SMTp: An Architecture for Next-generation Scalable Multi-threading

Mainak Chaudhuri
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853
mainak@csl.cornell.edu

Mark Heinrich
School of Computer Science
University of Central Florida
Orlando, FL 32816
heinrich@cs.ucf.edu

Abstract

We introduce the SMTp architecture—an SMT processor augmented with a coherence protocol thread context, that together with a standard integrated memory controller can enable the design of (among other possibilities) scalable cache-coherent hardware distributed shared memory (DSM) machines from commodity nodes. We describe the minor changes needed to a conventional out-of-order multithreaded core to realize SMTp, discussing issues related to both deadlock avoidance and performance. We then compare SMTp performance to that of various conventional DSM machines with normal SMT processors both with and without integrated memory controllers. On configurations from 1 to 32 nodes, with 1 to 4 application threads per node, we find that SMTp delivers performance comparable to, and sometimes better than, machines with more complex integrated DSM-specific memory controllers. Our results also show that the protocol thread has extremely low pipeline overhead. Given the simplicity and the flexibility of the SMTp mechanism, we argue that next-generation multithreaded processors with integrated memory controllers should adopt this mechanism as a way of building less complex high-performance DSM multiprocessors.

1. Introduction

Hardware multi-threading and integrated memory controllers are two recent trends in commodity high-performance computing. The former, in the form of hyper-threading (HT) or simultaneous multithreading (SMT) [41], appears in the Intel Xeon and Pentium 4 [21, 27], and the IBM POWER5 [17]. The latter appears in Compaq's 21364 [7], the IBM POWER5 [11], the AMD Opteron [18], and the Sun UltraSPARC III and IV [39, 40]. Multi-threaded architectures help address the growing need for memory-level parallelism, while integrated memory controllers reduce both memory latency and system cost.

Multiprocessor systems built from microprocessors with integrated memory controllers are naturally distributed memory machines (each processor has its own local memory). For example, the AMD Opteron connects up to 8 processors via their coherent HyperTransport interconnect to form a snoop-based distributed shared memory (DSM) non-uniform memory access (NUMA) machine [18]. Scaling such machines to higher processor counts requires only the introduction of a switch-based network fabric capable of routing the low-level messages that comprise the cache coherence protocol between the integrated memory controllers. Examples of low-latency, high-bandwidth switched fabrics include the Craylink fabric used in the SGI Origin machines [9], "raw" mode in InfiniBand [16], and PCI Express AS [33]. However, continuing to snoop in large-scale DSM machines is problematic both in terms of bandwidth and complexity, and machines typically instead employ directory-based coherence protocols that track sharing information in the system and send point-to-point messages to enforce coherence [1, 2, 22, 23, 24, 25, 26, 30].

In fact, even implementing the snoop-based coherence protocol at small-scale in this natural DSM configuration is awkward. Unfortunately, directory-based coherence protocols have traditionally required more sophisticated memory controllers with complex state machines that inspect directory entries and initiate the proper responses to processor, network, and I/O requests. Often the protocols can be so complex that the memory controller implementation itself is programmable and has an embedded processor (or processors) that implements the protocol control [3, 22, 26, 30]. These DSM architectures are scalable and high-performance, but their reliance on complex memory controllers that contain functionality unneeded in uniprocessor or small-scale shared-memory systems has relegated these machines to low-volume (and therefore high-cost) production.

With the trend toward integrated memory controllers, one can ask how do you design the next-generation scalable DSM machine? Now modifying the memory controller not only has the old DSM disadvantages in complexity and cost, but it also means modifying the processor! Furthermore, a programmable DSM memory controller would now find itself in the strange position of having an embedded processor on the processor.

In this paper we combine the advantages of the trends of SMT and integrated memory controllers to show how a modern SMT processor with a normal integrated memory controller can implement a high-performance scalable cache-coherent DSM. We introduce *SMTp*—an *SMT* processor with a *p*rotocol thread that is invoked on the main processor on an L2 (or whatever the last level of cache is before main memory) cache miss in parallel with fetching the cache line data from the memory system. The protocol thread can then be used to implement the directory-based

cache coherence protocol necessary to build large-scale hardware DSM machines from suitably-connected commodity machines. We describe the small number of micro-architectural extensions necessary to implement SMTp in Section 2 and the reasons those changes are necessary.

Two of the most exciting features of an SMTp architecture are that the protocol thread need not be restricted to implementing basic coherence protocols, and that its benefits are not restricted only to multiprocessors. Schemes such as active memory address re-mapping [20] or fault tolerance schemes like ReVive [34] that are extensions of the coherence protocol can now be implemented as protocol threads. In addition, the SMTp mechanism effectively provides software control of the cache re-fill mechanism allowing the protocol designer the potential to transparently manipulate data. This could give rise to coherent encryption or compression schemes that are transparent to the programmer, among other possibilities.

In this paper though, we focus squarely on the architectural extensions necessary to implement SMTp and we study the performance impact of running a cache coherence protocol thread on the SMT processor across a variety of shared-memory parallel applications for several machine configurations. Specifically, we examine the impact of running a protocol thread (as needed) on an SMT processor that has 1, 2, or 4 application threads per node, and we simulate up to 32-node configurations (64 threads total). We compare the performance of SMTp to directory-based DSM machines with normal SMT processors and: (1) a non-integrated memory controller (e.g. SGI Origin); (2) a "perfect" integrated memory controller (to show an upper bound on performance); and (3) two more realistic integrated memory controller configurations. We find that SMTp always performs better than DSMs constructed from non-integrated memory controllers and performs at least as well (and sometimes better than) realistic implementations with integrated controllers. Further, we show that as the processor clock rate continues to outpace those of the rest of the system, SMTp maintains its excellent performance.

The goal of the evaluation in this study is not to show that SMTp beats an "all hardware" approach to designing processors with integrated DSM memory controllers that support directory-based cache-coherence (though it sometimes does!). Our key point is that if SMTp machines perform at least as well (or even almost as well) as these machines, then why design them? Instead, use the normal integrated memory controller that is already present on commodity processors, use the high-speed interconnect that is either already connected to the memory controller or coming soon, and make the few micro-architectural changes necessary for SMTp. Let the business of complex coherence protocols be handled in software by the protocol thread and gain the ability to construct hardware DSM machines from commodity nodes [15] and the ability to expand the protocol thread mechanism to support additional techniques beyond basic cache coherence.

The rest of the paper is organized as follows. In Section 2 we describe the basic SMT extensions needed to support SMTp. We also discuss changes necessary for deadlock avoidance and protocol thread performance. Section 3 describes the simulation environment we use in our evaluation. Section 4 shows the performance of SMTp relative to a wide range of machine configurations and sizes. In Section 5 we discuss related work and we conclude in Section 6.

2. SMTp Architecture and Mechanisms

This section discusses the small number of architectural modifications needed to enable a coherence protocol thread in a conventional out-of-order SMT architecture. Our baseline SMT design closely follows the design presented in [42]. We model nine pipeline stages: fetch, decode, rename, issue, two operand read stages, ALU/FPU, cache access, and commit. The fetch policy is ICOUNT with 8 instructions each from two threads being fetched every cycle, necessitating a dual-ported or double-pumped single-ported instruction cache. Instructions from the first thread are fed to decode until a predicted branch target (from the branch target buffer or return address stack) falls outside the 8 fetched instructions. At this point, instructions are taken from the second thread. All resources other than the return address stack (RAS), register map table, and parts of the branch predictor are dynamically shared. Every cycle the decode and rename stages process at most 8 instructions in fetch order.

