it reaches the bus after the Central Processor request. Any request still in the DRAM hardware coherence request queue when a Central Processor request arrives is a later request.

3.2 Data Placement

Cache coherence introduces a unique data placement problem that was not previously an issue in Active Page systems. The asymmetric nature of the computation, which has a single Central Processor coordinating hundreds of Page Processors, causes poor cache performance. The problem occurs when the Central Processor must coordinate with all Active Pages through some local synchronization variable which is present on each page. For example, if the Central Processor is trying to determine which Active Pages have completed a particular function, it may poll a "done" variable on every page. This variable would typically be stored at a constant offset in each page. Without coherence, this is not a performance problem since the processor may not cache variables that might have been changed in the Active Pages. With coherence, we would expect the Central Processor to poll on cached copies and only fetch new copies of variables that have changed.

Unfortunately, with power-of-two page sizes and local variables at the same offset in every page, many conflict misses occur. All of these variables map to the same location in the cache, as seen in Figure 3. Varying the offset of the local variables is problematic and would involve extra indirections to calculate offsets as a function of Active Page number.

A crucial observation is that the Active Pages themselves are not adversely affected by having their data at a constant offset. Only the Central Processor needs to be able to avoid constant offsets for these local variables.

Page coloring is a typical mechanism to alleviate conflict misses due to data placement. The block of pages associated with each Page Processor is a super-page. Changing the placement of pages within super-pages (within an Page Processor's sub-array) would require indirection in the Page Processors. In addition, we found that remapping at a page granularity did not scale, either. Once the number of Page Processors is greater than the capacity of the cache divided by the page size, we encounter the same problem. We present a scheme which allows interleaving at any power-of-two cache line granularity.

We introduce *remapping*, an addressing transformation implemented in the memory controller which takes these awkwardly placed synchronization variables and makes them appear in more cacheable locations for the Central Processor. Figure 3 illustrates the translation from the virtual address the Central Processor uses to the physical address in DRAM. Specifically, user-defined locations on each Active Page are remapped such that they are cached in contiguous, rather than conflicting, cache lines.

Conceptually, the memory controller must maintain a reverse TLB to take a physical address from the Central Processor, translate it back to a virtual address, do a bit shift as shown in Figure 3, and then to a remapped physical address in the Active Page DRAMs. In reality, we can skip the step from physical to virtual, removing the need for a reverse TLB. We perform the bit manipulation and then translation (physical-to-physical), but the virtual address information must be kept in order to keep the translation consistent with page table changes. For hardware coherence messages coming back from the Active Page memory, the reverse operation is applied and the coherence message sent up-stream to the processor cache.

To use this feature, we introduce a keyword “cached” into the compiler for our system. The programmer labels certain variables as ”cached” similarly to how the keyword ”register” is currently used. Variables with this keyword are mapped by the compiler to the beginning of each page. In the Central Processor’s code, the compiler addresses the variables in this new virtual space. The original address will no longer have coherence since the memory controller will remap Active Page coherence traffic into the new address space. Like the “register” keyword, “cached” is a compiler hint. If there are too many variables for the region, the compiler places some variables into the non-remapped memory space. The compiler inserts instructions at the beginning of the program to notify the memory controller as to how many cache lines should be remapped. This allows a gentle degradation in performance because the more cache lines that are remapped, the greater the potential for conflict misses when accessing variables that are in the same location in each super-page.

3.3 Passive Memory Chips

A potential difficulty arises from the DRAM pin-compatibility goal of previous Active Page implementations. DRAMs are passive devices and their interface does not allow them to initiate coherence transactions. To facilitate coherence, we must either change the interface or frequently poll every Active Page DRAM.

Polling via the Central Processor is clearly impractical from the standpoint of scalability and performance. Instead, we propose that the memory controller be modified to poll each DRAM and relay coherence actions on the bus. This polling organization, shown in Figure 2, allows the DRAM interface to remain unchanged. Allowing the interface to remain unchanged reduces the design time for the system, not saves hardware cost, since we are already modifying both the memory controller and the DRAM chips. The modified memory controller has a connection to the bus, allowing it to relay coherence operations from Active Pages to the Central Processor’s cache. The memory controller will be acting as another processor connected to the coherence lines. These modifications to the memory controller are less intrusive than processor changes. Comparable modifications have been proposed by other research projects [C$^+$99].

Our polling proposal, however, can introduce significant delays in Active Page operation. Active Page coherence actions are placed on a memory-mapped queue on each DRAM chip. A Page Processor stalls while a relevant coherence request waits for the memory controller to pull the request from its queue and satisfy it via the bus. Additionally, a single memory controller must satisfy coherence requests from several Active Page DRAMs, which translates to hundreds of Active Pages. This bottleneck may further degrade Active Page performance and implies that the Central Processor can have significantly more priority on the bus than any given Active Page.

The obvious alternative is to modify the DRAM interface so that coherence actions may be initiated. Surprisingly, we shall see that the asymmetry of Active Page systems favors the simpler polling design. This is because the only time the bus prioritization matters is when there is high contention. When there is high contention, the application is processor-saturated. Previous studies [OKH+00] have shown that when the system is saturated, the performance of the Page Processors is insignificant compared to the Central Processor. So this simplistic design is expected to cause little degradation when the load is light, and no degradation when load is heavy.

4 Coherence Protocols

All systems simulated utilize a write-through L1 cache and write-back L2 cache. With this in mind, we explore the following three coherence options:

1. Uncacheable Regions

   Uncacheable regions requires no change to the Central Processor. just the compiler. the programmer designates certain variables "uncached", just as programmers often use "volatile." In addition, the compiler must map these variables to uncacheable virtual pages (The Pentium allows each virtual page to be marked uncacheable). To give the best-case performance for the uncached regions, when an uncached variable is accessed, the single word is requested from memory rather than the whole line.

   Uncacheable Regions are desirable because the volatile keyword is already supported by compilers, is intuitive for the programmer, and requires no changes to the hardware.

   The disadvantages are correctness and performance. Although the volatile keyword is intuitive for the programmer, it is the source of many programming errors. If the programmer forgets to make a shared variable volatile, then the program will not be correct. Uncached regions also have a performance disadvantage. Variables that are not cached require a DRAM access for every read or write. Writes become write-throughs because they can not be cached. If the variables are never read, this can be an advantage as will be shown later. However, if the program reads several words of the same cache line in succession, the performance degrades considerably compared to a cached access. All of the words must be read in separately from the DRAM rather than one DRAM access and several cache accesses. Because the processor reads and writes at a word granularity, the processor does not gain the advantage of reading in the whole line. Uncached variables require a separate DRAM access regardless of their location.

