SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches *

Seth H. Pugsley, Josef B. Spjut,
David W. Nellans, Rajeev Balasubramonian

School of Computing, University of Utah

{pugsley, sjosef, dnellans, rajeev} @cs.utah.edu

ABSTRACT

Snooping and directory-based coherence protocols have become the de facto standard in chip multi-processors, but neither design is without drawbacks. Snooping protocols are not scalable, while directory protocols incur directory storage overhead, frequent indirections, and are more prone to design bugs. In this paper, we propose a novel coherence protocol that greatly reduces the number of coherence operations and falls back on a simple broadcast-based snooping protocol when infrequent coherence is required. This new protocol is based on the premise that most blocks are either private to a core or read-only, and hence, do not require coherence. This will be especially true for future large-scale multi-core machines that will be used to execute message-passing workloads in the HPC domain, or multiple virtual machines for servers. In such systems, it is expected that a very small fraction of blocks will be both shared and frequently written, hence the need to optimize coherence protocols for a new common case. In our new protocol, dubbed SWEL (protocol states are Shared, Written, Exclusivity Level), the L1 cache attempts to store only private or read-only blocks, while shared and written blocks must reside at the shared L2 level. These determinations are made at runtime without software assistance. While accesses to blocks banished from the L1 become more expensive, SWEL can improve throughput because directory indirection is removed for many common write-sharing patterns. Compared to a MESI based directory implementation, we see up to 15% increased performance, a maximum degradation of 2%, and an average performance increase of 2.5% using SWEL and its derivatives. Other advantages of this strategy are reduced protocol complexity (achieved by reducing transient states) and significantly less storage overhead than traditional directory protocols.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles–Cache Memories

General Terms

Design, Performance, Experimentation

Keywords

Cache Coherence, Shared Memory

*This work was supported in parts by NSF grants CCF-0430063, CCF-0811249, CCF-0916436, NSF CAREER award CCF-0545959, SRC grant 1847.001, Intel, HP, and the University of Utah.

PACT'10, September 11–15, 2010, Vienna, Austria.

1. INTRODUCTION

It is expected that multi-core processors will continue to support cache coherence in the future. Cache coherence protocols have been well-studied in the multi-socket multiprocessor era [9] and several snooping-based and directory-based protocols have emerged as clear favorites. Many of these existing solutions are being directly employed in modern multi-core machines. However, we believe that there are several differences between the traditional workloads that execute on modern multiprocessors, and workloads that will be designed for future many-core machines.

Expensive multi-socket systems with hardware cache coherence were designed with specific shared-memory applications in mind, *i.e.*, they were not intended as general-purpose desktop machines that execute a myriad of single and multi-threaded applications. Many of the applications running on today's multi-core machines are still single-threaded applications that do not explicitly rely on cache coherence. Further, future many-cores in servers and datacenters will likely execute multiple VMs (each possibly executing a multi-programmed workload), with no data sharing between VMs, again obviating the need for cache coherence. We are not claiming that there will be zero data sharing and zero multi-threaded applications on future multi-cores; we are simply claiming that the percentage of cycles attributed to shared memory multi-threaded execution (that truly needs cache coherence) will be much lower in the many-core era than it ever was with traditional hardware cache coherent multiprocessors. If the new common case workload in many-core machines does not need cache coherence, a large investment (in terms of die area and design effort) in the cache coherence protocol cannot be justified.

Continuing the above line of reasoning, we also note that many-core processors will also be used in the high performance computing (HPC) domain, where highly optimized legacy MPI applications are the common case. Data sharing in these programs is done by passing messages and not directly through shared memory. However, on multicore platforms these messages are passed through shared memory buffers, and their use shows a strong producer-consumer sharing pattern. State-of-the-art directory-based cache coherence protocols, which are currently employed in large-scale multi-cores, are highly inefficient when handling producer-consumer sharing. This is because of the indirection introduced by the direc-

tory: the producer requires three serialized messages to complete its operation and the consumer also requires three serialized messages. Thus, directory protocols are likely to be highly inefficient for both the on-chip and off-chip sharing patterns that are becoming common in large-scale multi-cores.

Given the dramatic shift in workloads, there is a need to reconsider the design of coherence protocols; new coherence protocols must be designed to optimize for a new common case. We must first optimize for the producer-consumer sharing pattern. Secondly, if most blocks are only going to be touched by a single core, the storage overhead of traditional directories that track large sharing vectors is over-provisioned and should be eliminated. Finally, we need simpler protocols that can lower design and verification efforts when scaling out. In this paper, we propose a novel hardware cache coherence protocol that tries to achieve the above goals.

