To Appear in IEEE Transactions on Computing

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.

The Active Page model inherently limits sharing, keeping coherence tractable, but exacerbates saturation problems. We first present a hybrid snoopy/directory protocol for use in Active Pages. Limited sharing allows for a low-latency, low-bandwidth hybrid protocol. A transparent remapping mechanism is added for efficient caching. On smaller data sizes, explicit flushing and hardware coherence exhibit similar performance, but hardware coherence is easier to program and uses less bandwidth. Finally, we examine SMP multiprocessor systems to mitigate saturation effects. As the number of threads increases, the bandwidth needs increase, making hardware coherence even more attractive.

Keywords: intelligent memory, merged DRAM logic, cache coherence

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 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 accelerate computations on commodity workstations and PCs. A fundamental limitation to any intelligent memory system, however, is maintaining data coherence between processor caches and intelligent memory[KCH02].

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

Active Pages is a page-based model of computation which associates simple functions with each page of memory. Active Page systems have embedded small processors, Page Processors, with each sub-array of memory in DRAM. Each Page Processor may only access the memory associated with it, whereas the conventional processor, or Central Processor, may access the complete address space.

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 Page Processors. At large problem sizes, the bandwidth consumed by flushing may become prohibitive, and non-optimally placed flushes can degrade performance. Because Active Pages 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. The goal is to integrate Active Pages with standard systems, so the existing bus-based system protocol should not change, and bus-based protocols have well-known performance drawbacks in such large-scale environments.

In this paper, we make three primary contributions. First, we present a protocol and system design which efficiently supports coherence with minimal modification to commodity systems. We also introduce a transparent, cache-conscious remapping mechanism that allows data to be easily scattered to multiple Active Pages. Finally, we show that applications that are Processor-saturated benefit from additional Central Processors.

2 Active Pages

In this section, we give a brief description of the Active Pages system so that the reader can better understand the cache coherence issues in a specific intelligent memory system. Further details can be found in [Kee02].

2.1 DRAM Hardware Design

Current high-density DRAMs are divided into sub-arrays, complete with row and column decoders [I^+97]. Our proposed Active Page implementation exploits this natural structure,

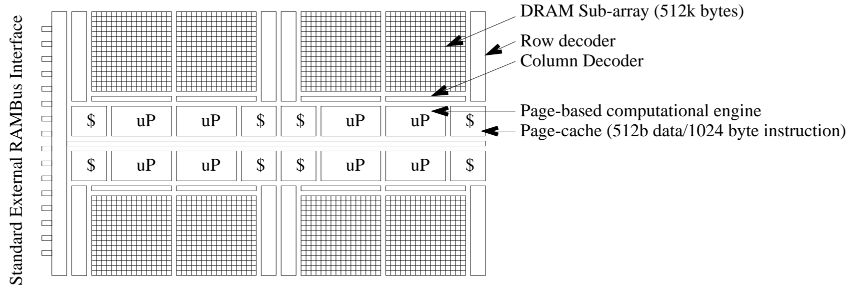

Figure 1: Active Page architecture (8 pages)

treating each sub-array as an Active Page. As shown in Figure 1, small VLIW processor and cache are embedded next to each sub-array to implement Active Page functions [OHK+99]. With a sub-array size of 512K-bytes, the embedded processors are expected to consume less than 31% of chip area. The operating system allocates Active Pages aligns the data within the sub-array, loads the proper instructions, and activates the Page Processor.

To minimize DRAM modification and reduce hardware overhead, our proposed Active Page designs do not provide hardware support for direct communication between Active Pages. Instead, 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 processor-mediated approach can have several advantages. First, Active Page DRAMs are kept simple. Direct inter-page communication would require a communication network and arbiters to transfer the messages, handlers that place overflow into system or the process' memory, and virtual memory support in the Page Processors. Second, the operating system maintains control of virtual memory because all accesses and translations go through the Central Processor. Without this, additional synchronization would be required to prevent a Page Processor from communicating with another Page Processor that is being swapped out to disk. Finally, sharing occurs only between the Central Processor and the Active Pages, never directly between Active Pages. With unlimited sharing, one Active Page could hold lines from all Active Page DRAMs. This last advantage is critical to our inexpensive designs for maintaining coherence.