2. Explicit Flushes

   Explicit flushing requires the programmer to decide when to synchronize the data. An explicit flush writes all dirty data back to the DRAM and invalidates that page from the Central Processor cache. Previous publications involving Active Pages used explicit flushes for coherence. Each time an Active Page function was called, the Central Processor cache was flushed. If the Active Page remains active while the Central Processor changes data, and the Central Processor does not use explicit flushes, the Active Page can receive data out of order, resulting in incorrect execution. The order of write-back to the DRAM is determined on the pattern of future accesses, not the order in which writes are performed. The flush is a software command with an address as the argument. The cache expels the appropriate address from the cache. If the line is dirty, the line is written to the DRAM.

   In theory, explicit flushing allows the data to reside in the cache for limited amounts of time. If the processor needs the data several times in a row, it incurs one DRAM access per line, not per access. In addition, if there are sequential accesses, it can now take advantage of reading in the whole line. A problem arises between the time the values are written and when the flush occurs. A flush guarantees the last possible time a variables is written, but if a line gets evicted before the explicit flush, the Active Page will receive the new value before the flush. The time at which each cache line gets written back to the DRAM is nondeterministic, so the Active Pages may see writes in a different order than they occurred. This makes the placement of the flushes crucial to program correctness. Program correctness is, however, always possible. If every read or write to Active Page memory is preceded and followed by explicit flushes, then we maintain correctness (assuming that the programmer has properly maintained mutual exclusion for shared variables). The performance, however, degrades to be equal or worse than uncached regions.

   To illustrate the difficulty in programming an efficient program with explicit flushes, we give a code example in Figure 5. We have a software queue, p-¿IQ, that is shared between

```
**********  Original code ************
void FillQueue(Page *p, int n, int *v)
{
  int i = 0;
  // while there is room in the circular
  // buffer AND there are more elements
  // to add, insert an element

  while((i < n) &&
    (((p->IQTail+1) & QMASK) != p->IQHead))
  {
    // place an element in the buffer
    p->IQ[p->IQTail] =  v[i++];
    // increment the tail value
    p->IQTail = (p->IQTail + 1) & QMASK;




  }
}
```

```
*********  Inefficient solution  ********
void FillQueue(Page *p, int n, int *v)
{
  int i = 0;
  // flush to read the most
  // recent values of head and tail
  flush(&(p->IQHead));
  flush(&(p->IQTail));
  while((i < n) &&
   (((p->IQTail+1) & QMASK) != p->IQHead))
  {
    p->IQ[p->IQTail] =  v[i++];
    // send that value to memory
    Flush(p->IQ);
    p->IQTail = (p->IQTail + 1) & QMASK;
    // send the tail value to memory
    flush(&(p->IQTail));
    // read in the newest head value
    flush(&(p->IQHead));
  }
}
```

```
**********  Incorrect solution ***********
void FillQueue(Page *p, int n, int *v)
{
  int i = 0;
  flush(&(p->IQHead));
  flush(&(p->IQTail));



  while ((i < n) &&
    (((p->IQTail+1) & QMASK) != p->IQHead))
  {
    p->IQ[p->IQTail] =  v[i++];
    p->IQTail = (p->IQTail + 1) & QMASK;
  }
  // send queue values to memory
  Flush(p->IQ);
  // send tail to memory
  Flush(&(p->IQTail));
}
```

```
**********  Optimal solution **********
void FillQueue(Page *p, int num, int *v)
{
  int i = 0;
  flush(&(p->IQHead));
  flush(&(p->IQTail));
  // make a local copy of head & tail
  int head = p->IQHead;
  int tail = p->IQTail;
  while((i < n) &&
    ((tail+1) & QMASK) != head))
  {
    p->IQ[tail] =  v[i++];
    tail = (tail + 1) & QMASK;
  }
  Flush(p->IQ);
  // copy local tail to page IQTail
  p->IQTail = tail;
  Flush(&(p->IQTail));
}
```

Figure 5: Four examples using explicit flushes. The first shows the original code without explicit flushes. The second is the simple approach, placing a flush before each read and a flush after each write. This nets correct execution but has the performance of uncached regions. The third example can result in incorrect execution. The fourth is correct and high-performance. Producing code that is both correct and efficient is not straightforward.

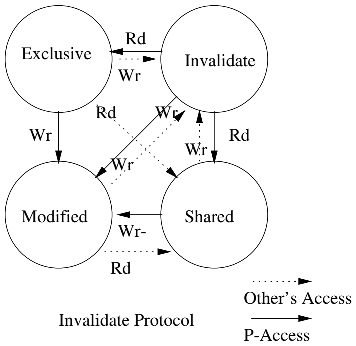

Figure 6: Invalidate Protocol in Central Processor cache

the Central Processor and the Page Processor. The Central Processor places a number of elements into the queue, and the Page Processor reads them out of the queue. We use a circular buffer. The first example shows the simple approach to adding flushes - add a flush before every load and after every store. This reduces the performance to uncached regions, and the results will show that uncached regions performs worse than optimized flushing and coherence. The second example shows a more optimized flush. We can remove the flushes of the head because if we get a slightly old value for head, then we will just exit the loop earlier and insert more items at a later call. We only flush values to memory at the end of the loop. Unfortunately, this code can result in incorrect execution. If an interrupt occurs that causes some of the cache to be written back to memory but not all of it, p-¿IQTail could be written back without the accompanying values in p-¿IQ. To solve this problem, we arrive at the third, correct solution. We make a local copy of the tail variable and flush p-¿IQ before we change p-¿IQTail. To successfully use flushing, the programmer must be able to optimize the flushes while still guaranteeing correctness. As illustrated, this takes a good understanding of what memory orderings will result in incorrect execution in order to be able to remove only the unnecessary flushes.

3. Hardware Coherence

Our Central Processor uses the same cache coherence protocol as the Pentium II [Int97]. Figure 6 shows the four-state MESI protocol - Modified, Exclusive, Shared, and Invalid. If a line is in the Exclusive state, then it moves to the Modified state if the processor writes to it. This transition does not require a coherence request. If another processor reads in the line, the line changes to the Shared state. In an invalidate protocol, the first time a processor writes to a shared line, it invalidates all other copies of a line. Then it can write several times to the line in the cache rather than on the bus. It continues to do so until another processor reads the line. If a write appears on the bus, then a processor invalidates its copy of the line. This way, if another processor no longer has the line in its cache, only the invalidate is a waste of bus bandwidth, not multiple writes. Details of the implementation were given in Section 3.1.

We can exploit on-chip DRAM to give Active Pages a one-bit sharing directory to help optimize coherence actions. We use this bit to determine whether the Central Processor has read the line since the last time it was sent an invalidation. Coherence traffic between an Active Page and the Central Processor only occurs if this shared bit is set.