Our protocol (named SWEL after the protocol states) is based on the premise that a large fraction of blocks are either private to a core or are shared by multiple cores in read-only mode. Such blocks do not require any cache coherence. Blocks must be both shared and written for coherence operations to be necessary. A key example of such blocks are those used in producer-consumer relationships. We recognize that blocks of this type are best placed in the nearest shared cache in the memory hierarchy, eliminating the need for constant, expensive use of coherence invalidations and updates between local private caches.

By eliminating the traditional coherence invalidate/update pattern, we can avoid implementing a costly sharer-tracking coherence mechanism. Instead, we propose using a simpler mechanism that can categorize blocks in one of only two categories (private or read-only vs. shared and written). Traditional directory overheads are now replaced with the book-keeping state required to achieve this categorization. This new protocol therefore has lower storage overhead and fewer transient states. The protocol does have some disadvantages, borne out of the fact that some blocks are relegated to a slower, shared cache level (L2 in this paper), and are therefore more expensive to access. Our results show that on average a SWEL based protocol can outperform a traditional MESI directory-based protocol by 2.5% on multi-threaded workloads from PARSEC, SPLASH-2, and NAS. This improvement is accompanied by lower storage overhead and design effort.

In Section 2 of this paper, we discuss the background of coherence protocols, their purposes, and the motivation for the features SWEL offers. Section 3 outlines the SWEL protocol, and we discuss its operations, drawbacks and how some of those drawbacks can be improved upon. In Section 4 we discuss the theoretical differences between the performance of MESI and SWEL, and the circumstances under which each should have optimal and worst performance. Sections 5 and 6 deal with our simulation methodology and results. In Section 7 we discuss related work and in Section 8 we wrap up our discussion of the SWEL protocol.

2. BACKGROUND & MOTIVATION

2.1 Data Sharing in Multi-threaded Workloads

All cache coherence protocols operate on the basic assumption that all data may be shared at any time, and measures need to be taken at every step to ensure that correctness is enforced when this sharing occurs. Traditional coherence systems over-provision for the event that all data may be shared by every processor at the same time, which is an extremely unlikely scenario. While private data never will require coherence operations, shared data may or may not require coherence support. Shared data can be broken down into two classes: read-only and read-write. Shared, read-only blocks are not complicated to handle efficiently, as simple replication of the data is sufficient. Shared data that is also written to, however, must be handled with care to guarantee that correctness and consistency is maintained between cores.

Figure 1 shows the sharing profile of several 16 threaded applications from the NAS Parallel Benchmarks [4], Parsec [5], and Splash2 [22] suite by (a) location, and (b) references. Breaking down sharing by locations and references provides two different views of how sharing occurs. Figure 1a indicates that very little data is actually shared by two or more cores; on average 77.0% of all memory locations are touched by only a single processor. Figure 1b however, shows that in terms of memory references, private data locations are accessed very infrequently (only 12.9% of all accesses). This implies that the vast majority of accesses in workload execution actually reference a very small fraction of the total memory locations. While the majority of accesses are to locations which are shared, very few of those locations (7.5% on average) are both shared and written, the fundamental property on which we base this work. Because shared/written data is a fundamental synchronization overhead that limits application scalability, we expect future workloads to try and minimize these accesses even further.

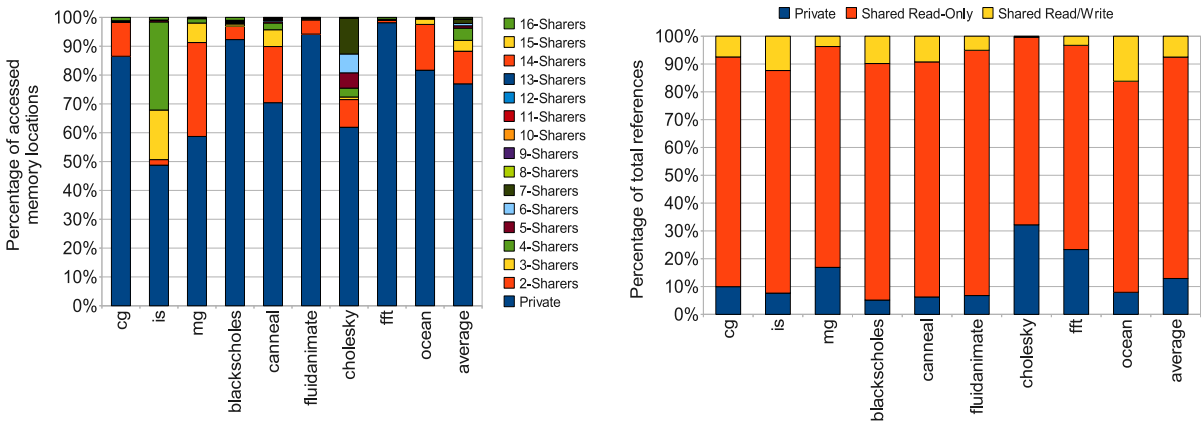
2.2 Coherence Protocols

For this study, we assume an on-chip cache hierarchy where each core has private L1 instruction and data caches and a shared L2 cache. The L2 cache is physically distributed on-chip such that each processor "tile" includes a bank of the L2 cache. We assume an S-NUCA [16] style L2 cache, where the address is enough to identify the bank that contains the relevant data. Our focus is the implementation of hardware coherence among the many private L1s and the shared L2, though our proposal could easily be extended to handle a multi-level hierarchy.