Our work builds upon a wealth of previous projects in cache coherence, parallel process-

ing, and intelligent memory. Although we do not have space to list these projects in this short paper, we refer the reader to [Kee02] for a thorough discussion of related work.

3 Coherence Design Issues

Our primary goal is to design and evaluate an efficient implementation of coherence supported in hardware. We will then compare this implementation to other options. The design of our protocol is influenced by two main factors. Each component altered costs design and verification time and a loss of commodity[1]. In addition, our programming model is an asymmetric model in which there is one Central Processor and many Page Processors. We begin with our basic hybrid protocol and follow with an important data placement optimization.

3.1 Hybrid Coherence Protocol

The Central Processor, running a MESI protocol (used in Intel Pentium II processors [Int97]), is left unaltered. From the Central Processor's point of view, then, the system must run a bus-based snoopy protocol. The Page Processors also use the MESI protocol, but from their points of view, a directory protocol is desirable to reduce unnecessary bus traffic. In light of those constraints, two fundamental design questions arise. 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, coherence messages are limited 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 Processors run a snoopy protocol, so they will snoop the request off of the bus. The Memory Controller acts as an agent for the Page Processor, setting the signal on the coherence line to indicate it is not ready to respond. It directs the request to the appropriate Page Processor and begins a

[1] Loss of commodity is the cost increase due to low sales volume

read operation in parallel (if the coherence operation includes a miss). The Page Processor checks to see if that cache line is in its cache. Sharing between Page Processors would require broadcasting every request or accessing a large directory to route the request to the appropriate Page Processor(s). The Memory Controller responds when it receives the sharing information from the DRAM. The DRAM request is canceled if another Central Processor can satisfy the request.

On the DRAM side, only Page Processor requests to shared data should reach the bus because there are potentially hundreds of Page Processors running at once. For Page Processor coherence operations, a single bit indicates whether a line of data has been read into a Central Processor's cache. If the data is shared, the Page Processor places the request in a hardware queue in the DRAM, and the Memory Controller places those requests on the bus.

Finally, operations must be serialized. If a protocol has no serialization point, then some writes can be lost in the system. For example, if two processors invalidate the same line at the same time, and they both receive the other's invalidation request after they have performed their writes, then both lines will be invalidated, losing both changes. Instead, one must complete, and the other must become a read-invalidate to obtain the changes from the first. In a snoopy protocol, the serialization point is the bus, and each processor has the necessary hardware to match new requests with outstanding requests. In a directory protocol, the serialization point is the directory. The serialization point is not as obvious in this design because Central Processors snoop the bus (which the Page Processors can not), and Page Processors use a directory. Page Processors place coherence requests in a hardware queue for the memory controller to place on the bus. 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, but otherwise the Page Processor stalls until its request reaches the bus (and in some cases completes).

The Central Processor already has the hardware necessary to reconcile requests for the same address by bus snooping. For the Page Processors, conflict-resolution is performed 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 Central Processor invalidate, it will find the Page Processor 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 occurs later because it reaches the bus after the Central Processor 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 Page Processors through some synchronization variable present on each page. For example, if the Central Processor is trying to determine which Pages Processors have completed a particular function, it may poll a "done" variable on every page.

Unfortunately, with power-of-two page sizes and sync variables at the same offset in every page, all sync variables map to the same location in the cache, as seen in Figure 2. Varying the offset of these variables would involve extra indirections to calculate offsets as a function of Active Page number.

With hundreds of Page Processors running, the Central Processor can not store all of these sync variables at the same time. Without hardware coherence, this is not a performance problem since the processor can not cache any of these variables (since they might have been changed by the Page Processors). With coherence, one would expect the Central Processor to poll on cached copies and only fetch new copies of variables that have changed. A crucial observation is that the Page Processors themselves are not adversely affected by having their data at a constant offset. Only the Central Processorneeds to avoid constant offsets for these sync variables.