5 Applications

To illustrate the differences between coherence protocols on our system, we chose applications from desktop and business domains. These applications include: Java garbage collection (JavaGC), a Bridge-playing program (Bridge), a decision-support code (C4.5), and a kernel for an online gaming application. This set of applications requires substantially more sharing between the Central Processor and Active Pages than previous benchmark suites. Table 1 summarizes the attributes of these applications. We describe the algorithms and sharing characteristics in this section.

5.1 Java Garbage Collection

Garbage collection in Java 1.1 is a mark-and-sweep algorithm. In a mark-and-sweep algorithm, the garbage collector begins with a list of root addresses (global and stack data). Each object is marked, and all off the objects to which the marked objects are marked, and so on, until all reachable objects are marked. All objects that are not marked are unreachable, so they may be deleted (or swept). Java uses handles rather than direct pointers. Each handle determines object type, allowing the collector to find all pointers within the object. The collector marks each visited object as seen and continues until all reachable objects have been marked.

The Active Page version accelerates pointer chasing and marking by using Page Processors to follow pointer chains local to their pages. The heap is distributed between pages. The first part of each page contains handles, and the rest the objects. Each page also contains an “in” queue of pointers to be chased and an “out” queue of pointers from chains that leave the page. The

Applications			
Name	Application	Central Processor Computation	Active Page Computation
JavaGC	Java Garbage Collection	Feed pages pointers and collect off-page pointers	Mark reached objects and chase pointers
Bridge	Min-Max search Branch and Bound	Combine results, distribute tasks	Perform search
C4.5	Decision tree generation	Combine Active Page values, perform multiplies and make decisions	Count active records, give multiplies to proc, activate and deactivate records
Minima	Online Game	Migrate objects across pages	evolve its portion of virtual world

Table 1: Summary of partitioning of applications between Central Processor and Active Pages

Central Processor starts the mark-and-sweep algorithm by filling the in-queues of appropriate pages with pointers from the root set. Each Page Processor marks all objects reachable from pointers in its in-queue, placing pointers in its out-queue whenever they refer to objects not in the local page. The Central Processor iteratively copies pointers from out-queues to appropriate in-queues until all queues are empty. For this application, all sharing is through the queues.

5.2 Bridge

Bridge is an example of a classical min-max branch-and-bound game search. The program plays bridge with full knowledge of everyone's cards. It simulates the next several steps of the game with each possibility that may produce a better result than one already found. Each potential hand has a value. We search to a certain depth in the decision tree. For the first card, we want to maximize its worth, because it is the card we will play. For the second card, we want to minimize its worth, since this is what the opponent will play (because it will try to maximize its worth, which is the same as minimizing ours). Hence the min-max search. Each internal node of the tree receives either the maximum or minimum values of its children. The leaves receive the worth of the hand at that time. Branch-and-bound refers to a pruning mechanism to allow early exit of branches in the search tree that will not affect the parent node's value.

The Active Page version has the Central Processor generate all the possible first and second moves and distributes them to the Active Pages. Each Active Page performs the algorithm on a different choice of first and second move. The Active Page returns the value of that move and receives another hand to simulate.

Each time the Active Page receives a new job, it receives the state of the cards with the first two cards played. This means that the Central Processor writes a large block of data to the

Records

Hair Color	Age	Eye Color	Outcome
Blonde	10	Blue	Pizza
Blonde	50	Green	Filet Mignon
Brown	10	Green	Spaghetti
Black	50	Brown	Filet Mignon

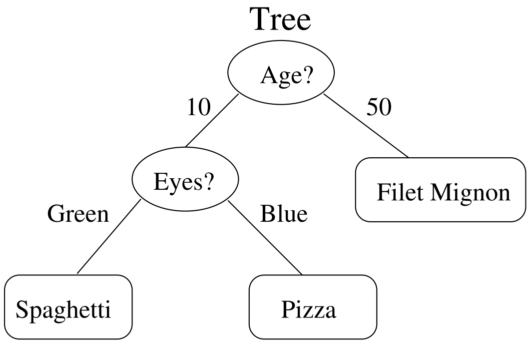

Figure 7: C4.5 Example - Records with completed decision tree

Active Page, but it reads in a single result. This will become important in our discussion later. In addition, the access pattern for polling is the same as for JavaGC. The Central Processor polls all the pages, waiting for a finished page to which it distributes another hand.

5.3 C4.5 - Decision Tree Generation

C4.5 [Qui93] constructs a decision tree from a database of records. Figure 7 shows a trivial example with four records. For each person, three are attributes: hair color, age, and eye color. The outcome is the person's favorite food. The purpose of a decision tree generator is to decide, given a record, what the most likely outcome is. So, given the hair color, age, and eye color of a person, how do we decide what his or her favorite food probably is? Given the above four records, the attribute which is correlated highest to the outcome is the age. The program would make a decision node with the attribute "age". Each possible value of the attribute is an edge branching to another node in the decision tree. The program then looks at the remaining attributes and attempts to find one that would improve the correlation when combined with the age attribute. If no improvement is found, no new nodes are added, and the last node is a leaf.

The correlation for every potential new node is calculated by first forming a count table. A remaining attribute is chosen (eg. hair color) and every possible value forms a row in the count table (eg. blonde, brown, and black). The possible outcomes (eg. filet mignon, pizza, and spaghetti) form the columns. The entries are calculated by counting the number of records that have the attribute value of the row, the attribute values determined thus far (eg. age of 10), and the outcome of the column. We can then assign outcomes to attribute values using the largest values in each row. Given this assignment, a correlation can be calculated. As an added wrinkle, records may have unknown values for some attributes. A record with all known values always has a weight of one since it is always counted as a full record. Records with unknown values, however, must spread their weight over multiple attribute values for a given tree. If a particular attribute is unknown, then it is not used to decide the best attribute for that node. However, after this node is formed and the program must determine its subtrees, the records with the unknown attribute value are given a fractional weight to be used in forming the count tables for the subtrees. The same process is used to determine each node of the tree, except that only a fraction of the records are used in subtrees (only those that have the attribute values of the branches leading to the node are active). For more information, see C4,5: Programs for Machine Learning [Qui93]

The Active Page version distributes the database between the pages. The Active Pages perform the counting, and the Central Processor accumulates the results and makes decisions. In addition, the Central Processor calculates the weights of records with unknown attribute values.

The count table and set of records with unknown attributes are both read and written by the Central Processor and Page Processor. The communication pattern does not, however, involve the same type of iteration as in JavaGC because the Central Processor does not continually poll all of the pages to find the one that needs to be serviced. For the floating point values, all of the pages must complete for the Central Processor to continue. For the count tables, the Central Processor waits for a Page Processor to complete, accumulates its count values, and starts the Page on the next attribute. The Central Processor waits at each page until it is finished.