During commit, the processor examines the top of the active lists (or reorder buffers) of all threads in a round robin fashion and continues retiring ready-to-retire instructions until the commit bandwidth is exceeded or there are no more retireable instructions from any thread. On the next cycle it starts from the thread with the highest round robin priority—thus the commit policy is round robin both within and across cycles. More details on the processor configuration can be found in Section 3.

2.1. Basic SMT Extensions

Figure 1 shows a complete node diagram of the proposed SMTp architecture. All parts other than the SDRAM and the network router are on the die. Figure 1 clearly designates the two halves of the design—the processor core and the memory controller. The shaded components are the necessary logic and storage additions for SMTp. Some changes are needed to avoid deadlock situations (described in Section 2.2) while others are needed for performance reasons (described in Section 2.3). First, we will discuss the basic extensions needed focusing on a single protocol thread. We leave out the design issues related to enabling multiple protocol threads for future research.

In the current design we statically bind the protocol thread to a hardware context that does not participate in context switching carried out by the kernel because we do not make this thread visible to the kernel. Alternative designs with dynamic protocol thread binding are possible, but require more complex deadlock analysis than the design we present here. Further, as in conventional DSM multiprocessors, in the proposed architecture the coherence protocol code is provided by the system not by the user, despite the fact that the protocol thread runs on the main processor.

An application L2 cache miss is queued in the Local Miss Interface of the memory controller. Similarly, an incoming network transaction queues a message in the Network Interface at the destination (note, this is also the I/O connection and hence is present in standard processors with

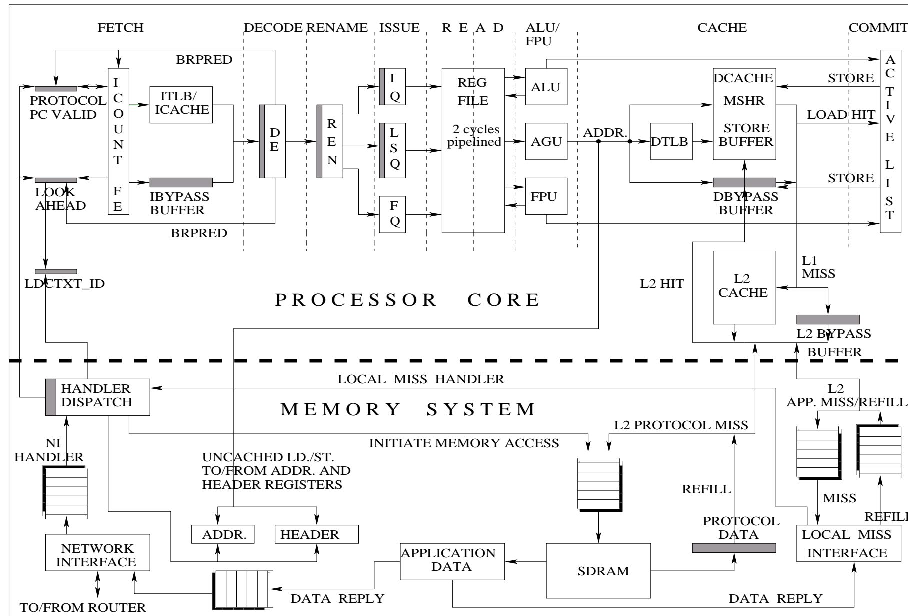

Figure 1: Node architecture of SMTp. All the components other than the SDRAM and the router are on the die. The shaded parts represent the logic and storage added on top of a conventional SMT.

integrated memory controllers). The handler dispatch unit selects a message from one of these two queues and extracts the address and the header of the message. It also initiates a memory access to the requested address if the message expects a cache line data reply—thus, the protocol thread (control) operates in parallel with the cache fill (data transfer). Next, the dispatch unit looks up a table to obtain the PC of the protocol thread handler for this transaction. To control fetching from the protocol thread, we introduce a one bit state named "Protocol PC Valid" or PPCV. The handler dispatch unit sets this bit along with the protocol thread PC when there is a request waiting to be serviced. If PPCV is set, the protocol thread participates in the ICOUNT decision policy and thus the processor will fetch protocol instructions. Note that the protocol thread shares the instruction cache with the application threads. However, since the protocol code and data are placed in an unmapped portion of the physical memory, the protocol thread never accesses (or pollutes) the ITLB or the DTLB. After the last handler instruction is fetched, the fetcher clears the PPCV bit and it becomes the responsibility of the handler dispatch unit or the mis-speculation recovery hardware to set this bit again as appropriate. The last two instructions in every handler are two special uncached load instructions. The first one (called `switch`) loads the header of the next request while the second one (called `ldctxt`) loads the address of the next request. To detect when the last instruction is being fetched we have quick compare logic that looks for `ldctxt` in the fetch bundle. If there is no request waiting to be serviced, `switch` stalls the head of the load/store queue section of the protocol thread (although the load/store queue buffers are dynamically shared across the threads, each thread gets separate logical head and tail pointers thereby forming separate logical load/store queues). In this case the memory controller unblocks the `switch` when the next request arrives. After the `ldctxt` executes it raises a `handlerCompletion` signal which in turn triggers the handler dispatch unit to set the PC of the next handler and the PPCV bit.

All the protocol load/store instructions go through the shared L1 data cache and, in case of a miss, to the L2 cache. However, a protocol instruction or data miss in the L2 cache behaves differently than an application miss. Protocol misses are not serviced recursively by another protocol thread; instead they bypass the Local Miss Interface and simply request the cache line from the local SDRAM. In the current design we provide a separate 64-bit bus to handle protocol misses. This decision is taken so that application and protocol data/instruction transfers can proceed in parallel. Alternatively, the same bus could be shared and the Local Miss Interface could be slightly modified to ensure that the protocol requests and refills bypass application misses.

Other than `switch` and `ldctxt`, there are a few more flavors of uncached load/store instructions that the protocol thread must support. One such example is a `send` instruction, which instructs the memory controller to initiate data transfer or control messages. These messages may send cache line-sized data or interventions either to the local L2 cache controller through the Local Miss Interface or to remote nodes via the Network Interface. The `send` instruction carries out two uncached stores that write to the header and the address registers in the memory controller (see Figure 1), and initiate the send. Therefore, minor modifications are needed in the load/store unit to properly identify these instructions. All these instructions must execute non-speculatively because undoing their effects is very hard or impossible (e.g. it is impossible to undo a `send`). As a result, the load/store unit needs to co-ordinate with the graduation unit before executing these instructions.

Finally, in this design we assume the existence of some special ALU instructions that can carry out bit manipulations common in protocol thread code sequences [14]. The

Alpha ISA already contains some of these instructions such as the "population count" and the "count trailing zeros" instructions. However, even without these instructions, we found that the SMTp execution time on 16 nodes increases by less than 0.3% on average and at most by 0.8% of what we report in Section 4.