Page coloring is a typical mechanism to alleviate conflict misses due to data placement. Unfortunately, once the number of Page Processors exceeds the capacity of the cache divided by the page size, the same problem arises. We present a scheme which allows interleaving at any power-of-two cache line granularity.

Figure 2: Mapping of variables into cache before and after remapping.

Figure 3: Remapping operation

Remapping is an address transformation implemented in the memory controller that gives these sync variables more cache-friendly addresses for the Central Processor. 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 2, and then convert that result to the DRAM physical address. In reality, we can skip the step from physical to virtual, removing the need for a reverse TLB. Only the bit manipulation and one translation (physical-to-physical) are performed, but the virtual address information must be saved 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 is sent up-stream to the processor cache.

To use this feature, a keyword "cached" is introduced into the compiler for our system. 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. "Cached" is only a compiler hint; If there are too many variables for the region, the compiler places some variables into the non-remapped memory space.

4 Coherence Alternatives

With our hardware coherence protocol as one option, we now describe other options for comparison. All systems assume a write-through L1 cache and write-back L2 cache.

Uncached regions require no change to the Central Processor, just the compiler. The programmer designates certain variables "uncached", similar to how programmers use "volatile." The compiler must map these variables to uncached virtual pages (The Pentium allows each virtual page to be marked uncached). 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.

Uncached regions are desirable because the uncached keyword can be easily added to compilers, is intuitive for the programmer, and requires no changes to the hardware.

The disadvantages are correctness difficulties and performance. Although the volatile keyword is intuitive for the programmer, it is the source of many programming errors, as with the uncached keyword. If the programmer forgets to make a shared variable uncached, then the program may not be correct. Uncached regions also have a performance disadvantages due to write-through latencies and inability to exploit both temporal and spatial locality.

Explicit flushes require the programmer to decide when to synchronize the data. An explicit flush writes dirty data back to the DRAM and invalidates that line from the Central Processor cache. Previous Active Pages publications used explicit flushes. Each time an Active Page function is called, the Central Processor cache is 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. Unfortunately, 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 if flushes are placed too far after a write. This makes the placement of the flushes crucial to program correctness. Program correctness is, however, always possible. If

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: Partitioning of applications between Central Processor and Active Pages

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

Parameter	Value
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 2: Active Page reference parameters

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.

Hardware coherence was described in Section 3.1. The processor in our reference system uses the MESI protocol as in Pentium II [Int97]. The Central Processors run a snoopy protocol, whereas the Page Processors utilize a directory containing one bit per line.

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 (Minima). 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.

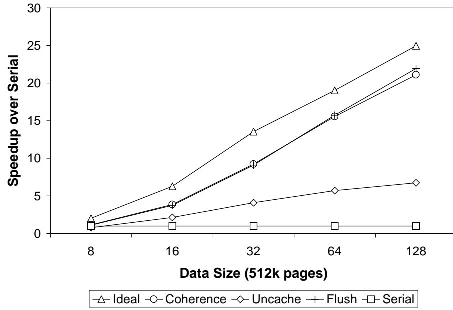

Figure 4: C4.5 coherence performance

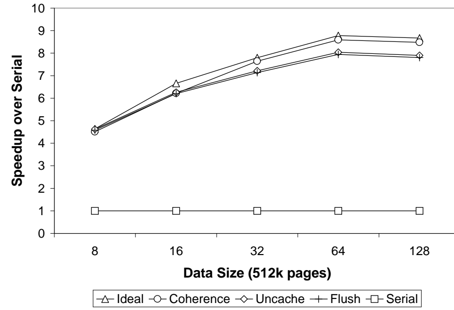

Figure 5: Bridge coherence performance

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. The machine parameters are listed in Table 2. For comparison, we executed an "ideal" protocol that simulates coherence with zero cost. For each protocol, data size was varied from 8 pages (4MB) to 128 pages (64MB).