5.4 Minima

Minima is the client kernel of a network-game which simulates a hypothetical network-based multiuser two- dimensional "RPG" style of game. In this style of game, there are several "live" objects on a two-dimensional map that move around and interact with each other. Here, we assume the clients each evolve the virtual world whenever non-random evolution can proceed. Further, it is assumed that random game events are coordinated across the network. In this application, the Active Pages evolve the virtual world while the Central Processor coordinates activity and migrates game objects across Active Pages.

Within the minima-online game simulation, communication between the Central Processor and Active Pages is coarse grained. The Central Processor reads and writes several scattered locations on each Active Page. In addition, a queue is maintained between the Page Processor and the Central Processor. These factors imply that coherence mechanisms that attempt to

Coherence Option	Implementation Variances	
	Base	Variance
Remapping	Remapped	Remapped, Non-Remapped
Bus Priority	Polling	Polling, Direct
Directory Size	Two Bits	Two Bits, One Bit

Table 2: Configurations Tested

Parameter	Value
CPU Clock	1 GHz
CPU Configuration	Single-Issue
L1 I-Cache	64K bytes
L1 I-Cache Associativity	
L1 D-Cache	64K bytes
L1 D-Cache Associativity	
L2 Cache	1M byte
L2 Cache Associativity	
L2 Line size	128 bytes
External Memory Speed	50 ns
Memory / Processor Bus	PC-100
AP Logic Clock	100 MHz
AP Line size / Bandwidth	256 bits
AP Cache size	4096 bytes
AP Cache Associativity	4
AP Cache Organization	Split I/D
AP Logic Element	Mini-RISC

Table 3: Active Page reference parameters

exploit only localized communication traffic will be insufficient.

6 Methodology

Our Active Pages system was simulated using sim3, an object-oriented derivative of SimpleScalar v2.0 toolset [BA97]. We simulated cycle-by-cycle the processors, caches, buses, and DRAMs. The Page Processor code was compiled using a standard C++ compiler with optimizations enabled.

Each application was run with the same machine parameters for all of the base configurations. The machine parameters are listed in Table 3. For comparison, we executed an “ideal” protocol that simulates coherence with zero cost. Explicit flushing is used for the flushing option. For each protocol, we varied the data size from 8 pages (4MB) to 128 pages (64MB).

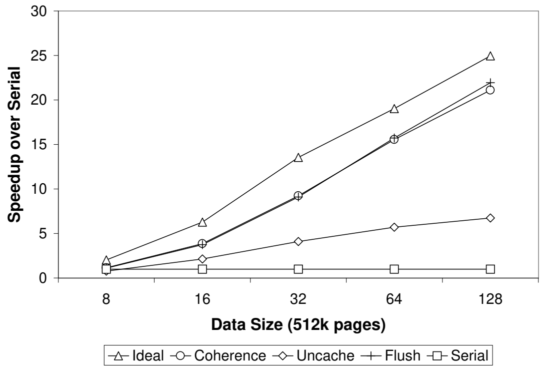

Figure 8: C4.5 with base configurations

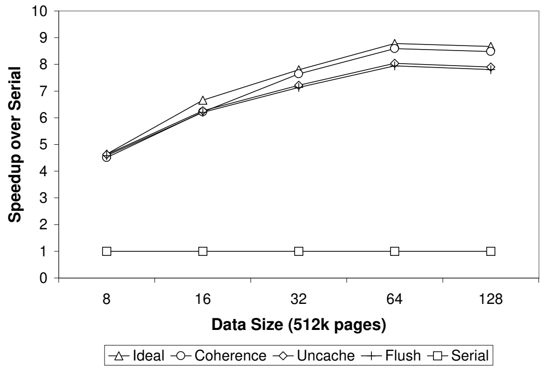

Figure 9: Bridge with base configurations

Name	Serial Computation	Page Processing Unit	Accumulation Time
C4.5	Medium	Medium	Low
JavaGC	Low	Low	Medium
Bridge	Low	High	Low
Minima	Medium	High	Medium

Table 4: Relative allocations of computation time to serial, parallel, and coordination tasks

7 Results

In this section, we present results to support five conclusions. We normalize our speedups to the speedup over serial, which is the same application run with a conventional memory system system rather than Active Pages. This is to show how the overall speedup changes as the problems scale. First, hardware coherence performs better than existing hardware only and as well or better than flushing. In addition, hardware coherence uses much less bandwidth than flushing. Second, our remapping mechanism significantly improves performance. Third, we find that polling for coherence requests saves hardware complexity with little or no degradation in performance. Fourth, a smaller one-bit entry in the pseudo-directory is sufficient to sustain acceptable performance for our system. Finally, processor saturation can be counteracted with multiple Central Processors.

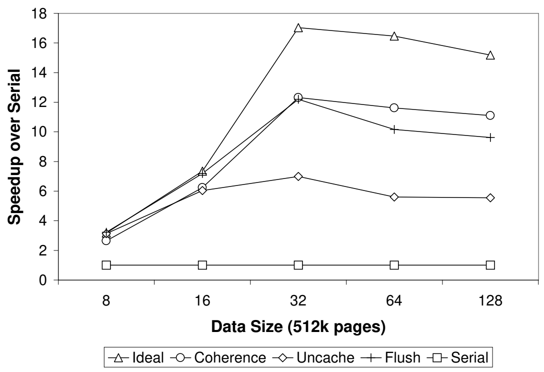

Figure 10: GC with base configurations

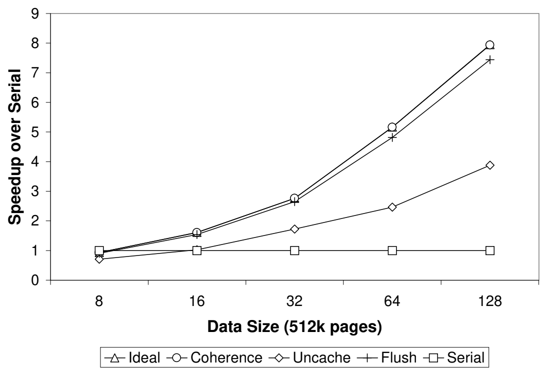

Figure 11: Minima with base configurations

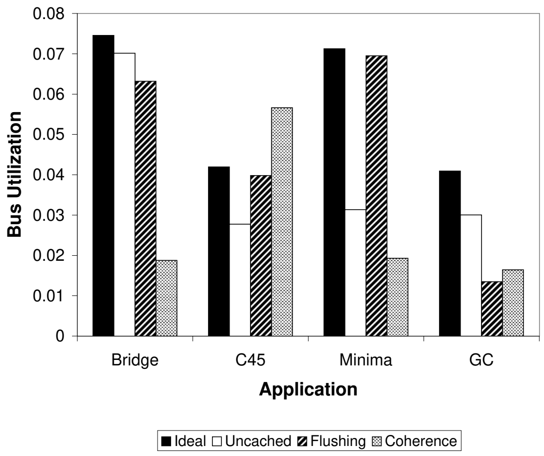

Figure 12: Bus Utilization of protocols at 128 page data size.