2.2. Deadlock Avoidance

In a conventional SMT processor, if fairness is guaranteed, there are no cyclic dependences among the threads. However, in an SMTp architecture the graduation of an application load/store instruction that misses in the L2 cache depends on the forward progress of the protocol thread. Since the forward progress of the protocol thread depends on the availability of pipeline resources, this creates a cyclic dependence between the application threads and the protocol thread. The pipeline resources involved in this dependence are front-end queue buffers i.e. decode and rename queue buffers, branch stack space (used to checkpoint register maps for in-flight branches), integer registers, integer queue entries, load/store queue buffers, speculative store buffers and miss status holding registers (MSHRs). Since in this design we consider per-thread active lists (or reorder buffers), they are not involved in this kind of resource deadlock. We avoid any deadlock involving these resources by maintaining one reserved instance of each of these resources that is usable by the protocol thread only. Thus if the processor has eight decode queue buffers, only seven can be used by the application threads while all eight buffers are open to the protocol thread. Alternatively, following the hyper-threaded Pentium 4 design we could statically partition some of these resources to simplify deadlock avoidance.

Our solution requires some changes in the decode and rename queue scheduling algorithm because now one slot may remain empty and hence these queues are no longer purely circular FIFO. Although the queue slots remain dynamically shared among the threads, we maintain two separate head and tail pointers for application instructions and protocol instructions. Thus two separate logical queues are formed—one for application instructions and the other for protocol instructions. The decoder or the renamer visits each section of the respective queue every cycle, but the priority changes cyclically i.e. if in this cycle it visits the application section first and then the protocol section, the next cycle it visits the protocol section first and then the application section. Within each section of the queues the instructions are still processed in fetch order.

The reservation policy for integer registers requires some explanation because, in general, a single reserved integer register is not sufficient to guarantee forward progress of the protocol thread. The protocol boot sequence, which is a part of the machine boot code, initializes all 32 logical protocol registers. This guarantees that all the protocol registers remain mapped all the time. Therefore, one single reserved integer register is sufficient to avoid deadlock because the protocol instruction taking the reserved register is guaranteed to free the previous mapping when the instruction graduates. An alternative solution of keeping 33 reserved integer registers and not initializing unused registers actually hurts performance in some cases because most of the logical protocol registers get defined during the execution of an application. We will further comment on some optimizations related to integer registers in Section 2.3.

A different kind of deadlock situation can arise due to conflicts in the L1 and L2 caches. For example, a protocol load/store miss may conflict (i.e. map to the same cache line index) with an in-flight application load/store miss in the cache and the conflicting protocol load/store may belong to the handler that is servicing the application load/store miss. Clearly, we cannot delay the protocol load/store until the application load/store finishes because that will lead to a deadlock. So we provide small fully associative "bypass buffers" with the L1 instruction and data caches and the L2 cache (see Figure 1) to be used by the protocol thread only. Whenever a protocol thread load/store miss conflicts with an in-flight application load/store, the protocol thread, instead of allocating a cache line, allocates a cache line-sized bypass buffer line. While fetching from the protocol thread or executing protocol load/store instructions the cache and the corresponding bypass buffer are looked up in parallel. The L1 and L2 cache controllers are augmented with the necessary hardware to maintain inclusion after the bypass buffers are added. The bypass buffers are sized to capture the most pathological situation where all the protocol load/store misses are conflicting with in-flight application load/stores, and therefore this size is determined by the number of MSHRs.

2.3. Performance Issues

While evaluating the SMTp architecture we uncovered three performance bottlenecks. First, the short protocol thread handlers suffer greatly due to pipeline latency. We simulate nine pipe stages and there are critical handlers that are only six instructions long. Recall from the previous discussion that the starting PC of the next handler is handed out only after the last instruction from the previous handler graduates. We could hide the pipeline latency of the short handlers if we could hand out its starting PC to the fetcher as soon as possible. We call this optimization "Look Ahead Scheduling" (LAS) where we augment the dispatch hardware to hand out the PC of the next handler as soon as the PC is ready and the fetching of the previous handler is finished. We call this handler a look ahead handler. Currently, our design supports only one look ahead handler.

However, this requires special consideration of protocol branch misprediction recovery because if the look ahead handler gets completely squashed due to a mispredicted branch in the normal handler, machine state must reflect that correctly. Hence, when starting a look ahead handler we set a one bit state named `Look Ahead` and also store the active list sequence number of the last instruction (which is always `ldctxt`) from the last handler in `ldctxt_id` (see Figure 1). The `Look Ahead` state is cleared when this `ldctxt` graduates. While recovering from a mispredicted protocol branch, if the sequence number stored in `ldctxt_id` falls within the squash range and if `Look Ahead` is set, we know that the offending branch belongs to the normal handler and that we have squashed the look ahead handler completely. So at this point the `Look Ahead` state is cleared and as usual a re-fetch is initiated from the correct branch target in the normal handler. When the fetcher completes fetching the current handler it will check the

`Look Ahead` state and start fetching the look ahead handler again. LAS improves the performance by up to 3.9% for various applications and as a result we decided to include it in our design.

We found the number of integer registers to be a second bottleneck, but only for one application (FFTW; see Section 3). We looked at one optimization in this regard. Every protocol handler uses some scratch registers to carry out certain computations on the directory entry, the header, or the address. The values in these scratch registers are not needed after the handler completes. Therefore, we chose to free these scratch registers at the time the last instruction in a handler graduates. We implemented this optimization with a single extra read port to the protocol thread's register map table and one extra write port to the integer free list to be used by the graduation unit only (more than one port did not improve performance). This optimization improved performance of FFTW by at most 3.2%. However, when we implemented this in conjunction with LAS described above, we observed very little performance gain because with LAS two handlers may be executing in the pipe and though we can free the scratch registers used by the current handler, we cannot free those used by the look ahead handler. In addition, we need to keep track of the scratch registers of the look ahead handler since these cannot be freed. As a result, we decided not to include this optimization in our design.

Finally, since the L1 data cache and the L2 cache are shared between the application threads and the protocol thread, the number of conflict misses (mostly in the L1 data cache) increases in the SMTp architecture for the same cache sizes. However, the protocol thread provides the flexibility to explore smart directory address mappings to solve this problem, and this remains one area of ongoing research in the SMTp design. Mapping functions that avoid application cache lines from colliding with their own directory entries did not improve performance indicating that "self-collision" is not a problem. We are also exploring micro-architectural techniques such as dynamic detection and avoidance (by leveraging the bypass buffers) of conflicts between "hot" application lines and directory lines, and directory value prediction to cover the directory misses in the L1 data cache. To get an idea about how much performance is lost due to conflict misses, we simulated separate and perfect protocol instruction and data caches. In some cases the performance improved by 0.9-3.2% and in one case by 5.1%. We also found that sharing the instruction cache does not lead to any performance degradation for the applications we consider.

3. Applications and Simulation Methodology

This section discusses the applications and the simulation environment we use to evaluate our proposed architecture.