7 Results

In this section, we present results to support three conclusions. First, hardware coherence performs better than existing hardware only (uncacheable regions) and as well or better than explicit flushing. In addition, hardware coherence uses much less bandwidth than flushing. Second, our remapping mechanism significantly improves performance. Finally, processor saturation can be counteracted with multiple Central Processors.

7.1 Coherence Protocols

Figures 4-7 depict application performance when using different communication options for increasing problem sizes. Results are given as the speedup over serial, which is the same application run with a conventional memory system system rather than Active Pages. 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

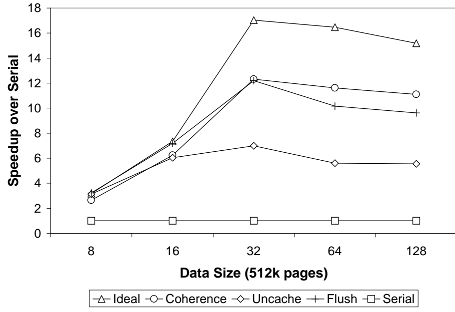

Figure 6: JavaGC coherence performance

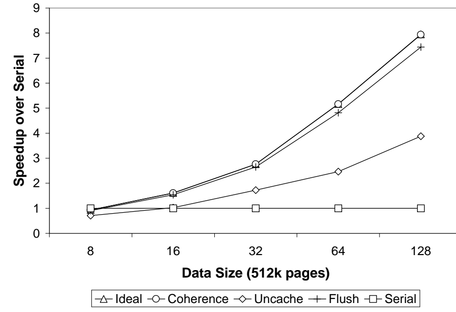

Figure 7: Minima coherence performance

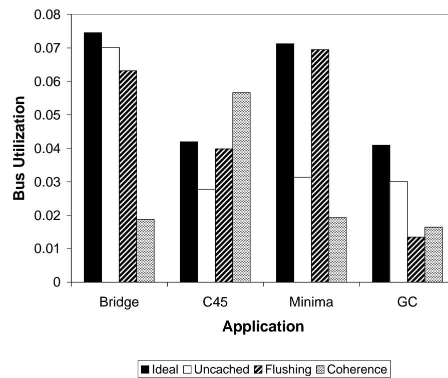

Figure 8: Bus Utilization of protocols at 128 page data size.

bandwidth. 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. At larger problem sizes, in general, hardware coherence gets closest to ideal, then flushing, and then uncached regions. In addition, Figure 8 shows that when the coherence mechanism does not affect performance, hardware coherence uses much less bandwidth. The reason hardware coherence has higher utilization in JavaGC 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. C4.5 (Figure 4) and Minima (Figure 7) 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 JavaGC, 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 5) also scales poorly. This is for a different reason than JavaGC. 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. The traditional Amdahl's Law comes into play because the increase in parallelism does not decrease the total runtime. In bridge, there are a certain number of potential hands that need to be simulated, and Central Processor dispatches one to each Page Processor as the Page Processor completes its previous hand. There is not enough parallelism to sustain 128 pages. The communication option is not significant because the execution time to simulate a potential hand is much higher than

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

Table 3: Coherence options and their desirability in several areas

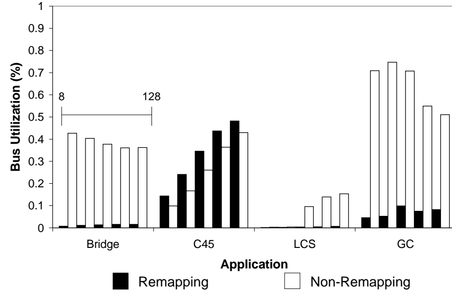

Figure 9: Bus utilizations with and without remapping with increasing data sizes.

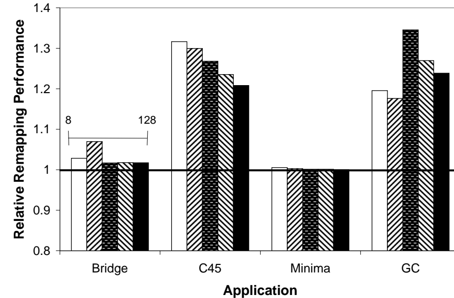

Figure 10: Speedup gained by remapping with data sizes 8 to 128 pages.

accumulating the result from that hand. Communication is a very small proportion of total runtime.