7.1 Coherence Protocols

Figures 8- 11 depict application performance when using different communication options for various problem sizes. Along with the available configurations, we plot an "ideal" protocol where coherence between Active Pages and Central Processor caches is maintained instantaneously and at zero cost in bus bandwidth. We observe that, for small problem sizes, nearly any choice of coherence protocol is sufficient. However, as problem size increases beyond 32 pages (16M-bytes), the difference between protocols becomes evident. This is expected, since small problem sizes will generate less coherence traffic. At larger problem sizes, we see that, in general, hardware coherence gets closest to ideal, then flushing, and then uncached regions. In addition, Figure 12 shows that when the coherence mechanism does not affect performance, hardware coherence uses much less bandwidth. The reason hardware coherence has higher utilization in GC is because it has such better performance.

The communication options affect the applications differently depending on their ratios of true serial computation, parallel computation, and time to consume (or accumulate) a result (See Table 4). C4.5 (Figure 8) and Minima (Figure 11) both have large phases of parallel work followed by large phases of serial computation. In between these phases, they communicate with the pages. This makes the communication itself slow down the processor, but the type of coherence used is not as important. In Minima, only the uncached regions configuration performs poorly.

In Garbage Collection, on the other hand, the Page Processor does very little work for each pointer it receives, so the ratio of communication time to computation time is very high. This fine granularity of communication makes the speed of the communication a critical component in overall execution time. The fine granularity also means that the Central Processor must perform more work per Page than more coarse-grained applications. The application quickly becomes processor-saturated. It does not gain from the extra pages since many Page Processors will be idle at one time waiting for the Central Processor to give them more work. It does, however, observe higher performance from coherence than explicit flushing.

Bridge (Figure 9) also scales poorly. This is for a different reason than GC. In Bridge, the problem size does not actually grow with the number of processors. It is still simulating a single game, starting with fourteen cards in each of four hands. We are hit by the traditional Amdahl's Law in which the increase in parallelism does not decrease the total runtime. In bridge, we have a certain number of potential hands that need to be simulated, and we just dispatch them to each Page Processor as it completes its previous hand. There is not enough parallelism to sustain

Name	Programmability	Bandwidth Needs	Performance	Cost
Hardware Coherence	++++	+++	++++	++
Explicit Flushes	+	+	++++	++++
Uncached Regions	++++	++	+	++++

Table 5: Coherence options and their desirability in several areas

128 pages. The communication option is not significant because the execution time to simulate a potential hand is much higher than accumulating the result from that hand. Communication is a very small proportion of total runtime.

Table 5 summarizes the comparison between coherence protocols. Performance alone is reason enough to count out uncached regions. Even though it is cheap and easy to program, it is not a viable solution. Between hardware coherence and explicit flushes, programmability and bandwidth needs are clearly better for hardware coherence. As shown in Figure 5, subtle changes in the positions of the flush statements can change the efficiency and correctness of the code. The performance is roughly equivalent between the two, whereas the cost is higher for hardware coherence. We think the bandwidth and programmability advantages in hardware coherence make coherence a better option than explicit flushing. For this reason, we will use hardware coherence in the rest of the results. First we will optimize the hardware coherence design, then we will explore adding Central Processors to alleviate the processor saturation problem.

7.2 Memory Remapping

We now look at how different design options for hardware coherence affect the performance of our applications. The first of these is the use of remapping. Figure 14 shows the speedup whereas Figure 13 shows the bus utilization. We note that most applications increase in performance when remapping is utilized, with the greatest gains being demonstrated by C4.5 and GC.

C4.5 obtains its remapping benefit because the count tables are zeroed by the Central Processor after it reads each line. With remapping, if the count table is fairly sparse, the Page Processor will not write to every line, and several lines will still be resident in the cache when the Page Processor completes its counting. This substantially decreases the time the Central Processor takes to read in the count tables. The bus utilization does not change dramatically because the total execution time is compressed. It is the total bus traffic that has been reduced, and utilization measures the bus traffic in time.

GC gets the large speedup because communication is such a high proportion of its total

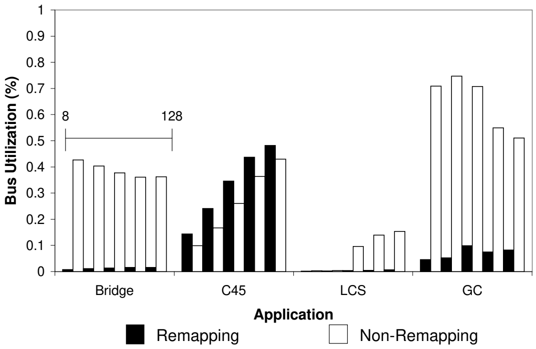

Figure 13: Bus utilizations of several protocols with and without remapping. Bar groups indicate data sizes of 8 to 128 pages.

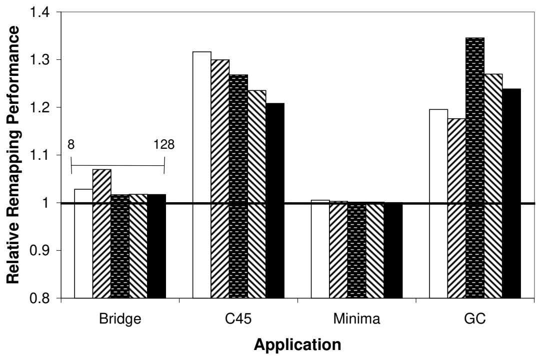

Figure 14: Speedup gained by remapping. Each is normalized to its unremapped time with data sizes 8 to 128 pages.

runtime. Garbage Collection and Bridge have the same polling pattern - they continually poll the different pages to find out which one is ready to be serviced. Because Garbage Collection is saturated when it does this, it experiences both a dramatic reduction in wasted bandwidth and a performance gain. Bridge experiences a bandwidth savings but because the Page Processing time is so large compared to the communication time, it does not experience much performance benefit.

Note that as the number of Active Pages increases, the effectiveness of remapping is reduced. That is because the success of remapping is dependent on holding data in the Central Processor cache, and as the data size is increased, the Central Processor can hold less in its cache. Even when the performance of remapping does not increase, though, the bandwidth decreases. This is important when we consider adding Central Processors to the system. For processor-saturated applications, the purpose of adding the processors is to increase the communication that goes on between the Central Processors and Page Processors. A low-bandwidth communication mechanism is desirable in this instance.