Table 1 lists the six explicitly parallel shared memory programs that we use in this paper. FFTW is a 3D Fast Fourier Transform kernel operating on complex double points [8]. The other five applications are chosen from the SPLASH-2 suite [44]. The programs represent a variety of important scientific computations with different communication patterns and synchronization requirements. As a simple optimization, in Ocean the global error lock in the multigrid phase has been changed from a lock–test–set–unlock sequence to a more efficient test–lock–test–set–unlock sequence [13]. The input sizes are chosen to capture realistic machine behavior for these highly scalable shared memory programs. All the applications use proper page placement to minimize remote memory accesses, and where possible all applications other than Water use hand-inserted prefetch and prefetch exclusive instructions to hide cache miss latency. The applications use software tree barriers to implement scalable synchronization. Finally, the transpose phases in FFT and FFTW are carefully optimized using padding and tiling. The relative performance trends for less-tuned applications that do not use prefetching and page placement are qualitatively identical to the results presented in this paper for optimized applications.

Applications	Problem Sizes
FFT	1M points, blocked for DTLB
FFTW	8192×16×16 points, 32×32 block
LU	512×512 matrix, 16×16 block
Radix-Sort	2M keys, radix=32
Ocean	514×514 grid, tolerance 1e-5
Water	1024 molecules, 3 time steps

Table 1: Applications and Problem Sizes

In this paper we present simulation results for DSM multiprocessors up to 32 nodes where each node contains an out-of-order SMT processor with 1, 2 or 4 application thread contexts along with an additional protocol thread context that we enable only in our SMTp results. Therefore, results on 16 nodes with 4-way SMT nodes present the performance of 64-threaded execution of the above programs. For 32 nodes we only consider SMT nodes that have 1 or 2 application thread contexts thereby limiting this study to at most 64-threaded parallel applications. Table 2 shows the MIPS ISA based simulated processor configuration along with SMTp-specific reserved or extra resources.

We present detailed results for 2 GHz processor frequency and explore the impact of scaling the frequency to 4 GHz. We simulate a branch predictor similar to the Alpha 21264 Tournament predictor [19]. However, we provide a separate local branch history table, global path history, and choice predictor for each thread. The local prediction and global prediction saturating counters (pattern history tables) are shared. This leads to a predictor size of roughly 86 Kbits for a 4-way SMT machine. We do not update the global path history speculatively. The RAS hardware is augmented with a mechanism to restore both the top of stack pointer and its contents on mis-speculation recovery as proposed in [37]. As in the MIPS R10000 [45] the branch stack is used to checkpoint the register maps when a branch passes through the rename stage. The size of the branch stack determines the maximum number of in-flight branches. The number of physical registers is decided as follows. For an n-way SMT machine with an additional protocol thread we provide $32(n+1)+96$ physical registers. Therefore, 96 extra registers are provided for renaming purposes. Note that we provide an extra 32 registers to the baseline models (discussed below) where the protocol thread is not active i.e. the baseline models have the same number of registers as the SMTp configuration, but have one less active thread context. The integer and floating-point multiply

Parameter	Value
Frequency	2 GHz/4 GHz
Thread contexts	1/2/4+protocol thread
Pipe stages	9
Fetch policy	ICOUNT (2 threads)
Front-end width	8
Decode queue slots	8
Rename queue slots	8
BTB	256 sets, 4-way
Branch predictor	Tournament (21264)
RAS	32 entries (per thread)
Active list	128 entries (per thread)
Branch stack	32 entries
Integer Register	160/192/256
FP Register	160/192/256
Integer queue	32 entries
FP queue	32 entries
Unified load/store queue	64 entries
ALU	7 (one for addr. calc.)
Integer mult./div. latency	6/35 cycles
FPU	3
FP mult. latency	fully pipelined, 1 cycle
FP div. latency	12 (SP)/19 (DP) cycles
Commit width	8
ITLB	128/fully assoc./LRU
DTLB	128/fully assoc./LRU
Page size	4 KB
L1 Icache	32 KB/64B/2-way/LRU
L1 Dcache	32 KB/32B/2-way/LRU
Unified L2 cache	2 MB/128B/8-way/LRU
MSHR	16+1 for retiring stores
Store buffer	32
L1 cache hit	1 cycle
L2 cache hit	9 cycles (round trip)
SMTp specific	
Res. decode queue slots	1
Res. rename queue slots	1
Res. branch stack slots	1
Res. integer registers	1
Res. integer queue slots	1
Res. LSQ slots	1
Res. MSHR	1
Res. store buffer	1
IBypass buffer	16 lines/fully assoc./LRU
DBypass buffer	16 lines/fully assoc./LRU
L2 Bypass buffer	16 lines/fully assoc./LRU

Table 2: Simulated Processor Configuration

and divide latencies are taken from the R10000 processor. As in the R10000, the load/store issue logic preserves program order among memory operations, and on a memory order violation eight instructions are unmapped every cycle from the tail of the active list and a re-fetch is initiated starting from the offending load. The simulated processor provides a strict sequentially consistent memory model.

Each node also contains a memory controller capable of handling local cache misses from the processor and remote cache misses from the network interface (NI). The memory controller is also responsible for executing the underlying directory-based cache coherence protocol. However, in SMTp this happens in one of the thread contexts on the main processor. In our baseline non-SMTp models (discussed below) the memory controller is equipped with a programmable dual-issue protocol engine as in the FLASH multiprocessor [22]. However, the protocol engine is closer in design to the hub of the SGI Origin 2000 [23] with the exception that it is programmable. Our protocol code compilation tools generate an optimized schedule for the dual-issue engine. To keep the comparison fair, while evaluating SMTp we modify this code only slightly to eliminate some unnecessary NOPs. In the non-SMTp models the instruction and data cache behavior, and the contention effects of the protocol processor are modeled precisely via a cycle-accurate simulator similar to that for the protocol processor in [10]. The invalidation-based coherence protocol running under a slightly relaxed model (employs eager-exclusive replies [6]) is derived from the bitvector protocol of the SGI Origin 2000. The simulated directory entry (per 128 bytes of application data) is 32 bits up to 16 nodes and 64 bits for 32 nodes, 16 or 32 bits of which are dedicated to the sharer vector, respectively. The other details of the memory system are listed in Table 3.

Parameter	Value
System bus width	64 bits
System bus speed	Same as mem. controller
SDRAM access time	80 ns
SDRAM bandwidth	3.2 GB/s
SDRAM queue	16 entries
Local miss queue	16 entries
NI input queues	2 entries each, 4 queues
NI output queues	16 entries each, 4 queues
Virtual networks	4 (protocol uses 3)
Router ports	6 (SGI Spider [9])
Network topology	2-way bristled hypercube
Hop time	25 ns
Link bandwidth	1 GB/s

Table 3: Memory System Configuration

In this paper the router is always assumed to be a separate module that is not integrated on the die. Depending on the position and design of the memory controller (MC) we explore five different machine models as shown in Table 4. Each entry in Table 4 shows the position of the memory controller, the frequency of the memory controller, and the size of the directory data cache. Each model presents a possible point in the design space. Each of the non-SMTp models (i.e. `Base`, `IntPerfect`, `Int512KB`, `Int64KB`) has a programmable dual-issue protocol processor (PP) embedded in the memory controller. The performance of `Base` is expected to be very close to a conventional hardware DSM machine without an integrated memory controller, such as the SGI Origin 2000. At 400 MHz frequency almost all protocol processing overhead is hidden under the SDRAM access. Also, a 512 KB directory data cache shows close to 100% hit rate.