Table 3 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. The performance is roughly equivalent between the two, whereas the cost is higher for hardware coherence. The bandwidth and programmability advantages in hardware coherence make coherence a better option than explicit flushing, especially since scaling Active Pages will only make the bandwidth needs greater. For this reason, hardware coherence is used in the rest of the results. First memory remapping is used to enhance the coherence design, then Central Processors are added to alleviate processor saturation.

7.2 Memory Remapping

We now evaluate the performance impact of memory remapping. Figure 10 shows the speedup whereas Figure 9 shows the bus utilization. Most applications increase in performance when remapping is utilized, with the greatest gains being demonstrated by C4.5 and JavaGC.

C4.5 obtains its remapping benefit because the count tables are zeroed by the Central

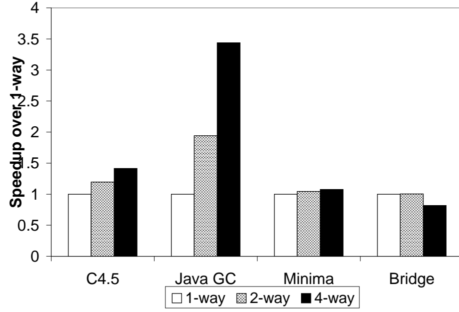

Figure 11: 1-, 2-, and 4-way SMP performance (128 Active Pages)

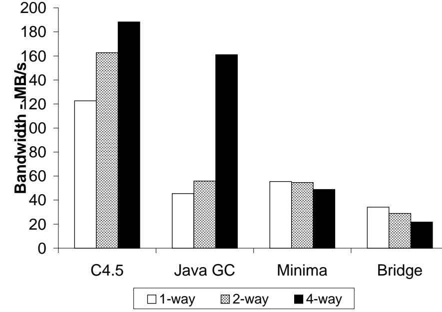

Figure 12: 1-, 2-, and 4-way SMP bus bandwidth needs (128 Active Pages)

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. The total bus traffic that has been reduced, and utilization measures the bus traffic in time.

JavaGC gets a large speedup because communication is such a high proportion of its total runtime. JavaGC and Bridge have the same polling pattern - they continually poll the different pages to find out which one is ready to be serviced. Because JavaGC 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.

As the number of Active Pages increases, the performance benefit of remapping decreases. That is because the success of remapping is dependent on holding data in the Central Processor cache, and as the data size is increases, the Central Processor can hold less in its cache. Even when the performance of remapping does not increase, though, the bandwidth needs decrease. This is important when considering adding Central Processors to the system.

7.3 Scaling

Now that we have obtained a low-bandwidth communication infrastructure, we can tackle one other limiting factor in scaling - processor saturation. The coordination portion of each

Central Processor code is parallelized (in our applications, that is the only portion of the serial code that is parallelizable, since the Page Processors are used for parallelism). Synchronization is performed through locks and barriers. Figure 11 compares the execution time of 1-, 2-, and 4-way SMP systems with an Active Page memory. Hardware coherence and memory remapping are used. JavaGC obtains the most benefit from additional processors. This is directly due to the fact that JavaGC 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 coordination is split up, the tasks are split 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 12 depicts the bus bandwidth of our single processor and SMP systems. Adding Central Processors to tolerate saturation requires additional memory bus bandwidth when speedup is obtained. 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.

8 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

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

[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.

[KCH02] D. Kim, M. Chaudhuri, and M. Heinrich. Leveraging cache coherence in active memory systems. In Proceedings of the 16th International Conference on Supercomputing, June 2002.

[Kee02] Diana M. Keen. Novel Designs and Uses of Communication in Auxiliary Processing Systems. PhD thesis, UC Davis, 2002.

[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.

[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.

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