7.3 Polling

We compare system designs which depend upon different DRAM interfaces. As described, our basic implementation uses a modified memory controller which polls each Active Page DRAM for coherence requests and collects them in a queue. We now compare this to a more complicated implementation that allows DRAMs to place coherence requests directly on the bus. This

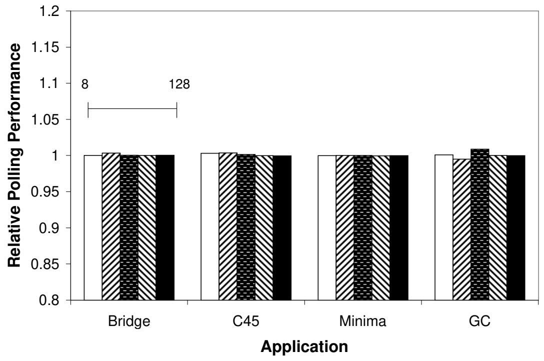

Figure 15: Speedup of DRAM-initiated coherence over Polling using Invalidate / Invalidate

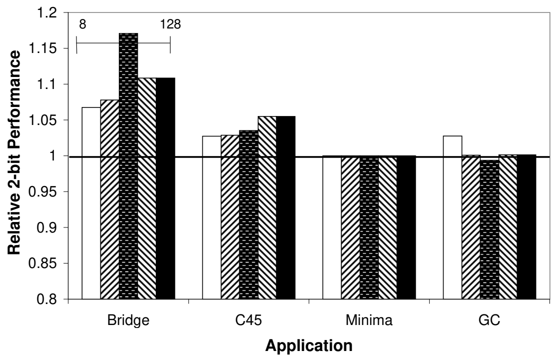

Figure 16: Speedup of a two-bit directory over a one-bit directory

hardware design choice can be recast as a question of priority; Do the programs perform better when the Central Processor or Page Processors have priority on the memory bus?

Note that our memory controller implements a polling policy that always favors the Central Processor. The memory controller only polls Active Page DRAMs for coherence requests when the bus is not being used by the Central Processor. We could schedule polling with higher priority, but we shall see that this is not desirable for Active Page computations.

If the bus between the memory controller and its DRAMs is idle often, then the polling and direct implementations are equivalent. In most cycles, the bus is free, so a Page Processor coherence request in the queue is usually scheduled in the next available slot. The only difference occurs when the bus utilization is high and Central Processor requests are occurring often enough to take bus cycles that a Page Processor coherence request would have used, delaying any polling to retrieve Page Processor requests.

Figure 15 depicts our results. The differences in execution time are negligible. There is less than a one percent performance difference. We conclude that the polling method is ideal because it requires a simpler transaction bus that is equivalent to the existing DRAM style of busses with no noticeable performance degradation.

7.4 Pseudo-Directory

Recall that our directory requires a single bit to tell if the Central Processor has read the line since the last time it was invalidated. A second bit can be used to distinguish between clean

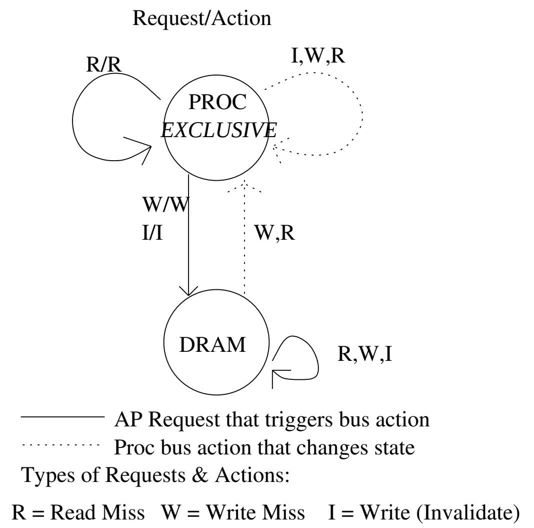

Figure 17: DRAM directory state diagram with one directory bit

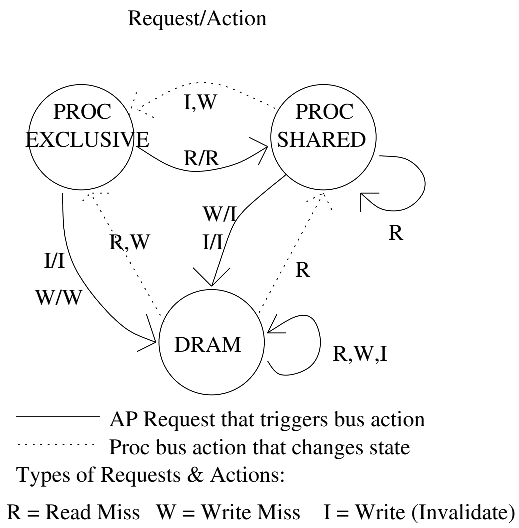

Figure 18: DRAM directory state diagram with two directory bits

and potentially dirty copies in the Central Processor's cache, possibly decreasing unnecessary coherence operations. Figure 17 shows the memory actions with one bit. The dotted lines are state transitions resulting from bus transactions initiated by the Central Processor. The solid lines represent requests from the Page Processor. The DRAM state indicates that the line is definitely not in the Central Processor cache. The PROC EXCLUSIVE state indicates that the line might be in the Central Processor cache. In this state, all Page Processor requests must be placed on the bus, since the Central Processor might have written to the line.

Figure 18 shows the actions taken with a two bit directory. The PROC SHARED state is added to the directory. This extra state leads to two cases in which the coherence operation will differ. First, if a read request arrives from the Page Processor and the line is in the PROC SHARED state, then the DRAM fulfills the request itself. There is no need to go to the Central Processor, because dirty lines in the Central Processor must be in the PROC EXCLUSIVE state in the DRAM. Second, if the line is in the PROC SHARED state and the Page Processor has a write miss, then the DRAM will issue an Invalidate rather than a write miss. Because the line is clean in the DRAM, there is no reason to request this line from the Central Processor cache; It only needs to invalidate the line in the Central Processor cache. This greatly reduces the latency of the coherence operation.

Note that in both of these protocols, the directory can be out of date. The DRAM is not notified when the processor evicts a clean line, so some Page Processor requests are returned with no data. The DRAM then supplies the line and transitions the directory to the DRAM state.