`IntPerfect` represents the most aggressive hardware DSM machine with an integrated memory controller running at the same speed as the processor. Despite running at

Model name	Description
Base	Non-integrated PP/MC, 400 MHz, 512 KB direct mapped (DM) directory data cache
IntPerfect	Integrated PP/MC, processor frequency, perfect directory data cache
Int512KB	Integrated PP/MC, half processor frequency, 512 KB DM directory data cache
Int64KB	Integrated PP/MC, half processor frequency, 64 KB DM directory data cache
SMTp	Integrated standard MC (no PP), half processor frequency, protocol thread

Table 4: Machine Models (PP=Protocol Processor, MC=Memory Controller)

half the processor speed, Int512KB performs closely to IntPerfect given the high directory cache hit rate, but 512KB may be considered large for an on-chip directory cache in the integrated memory controller. Int64KB represents a more realistic design if one chooses to have a directory data cache with single-cycle access, as opposed to accessing a separate directory DRAM in parallel with the data DRAM as in the SGI Origin 2000. The latter would give an illusion of 100% directory hit rate at the cost of a complicated memory system design and larger directory lookup times. For all these four cases the protocol instruction cache is kept fixed at 32 KB direct mapped.

Finally, SMTp is the design we propose here. In this case the protocol code executes on a protocol thread context in the processor and hence at the processor speed. However, the standard integrated memory controller, which does not have any directory processing capability, still operates at half the processor speed. The protocol thread shares the L1 instruction and data caches, and the L2 cache with the application thread(s), and does not have separate protocol caches.

4. Simulation Results

In this section we present representative simulation results and a detailed performance analysis of SMTp. We first present results on single-node, 16-node, and 32-node DSM multiprocessors with 2 GHz SMT processors. In Section 4.1 we analyze the performance, overhead, and resource usage of the protocol thread. Section 4.2 presents selected results as the processor clock frequency scales to 4 GHz.

We begin our discussion with some general scalability results. Tables 5 and 6 present the speedup of all six applications on 16-node systems for Base and SMTp, respectively. In all the following results an n-way configuration corresponds to SMT nodes running n application threads—thus, the 4-way speedup corresponds to a 64-threaded execution. For SMTp there is an additional protocol thread. Speedups are calculated with respect to the single-node 1-way execution for each of Base and SMTp. These self-relative speedup numbers should not be used to compare the effectiveness of SMTp versus Base. We present these numbers to assert that at least for 1-way configurations applications are getting reasonable speedup. 2-way configurations improve performance further, while 4-way configurations hurt performance in most cases, due to cache conflicts and resource contention among the threads. We now examine relative performance across our machine models.

Figures 2, 3 and 4 present the relative performance of the five machine models on a single-node system with 2 GHz processors. The execution times are normalized to Base (the leftmost bar). The next four bars correspond to the integrated memory controller configurations. The last bar presents the results for SMTp, where the coherence protocol executes on a protocol thread context. Memory stall cycles are defined as the number of cycles, averaged over all the application threads, for which the graduation unit is stalled with a memory operation at the top of the active list. The remaining cycles contain the useful CPU cycles, squash cycles, synchronization cycles, and other stall cycles (from graduation viewpoint). So the non-memory stall cycles may vary across the machine models for the same application.

First, we observe that, as expected, memory controller

Application	1-way	2-way	4-way
FFT	13.87	19.32	19.35
FFTW	8.04	9.23	7.22
LU	14.61	14.98	13.15
Ocean	21.41	27.95	26.53
Radix-Sort	9.90	12.56	12.07
Water	12.58	18.60	21.25

Table 5: 16-node Speedup in Base

Application	1-way	2-way	4-way
FFT	14.04	19.11	19.54
FFTW	7.76	8.91	7.28
LU	14.15	14.45	12.72
Ocean	21.31	27.65	29.37
Radix-Sort	10.81	13.25	13.20
Water	13.25	19.96	22.37

Table 6: 16-node Speedup in SMTp

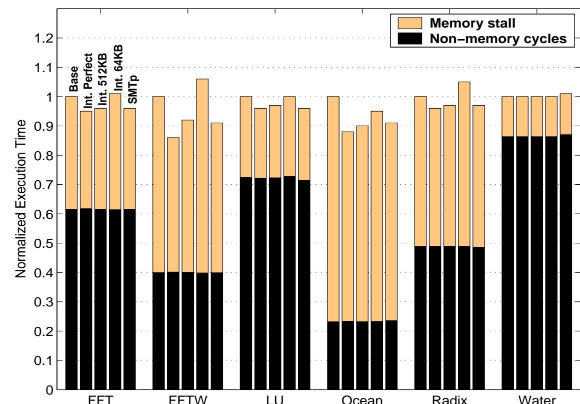

Figure 2: Performance on 1-way single-node configuration

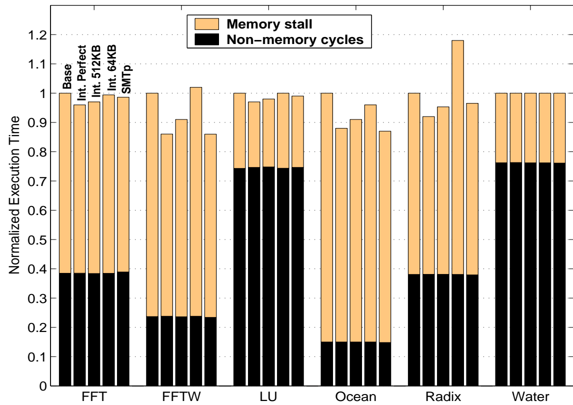

Figure 3: Performance on 2-way single-node configuration

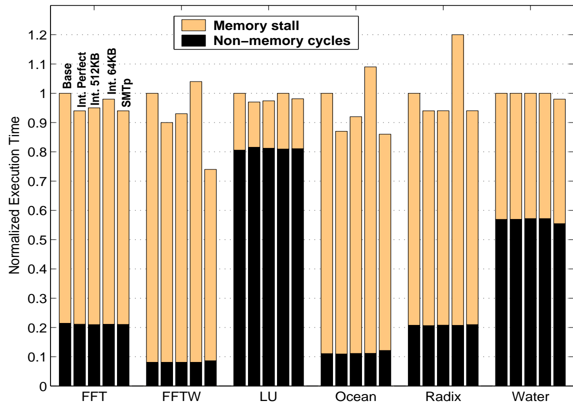

Figure 4: Performance on 4-way single-node configuration

integration improves performance. Ocean and FFTW see the maximum benefit, although LU and Water, the two computationally-intensive applications, remain largely insensitive to controller integration. Second, SMTp is always faster than `Base`, and in most cases performs on par with `Int512KB`. Only in a few cases does `Int512KB` outperform SMTp, and by at most 1.6%. `Int64KB` is the worst-performing integrated configuration, highlighting the importance of directory cache behavior for small machine sizes. Radix-Sort and FFTW are particularly sensitive to this issue. In fact, for a 4-way SMT machine `Base` is 20% faster than `Int64KB` for Radix-Sort. However, despite having a small shared 32 KB L1 data cache, SMTp is able to absorb the L1 directory misses in the 2 MB L2 cache. There are even a few cases where SMTp is faster than `IntPerfect`! The main reason for this is improved cache behavior due to changed timing of cache accesses leading to different LRU behavior.

Figures 5, 6 and 7 present results for 16-node systems. As the directory cache pressure decreases with machine size, the four machine models with integrated memory controllers tend to converge in terms of performance. However, `IntPerfect` still gets an edge over the other three due to the fact that its memory controller logic runs at

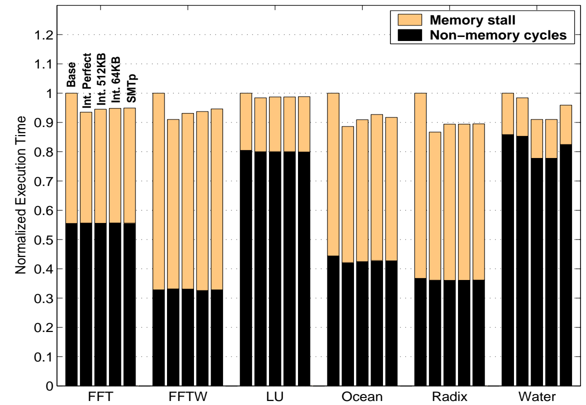

Figure 5: Performance on 1-way 16-node configuration

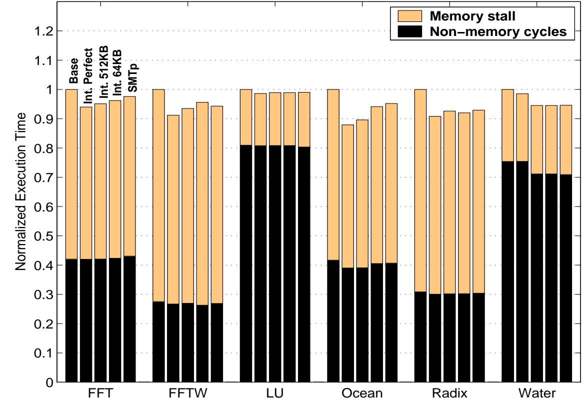

Figure 6: Performance on 2-way 16-node configuration

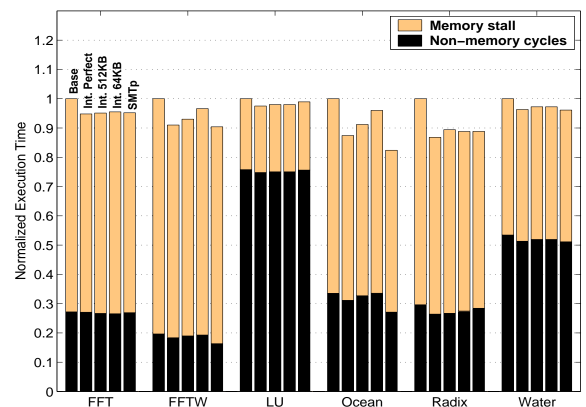

Figure 7: Performance on 4-way 16-node configuration

twice the speed of the others (see Table 4). SMTp continues to maintain its excellent performance, tracking that of `Int512KB` closely. Along with FFTW and Ocean, Radix-Sort also starts benefiting from memory controller integra-

tion. The main reason for the variation in the non-memory cycles of Water is the change in synchronization time across the machine models. Also, on 16 nodes, `Int64KB` delivers much better relative performance than on a single node. This is expected since on a 16-node system each node holds only a fraction of the whole directory space, and hence the smaller directory cache is less of a performance problem. As we have already mentioned in Section 2.3, we experimented with separate and perfect protocol instruction and data caches for the cases where `Int512KB` is more than 1% faster than SMTp. With perfect protocol caches, SMTp performed equally well or even better than `Int512KB` for most of those cases. In the remaining cases `Int512KB` outperformed SMTp by at most 1.1%. This shows that the main reason for any performance gap between SMTp and `Int512KB` is data cache pollution, particularly in the L1 data cache. We will further comment on the pipeline overhead of the protocol thread in Section 4.1.

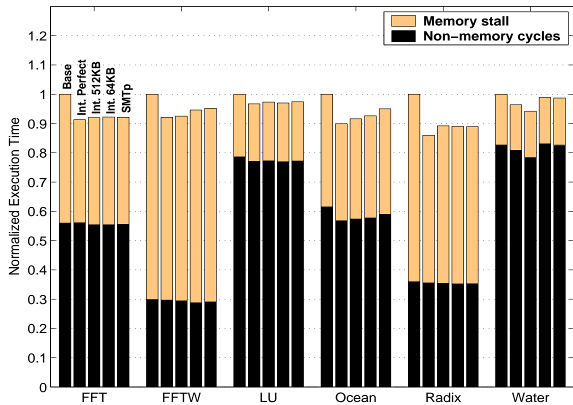

Figure 8: Performance on 1-way 32-node configuration

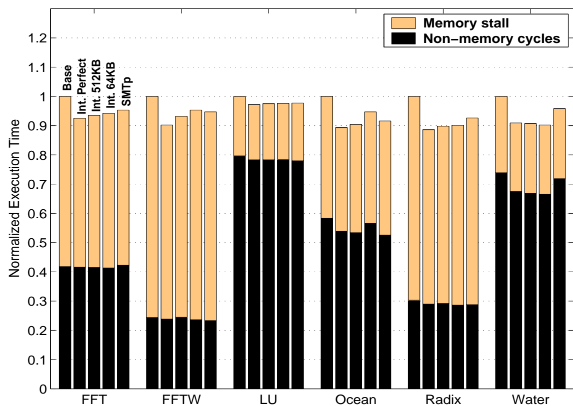

Figure 9: Performance on 2-way 32-node configuration

Figures 8 and 9 present results for 32-node systems. SMTp continues to deliver performance similar to `Int512KB` even for this medium-scale DSM.

4.1. Protocol Thread Characterization

This section discusses the pipeline resource overhead of the protocol thread and characterizes its performance. One of the benefits of clocking the memory controller faster is reduced execution time of the coherence protocol. This may not directly affect the observed latency of serving a message since the SDRAM access latency may dominate. However, this directly affects the queuing delay of a transaction and hence the overall pipeline *occupancy* of the protocol thread. Table 7 shows the peak protocol occupancy (among all the nodes) as a percentage of parallel execution time for `Base` and three integrated models on 16-node systems with 1-way SMT nodes. For non-SMTp models this is calculated as the percentage of parallel execution time the embedded protocol processor is active while for SMTp this is the percentage of time the protocol thread is active. As expected, the pro-

App.	`Base`	`IntPerf.`	`Int512KB`	SMTp
FFT	10.2%	3.6%	5.3%	5.8%
FFTW	14.6%	6.5%	8.4%	12.2%
LU	2.0%	0.4%	0.8%	1.2%
Ocean	25.0%	7.7%	12.3%	12.9%
Radix	12.1%	3.1%	5.6%	6.2%
Water	1.5%	0.3%	0.6%	0.7%

Table 7: 16-node Protocol Occupancy (1-way Nodes)

tocol occupancy of the integrated models is much smaller than the non-integrated model. `IntPerfect` enjoys the lowest protocol occupancy due to its faster memory controller logic. SMTp occupancy is similar to `Int512KB`, but suffers slightly from a larger cache miss rate for directory accesses. However, this small occupancy increase is rarely the cause of significant performance difference between SMTp and `Int512KB`. These results show that the protocol thread is active for a very small amount of time during the application execution, and hence is expected to have extremely small pipeline overhead. Interestingly, the occupancy percentages divide the six applications into two categories, namely, memory-intensive (FFT, FFTW, Ocean, Radix-Sort) and compute-intensive (LU, Water). In the previous results we have seen that SMTp delivers excellent performance for both these categories.