The worst-case scenarios for the two directories are much different. For the two-bit directory, we can limit the amount of coherence traffic. Each Central Processor read or write can produce a single coherence request from the Page Processor, placing it back in the PROC SHARED or DRAM state. With the one-bit directory, a Page Processor shared read goes to the Central Processor every time, erasing this limit. A pathological case could make the Page Processor go to the Central Processor every read. The results shown in Figure 16 show that only two applications benefit from the extra bit of storage, Bridge and C4.5. Both applications are not processor saturated, so the performance of the individual Page Processors is significant. These applications also have large data sections that are transferred to the Page Processor each time the Page Processor executes. This means that the Central Processor may have a lot of data in its cache that the Page Processor needs to read. While the Central Processor is polling, it still has data in its cache from the last time it wrote to the Page Processors. Once the Page Processor reads the data, the line is placed in the PROC SHARED state for the two-bit protocol, but remains in the PROC EXCLUSIVE state for the one-bit protocol. Consequently, subsequent reads from the Page Processor must obtain the line from the processor cache with the one-bit protocol, but may be serviced locally by the DRAM with the two-bit protocol. The extra bit speeds up the Page Processors, but does not degrade Central Processor performance. The gain is small in our applications, however, so we do not feel the extra space it requires is justified. Section 7.6 describes worst-case scenarios in which a 2-bit scheme may be desirable.

The importance of precision in the directory leads us to questioning the granularity of the directory information in the DRAM. If the DRAM keeps information per DRAM line, and the DRAM line is larger than the L2 cache line, then the information will be imprecise. The alternative would be customized Active Page DRAM chips for each L2 cache line size, which is not feasible economically. Because some imprecision does not affect the performance greatly, we can determine the granularity at which the DRAM will keep directory information, and if the L2 cache size is smaller, we will tolerate the imprecision.

7.5 Scaling

Now that we have obtained a low-bandwidth communication infrastructure, we can tackle one other limiting factor in scaling - *processor saturation.* We parallelize the coordination portion of each Central Processor code (in our applications, that is the only portion of the serial code that is

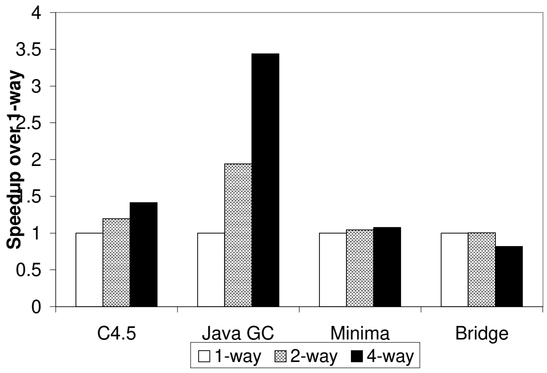

Figure 19: 1-, 2-, and 4-way SMP performance (128 Active Pages)

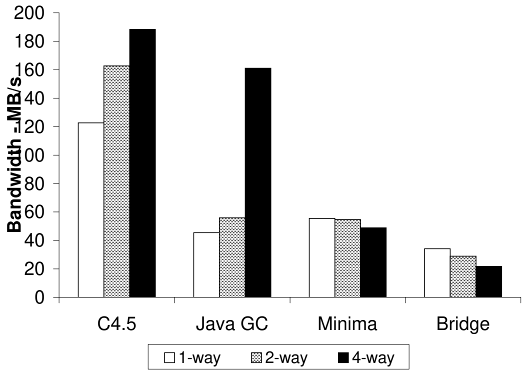

Figure 20: 1-, 2-, and 4-way SMP bus bandwidth requirements (128 Active Pages)

parallelizable, since we have the Page Processors for parallelism). Synchronization is performed through locks and barriers. Figure 19 compares the execution time of 1-, 2-, and 4-way SMP systems with an Active Page memory. Hardware coherence and memory remapping are used. We note that Java garbage collection obtains the most benefit from additional processors. This is directly due to the fact that Java is in a saturated state with a single Central Processor and a 64 page problem size. Our networked game example sees only marginal performance gains since it is unsaturated. Any gains are due to the parallelization of the former serial components of the code. C4.5 sees marginal improvement because the Page Processing time varies, so at times the Central Processor can be saturated. Bridge is interesting because it actually observes a slowdown at some points. This is caused by the combination of high Page Processing time and high variability in Page Processing time. The high Page Processing time makes the Central Processor computation time insignificant in the final equation, so parallelizing the Central Processor coordination code does not gain much. The high variability in Page Processing times means that changing the order in which tasks are executed can greatly affect the runtime. When we split up the coordination, we split up the tasks between Central Processors, changing the order in which the tasks will be executed.

Scaling magnifies the differences between protocols because the bandwidth needs are greater and the programming of explicit flushes in an SMP environment is exponentially more difficult. Figure 20 depicts the bus bandwidth of our single processor and SMP systems. Adding additional Central Processors to tolerate saturation requires additional memory bus bandwidth when we obtain a speedup. We expect that our existing bandwidth of 400 Mb/sec will be suitable for up to an 8-way SMP, but high performance busses will be required for larger parallel systems that use Active Page memory chips. This reaffirms our decision to use memory remapping for hardware coherence and to use hardware coherence rather than explicit flushes.

Not only do explicit flushes scale poorly in an SMP environment, they are also even more difficult for the programmer to use properly. An explicit flush only expels a cache line from one Central Processor's cache. This works fine for lines that have been written, because at the time the line is written, it can only be in one cache. Once the line is placed in another cache or expelled from the cache, it can be written back to the DRAM. If the flushes are being used to obtain data that the Page Processor wrote, however, it is more difficult. If two Central Processors have the line in their caches, then expelling one will only obtain the copy from the other Central Processor's cache, not from the DRAM or Page Processor's cache. This means that the programmer must guarantee that these lines are not in more than one Central Processor cache.

7.6 Discussion

In retrospect, the problem of cache coherence for Active Page systems can be solved with surprising efficiency. It is somewhat unexpected that our demanding applications exhibit less than 1% observed coherence traffic overhead. Are these poor applications or is there a more fundamental phenomenon? In fact, the rate at which the Central Processor creates sharing is limited by the DRAM latencies inherent in intelligent memory systems.

The following analysis assumes a 2-bit directory rather than the 1-bit scheme that we found was adequate for our applications. In the worst case, it is important to distinguish between clean and dirty data shared between Page Processors and Central Processors. We also assume that Active Page applications do not cause data to thrash between Central Processors and Active Pages. Since applications must be carefully partitioned and synchronized, this seems a reasonable assumption.

Suppose we set up a situation where we try to maximize the aggregate coherence overhead of an Active Pages application. Such an application would have the main processor stride-read cache-lines from the Active Page DRAM into the processor cache. This would cause the Active Pages to mark these cache lines as shared. The processor would be able to pipeline memory requests, and the DRAM would support multiple open sub-array banks and pipelined memory accesses. The processor would read from one bank, then the next, and so on, in sequence. The memory controller would pipeline these requests to the open DRAM sub-arrays, thus hiding the DRAM access latency. In this contrived example, the Active Pages follow after the processor, writing back to the same cache lines, but slightly after the processor reads them. This causes an invalidate to be sent to the processor. This example creates the highest possible sustained aggregate coherence traffic on the memory controller/DRAM bus given our assumptions.