We have already mentioned that the protocol thread sometimes suffers from a slightly larger cache miss rate for directory accesses due to sharing of caches with the application threads. This also slightly increases the cache miss rate of application threads in some cases. Some other aspects of the protocol thread execution are presented in Tables 8 and 9. The protocol thread branch misprediction rate, the percentage of cycles the graduation unit frees at least one squashed protocol instruction, and the retired protocol instruction count as a percentage of all retired instructions are shown in Table 8. The results are again for 16-node systems with 1-way SMT nodes. Other than Water, all applications have at least 95% protocol branch prediction accuracy, indicating that speculative execution in general and Look Ahead Scheduling in particular can provide good protocol performance. Also, the low squash cycle percentage of the protocol thread supports this (Squash % column). Lack of train-

App.	Br.Mis. Rate	Squash %	Retired Ins.
FFT	2.11%	0.02%	4.18% of all
FFTW	1.57%	0.04%	5.46% of all
LU	4.03%	0.01%	0.34% of all
Ocean	2.38%	0.06%	8.36% of all
Radix-Sort	4.23%	0.06%	6.81% of all
Water	10.91%	0.01%	0.19% of all

Table 8: Protocol Thread Characteristics for 16 Nodes (1-way)

ing is the main reason for poor branch prediction in Water. This is also supported by the extremely low retired protocol instruction count (absolute count is not shown here). All the applications also show a small fraction of graduated protocol instructions (last column) as compared to the graduated application instructions, highlighting the extremely low overhead of the protocol thread mechanism.

Finally, Table 9 lists peak protocol thread occupancy (when it is active) for some of the key pipeline resources (branch stack, integer registers, integer queue and load/store queue) for 16-node systems with 1-way SMT nodes. We show the peak occupancy numbers (left) and the average of the peak numbers (right) across all the nodes. Referring back to Table 2 we see that peak occupancy of the protocol thread can be surprisingly high—50%-87% for the branch stack, 61%-71% for the integer registers, 100% for the integer queue, and 33%-55% for the load/store queue. However, we must note that this is the peak occupancy across the whole system over the duration of complete application execution. With the exception of the IQ occupancy, the average numbers also show that there are nodes with lower peak resource usage. Further, the protocol thread is active for a very small amount of time and when it is not active all the resources other than the mapped 32 registers and two load/store queue slots (for `switch` and `ldctxt`) are freed. As expected, we found that, resource occupancy decreases slightly if Look Ahead Scheduling is turned off because only one protocol handler can be active at a time in that case.

App.	Br. Stack	Int. Regs	IQ	LSQ
FFT	26, 18	109, 100	32, 32	24, 19
FFTW	22, 12	100, 91	32, 32	21, 17
LU	26, 23	99, 88	32, 32	26, 22
Ocean	28, 21	113, 100	32, 32	31, 24
Radix-Sort	16, 13	111, 97	32, 32	20, 18
Water	22, 20	98, 95	32, 32	35, 27

Table 9: Active Protocol Thread Occupancy for 16 Nodes (1-way)

4.2. Clock Rate Scaling Study

This section explores the performance trends as the processor clock frequency scales to 4 GHz. In `Base` the memory controller still runs at 400 MHz, while the memory controller speeds of the integrated models are doubled accordingly. Figure 10 presents 8-node 1-way results. For comparison in Figure 11 we also present the corresponding results with a 2 GHz processor clock. It is clear that the performance trends remain completely unchanged as the processor clock frequency scales. The main difference is that the relative performance gain of the integrated models increase as the processor-memory gap widens. We saw similar results for other machine sizes. We conclude that SMTp will continue to deliver excellent performance as the processor clock frequency scales in the future.

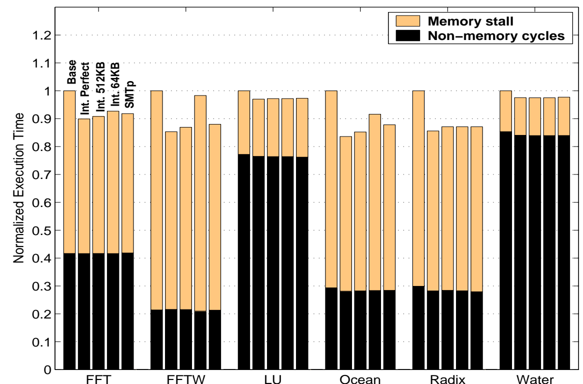

Figure 10: Performance on 1-way 8-node configuration with 4 GHz processor

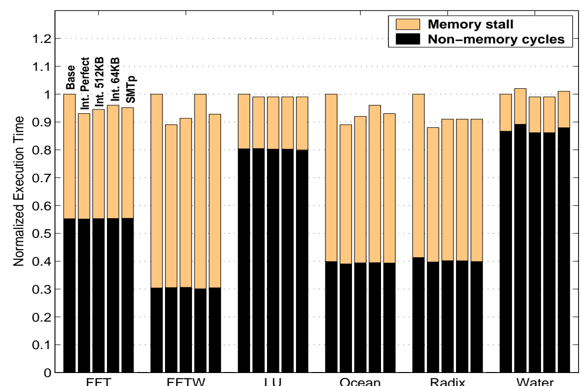

Figure 11: Performance on 1-way 8-node configuration with 2 GHz processor

5. Related Work

Several researchers have proposed using SMT to assist single-thread execution [4, 5, 36, 38, 46]. These proposals use otherwise unused thread contexts to either pre-execute delinquent loads, resolve hard-to-predict branches ahead of time, or use the values pre-computed by a leading thread. SMT architectures have also been used in the domain of redundant multi-threading as a form of fault tolerance [29, 38, 43]. These studies are largely orthogonal to the SMTp architecture that uses SMT threads to handle coherence decisions on processor cache misses.

Programmable coherence protocol engines have been studied and designed by several research groups. Some

of these are customized protocol processors e.g. the Piranha chip multiprocessor [3], Stanford FLASH multiprocessor [22], Sequent STiNG [26], and Sun S3.mp [30], while others use commodity off-the-shelf processors e.g. [28] and Typhoon [35]. However, we focus on combining the recent trends of integrated memory controllers and multi-threaded processors by using a system thread context in an SMT processor to provide the protocol programmability and by using a standard integrated memory controller. We propose micro-architectural extensions to enable our protocol thread mechanism and explain deadlock avoidance strategies particular to our design.

Grahn and Stenström implemented a cache coherence protocol entirely in software [12]. However the processors are not multi-threaded and running the coherence protocol requires interrupting application execution. The work examines strategies to reduce protocol overhead and assumes the availability of a fast low-level interrupt mechanism. In contrast, in SMTp the protocol thread does not interrupt the application threads and does not require a context switch. In fact, the resource overhead of the protocol thread is extremely low.

Parker et al. [31, 32] have proposed using SMT for user-level message passing. Since their work uses SMT threads to help support cluster computing, it is closest to the work proposed here. However, our focus is on the architectural changes needed to efficiently support the SMTp protocol thread mechanism, and on combining that with a standard integrated memory controller to create hardware cache-coherent DSM machines.

6. Conclusions

This paper presents the first design that exploits simultaneous multi-threading in conjunction with a standard integrated memory controller to enable a low-overhead coherence protocol thread running on the main processor that can create (among other possibilities) a DSM multiprocessor. We present a practical implementation of this SMTp architecture (*SMT* with a *p*rotocol thread) and explain in detail the minor microarchitectural changes necessary in a conventional out-of-order SMT pipeline to realize this design. The architectural modifications come in the form of a small number of reserved pipeline resources to break deadlock cycles inherent between the protocol thread and the application threads executing in the same pipe. We also add a small amount of state to implement a performance-enhancing optimization, namely, look ahead scheduling of protocol handlers.

A thorough evaluation up to 32 nodes and 64 threads compares the performance of SMTp against a baseline DSM design with a non-integrated memory controller as well as an array of DSM designs with integrated memory controllers. The results show that up to 32 nodes our design is always better than the baseline non-integrated design and is always within 6% and mostly within 3% of (and sometimes outperforms) an aggressive integrated design with a stand-alone 512 KB directory cache. Further, we find that on average (arithmetic mean) SMTp delivers performance equivalent to `Int512KB`. We also present results for the most aggressive integrated design to estimate an upper bound on achievable performance. We further examine the overhead and resource usage characteristics of protocol threads, and our study on clock frequency scaling shows that SMTp maintains its outstanding performance as the processor-memory gap widens.

The extensive evaluation also brings out one remaining bottleneck in our design, namely, the data cache pollution resulting from the protocol thread sharing the cache with application threads. However, the performance degradation is small compared to the added complexity of having a separate cache hierarchy for the protocol thread. In summary, the SMTp architecture not only simplifies the design of the memory controller needed to support scalable directory-based cache coherence, but also delivers performance comparable to conventional DSM multiprocessors with integrated coherence controllers.

Finally, we note that the SMTp mechanism need not be restricted to enabling coherent DSM multiprocessors with standard computing nodes, though that is our focus here. We are currently examining other potential SMTp applications including on-the-fly compression/encryption, active memory address re-mapping, fault tolerance techniques, and selective and accurate value prediction that can benefit from SMTp's ability to dispatch a particular protocol sequence on an L2 cache miss. As a design alternative we are also looking at using unused core(s) in a chip multiprocessor (CMP) for enabling coherence protocol thread(s) in a similar fashion as SMTp.

Acknowledgments

This research was supported by NSF CAREER Award CCR-0340600.

References

[1] G. Abandah and E. Davidson. Effects of Architectural and Technological Advances on the HP/Convex Exemplar's Memory and Communication Performance. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 318–329, June 1998.

[2] A. Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, June 1995.

[3] L. Barroso et al. Piranha: A Scalable Architecture Based on Single-chip Multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 282–293, June 2000.

[4] R. S. Chappell et al. Difficult-Path Branch Prediction Using Subordinate Microthreads. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 307–317, May 2002.

[5] J. D. Collins et al. Dynamic Speculative Precomputation. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 306–317, December 2001.

[6] D. E. Culler, J. P. Singh with A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[7] Z. Cvetanovic. Performance Analysis of the Alpha 21364-Based HP GS1280 Multiprocessor. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 218–228, June 2003.

[8] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the 23rd International Conference on Acoustics, Speech, and Signal Processing*, pages 1381–1384, May 1998.

[9] M. Galles. Spider: A High-Speed Network Interconnect. In *IEEE Micro*, **17**(1):34–39, January-February 1997.

[10] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, November 2000.

[11] P. Glaskowsky. IBM Raises Curtain on Power5. In *Microprocessor Watch*, Issue#113, October 27, 2003.

[12] H. Grahn and P. Stenström. Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 38–47, June 1995.

[13] M. Heinrich and M. Chaudhuri. Ocean Warning: Avoid Drowning. In *ACM SIGARCH Computer Architecture News*, **31**(3):30–32, June 2003.

[14] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[15] M. Heinrich, E. Speight, and M. Chaudhuri. Active Memory Clusters: Efficient Multiprocessing on Commodity Clusters. In *Proceedings of the Fourth International Symposium on High-Performance Computing, Lecture Notes in Computer Science*, Vol. 2327, pages 78–92, Springer-Verlag, May 2002.

[16] InfiniBand Architecture Specification, Volume 1.0, Release 1.0. InfiniBand Trade Association, October 24, 2000.

[17] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous Multithreading Implementation in POWER5–IBM's Next Generation POWER Microprocessor. In *Hot Chips 15*, August 2003.

[18] C. N. Keltcher et al. The AMD Opteron Processor for Multiprocessor Servers. In *IEEE Micro* **23**(2):66–76, March-April 2003.

[19] R. E. Kessler. The Alpha 21264 Microprocessor. In *IEEE Micro*, **19**(2):24–36, March-April 1999.

[20] D. Kim et al. Architectural Support for Uniprocessor and Multiprocessor Active Memory Systems. In *IEEE Transactions on Computers*, **53**(2):288–307, February 2004.

[21] D. Koufaty and D. T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. In *IEEE Micro*, **23**(2):56–65, March-April 2003.

[22] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[23] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[24] D. Lenoski et al. The Stanford DASH Multiprocessor. In *IEEE Computer*, **25**(3):63–79, March 1992.

[25] T. D. Lovett, R. M. Clapp, and R. J. Safranek. NUMA-Q: An SCI-based Enterprise Server. Sequent Computer Systems Inc., 1996.

[26] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 308–317, May 1996.

[27] D. T. Marr et al. Hyper-Threading Technology Architecture and Microarchitecture. In *Intel Technology Journal*, Vol. 6, Issue 1, pages 4–15, February 2002.

[28] M. Michael at al. Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 219–228, June 1997.

[29] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 99–110, May 2002.

[30] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 24th International Conference on Parallel Processing*, Vol. 1, pages 1–10, August 1995.

[31] M. Parker, A. Davis, and W. Hsieh. Message-Passing for the 21st Century: Integrating User-Level Networks with SMT. In *Proceedings of the 5th Workshop on Multithreaded Execution, Architecture and Compilation*, December 2001.

[32] M. Parker. A Case for User-Level Interrupts. In *HPCA Work-In-Progress*, February 2002.

[33] PCI Express Advanced Switching. Intel Press Release. Available at http://www.intel.com/pressroom/archive/releases/20030626net.htm.

[34] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 111–122, May 2002.

[35] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 34–43, May 1996.

[36] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th International Conference on High Performance Computer Architecture*, pages 191–202, January 2001.

[37] K. Skadron et al. Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms. In *Proceedings of the 31st ACM/IEEE International Symposium on Microarchitecture*, pages 259–271, December 1998.

[38] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, November 2000.

[39] Sun Microsystems. An Overview of UltraSPARC III Cu. White Paper, September 2003. Available at http://www.sun.com/processors/whitepapers/USIIICuoverview.pdf.

[40] Sun Microsystems. UltraSPARC IV Processor Architecture Overview. White Paper, February 2004. Available at http://www.sun.com/processors/whitepapers/us4_whitepaper.pdf.

[41] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[42] D. M. Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[43] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery Using Simultaneous Multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 87–98, May 2002.

[44] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[45] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. In *IEEE Micro*, **16**(2):28–40, April 1996.

[46] C. B. Zilles and G. S. Sohi. Execution-based Prediction Using Speculative Slices. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 2–13, July 2001.