Using our machine parameters, we can estimate the maximum coherence traffic in this worst-case example. In our machine, we utilize an L2 cache with a line size of 128 bytes. The DRAM bus supports transfers of 8 words per clock cycle. The memory latency is 5 DRAM bus cycles (50ns). Using pipelined reads with overlapping memory requests from the processor, we can conclude that:

$$\text{MemoryReadRequest} = 5 + 128/8 = 21 \text{ cycles}$$

Pipelining these requests we can achieve an aggregate access time of:

$$\text{PipeliningMemoryReadRequest} = 1 + 128/8 = 17 \text{ cycles}$$

This can create *at most* a single coherence request. This coherence request needs to be scheduled into the bus. Utilizing the polling mechanism of the memory controller, this request requires 2 cycles to transfer the request to the processor, and 1 cycle to transfer the result back to the DRAM. Assuming that additional cycles required to access the L2 cache to satisfy this request are overlapped with memory requests, we find that each memory request from the processor requires:

$$\text{PipelinedMemoryReadwithOverhead} = 17 + 3 = 20 \text{ cycles}$$

This implies that the overhead on the bus required to support coherence is a mere 3/20 cycles or 15%. Our applications show substantially less coherence traffic overhead than this, but this is clearly a contrived example designed to demonstrate the *maximum* possible coherence traffic overhead. It is interesting, though, that coherence traffic requires no more than 15% of the total aggregate bus bandwidth between memory controller and Active Page DRAM, suggesting that coherence is an inexpensive addition to the Active Page model of computation.

8 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 [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] partitions computation at loop-level granularity and distributes the loop iterations in a hierarchical manner. The Impulse project [C+99] uses transformations in the memory controller with similar goals to Active Pages, but without the goal of supporting computation in memory.

FlexRAM[KHY+99], a latter project to Active Pages, has a similar programming model but differs in two major ways. First, in addition to the page-processors, FlexRAM provides a processor on each DRAM chip that can address all the memory in the DRAM. Thus coordination and communication are performed on-chip rather than in the Central Processor. Second, the DRAM chip does not conform to the current chip packaging because of the addition of numerous pins.

Intelligent disks [AUS98] [G+98] have also been proposed. These systems are analogous to Active Pages, only at a different level. These systems are meant for streaming applications that do not fit in main memory, just Active Pages work well for applications which have working sets larger than the processor cache. Their other motivating applications are filter applications, like database queries, that perform preliminary computations to limit the amount of data that the processor receives. 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.

Several projects have altered cache coherence protocols to fit the development of hybrid architectures. The Stanford DASH project [LLG+90] combined a directory protocol and bus-based protocol. The directory is used for remote accesses, whereas local traffic is kept coherent by the snoopy-bus. The GLOW cache coherence protocol extensions provide a hardware mechanism to efficiently share data on different network topologies [KG96]. CACHET is a machine-independent compiler-directed cache coherence scheme [SAR99].

Network Interface cards have had a similar problem with performance degradation due to uncached variables. CNI [MFHW96] proposed using cache coherence protocols with the Network Interface card to allow data to be cached. We took a similar approach in adding an interface to the bus that allowed the DRAM to perform cache coherence requests in order to make data flow properly from the cache to the DRAM.

Our remapping scheme is similar to page coloring, which has been a part of compiler research. The focus is the same: if two pages are likely to be held in the cache at the same time, they should be mapped to different areas. The mechanism is different, however. Page coloring has the compiler give hints to the operating system so that the pages are placed correctly. In our case, the page placement is not variable, so the compiler uses a separate, unused address space to give new virtual and physical addresses to the same data.

For the applications we have examined, the Active Page programming model has proven highly successful in achieving high performance with low implementation cost. The key is in providing efficient support of both serial and parallel computation. It would be interesting to compare the Active Page approach to more uniform parallel processing substrates. Traditional parallel processors are too far beyond the cost domain of Active Page systems, but technologies such as simple chip multiprocessors may produce systems at least within an order of magnitude. While some applications may be fully parallelizable, we speculate that serial computations will remain an important factor in future systems.

9 Conclusion

Active Pages is a page-based computational model for intelligent memory which can lead to substantial performance benefits while limiting data sharing. Two factors can limit performance as we scale the number of Page Processors - communication costs and Central Processor coordination time. This study found that the classic Invalidate protocol with a hybrid directory and bus-based mechanism was best suited for this environment, providing the best balance between programmability, performance, bus bandwidth use and cost. The simplest hardware scheme was adequate for performance - we can use a one-bit directory entry and providing hardware queues on each DRAM that the memory controller polls for coherence requests. We also implemented a remapping of Active Page memory for the Central Processor so that it could better utilize its cache. Finally, we demonstrated scaling of Central Processors and coherence capabilities in an SMP system. Overall, our results show that Active Page systems can perform well on large problems with fine-grain communication requirements.

References

[ACK94] Abhaya Asthana, Mark Cravatts, and Paul Krzyzanowski. Design of an active memory system for network applications. In *International Workshop on Memory Technology, Design and Testing*, pages

58–63. IEEE Computer Society Press, 1994.

[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.

[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.

[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.

[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.

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

[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.

[Int97] Intel. *Intel Architecture Software Developer's Manual*. Intel, 1997.

[KG96] Stefanos Kaxiras and James R. Goodman. The glow cache coherence protocol extensions for widely shared data. In *Internation Conference on Supercomputing*. ICS, 1996.

[KHY+99] Yi Kang, Michael Huang, Seung-Moon Yoo, Zhenzho Ge, Diana Keen, Vinh Lam, Prattap Pattnaik, and Josep Torrellas. Flexram: Toward an advanced intelligent memory system. In *International Conference on Computer Design*. ICCD, October 1999.

[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.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *17th International Symposium on Computer Architecture*. IEEE, May 1990.

[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.

[MFHW96] S. Mukherjee, B. Falsafi, M. Hill, and D. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23th Annual International Symposium on Computer Architecture 1996*. ACM, May 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.

[OHK+99] Mark Oskin, Justin Hensley, Diana Keen, Frederic T. Chong, Matthew Farrens, and Aneet Chopra. Exploiting ilp in page-based intelligent memory. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, November 1999.

[OKH$^+$00] Mark Oskin, Diana Keen, Justin Hensley, Lucian-Vlad Lita, and Frederic T. Chong. Reducing cost and tolerating defects in page-based intelligent memory. In *Proceedings of the 2000 IEEE International Conference on Computer Design (ICCD'00)*, Austin, Texas, 2000.

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

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, 1993.

[SAR99] X. Shen, Arvind, and L. Rudolph. Commit-reconcile and fences (crf): A new memory model for architects and compiler writers. In *26th International Symposium on Computer Architecture.* ISCA, May 1999.

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