Coherence is typically implemented with snooping or directory-based protocols [9]. Bus-based snooping protocols are generally simpler to design, but are not scalable because of the shared bus. Directory-based protocols will likely have to be employed for many-core architectures of the future. As a baseline throughout this study, we will employ a MESI directory-based and invalidation-based coherence protocol. The salient features of this protocol are as follows:

- *Directory storage:* Each cache block in L2 must maintain a directory to keep track of how the block is being shared by the L1s. In an unoptimized design, each L2 cache block maintains a bit per L1 cache in the directory. Each bit denotes if the corresponding L1 has a copy of this cache block. The directory includes additional bits per cache block to denote the state of the cache block. Thus, the directory grows linearly with the number of cores (or private L1s). This storage overhead can be reduced by maintaining a bit per group of cores [9, 17]. If a bit is set in the directory, it means that one of the cores in that group of cores has a copy of the block. Therefore, when invalidating a cache block, the message must be broadcast to all the cores in a group marked as a sharer. This trades off some directory storage overhead for a greater number of on-chip network messages. It must also be noted that each L1 cache block requires two bits to track one of the four MESI coherence states.

- *Indirection:* All L1 misses must contact the directory in L2 before being serviced. When performing a write, the directory is often contacted and the directory must send out invalidations to other sharers. The write can proceed only after acknowledgments are received from all sharers. Thus, mul-



a. Sharing profile by memory locations

b. Sharing profile by memory references

Figure 1: Motivational Data: Memory sharing profile of 16 core/thread workloads

tiple messages must be exchanged on the network before the coherence operation is deemed complete. Similarly, when an L1 requests a block that has been written by another L1, the directory is first contacted and the request is forwarded to the L1 that has the only valid copy. Thus, many common coherence operations rely on the directory to serve as a liaison between L1 caches. Unlike a snooping-based protocol, the involved L1 caches cannot always directly communicate with each other. This indirection introduced by the directory can slow down common communication patterns. A primary example is the producer-consumer sharing pattern where one core writes into a cache block and another core attempts to read the same cache block. As described above, each operation requires three serialized messages on the network.

- *Complexity:* Directory-based coherence protocols are often error-prone and entire research communities are tackling their efficient design with formal verification. Since it is typically assumed that the network provides no ordering guarantees, a number of corner cases can emerge when handling a coherence operation. This is further complicated by the existence of transient coherence states in the directory.

In this work, we attempt to alleviate the above three negative attributes of directory-based coherence protocols.

3. SWEL PROTOCOL AND OPTIMIZATIONS

We first describe the basic workings of the SWEL protocol from a high level. The protocol design is intended to overcome three major deficiencies in a baseline directory-based protocol: the directory storage overhead, the need for indirection, and the protocol complexity. The basic premise is this: (i) many blocks do not need coherence and can be freely placed in L1 caches; (ii) blocks that would need coherence if placed in L1 are only placed in L2. Given this premise, it appears that the coherence protocol is all but eliminated. This is only partially true as other book-keeping is now required to identify which of the above two categories a block falls into.

If a cache block is either private or is read-only, then that block can be safely cached in the L1 without ever needing to worry about coherence. If the block is both shared (not private) and written (not read-only), then it must never exist in the L1 and must exist at the lowest common level of the cache hierarchy, where all cores have equal access to it without fear of ever requiring additional coherence operations. If a block is mis-categorized as read-only or as private, then it must be invalidated and evicted from all L1 caches and must reside permanently in the lowest common level of cache.

Consider from a high level how a block undergoes various transitions over its lifetime. When a block is initially read, it is brought into both L1 and L2, because at this early stage the block appears to be a private block. Some minor book-keeping is required in the L2 to keep track of the fact that only one core has ever read this block and that there have been no writes to it. If other cores read this block or if the block is ever written to, then the book-keeping state is updated. When the state for a block is "shared + written," the block is marked as "un-cacheable" in L1 and an invalidation is broadcast to all private caches. All subsequent accesses to this block are serviced by the lowest common level of cache, which in our experiments and descriptions is L2.

3.1 SWEL States and Transitions

3.1.1 States

We now explain the details of the protocol and the new states introduced. Every L2 cache block has 3 bits of state associated with it, and every L1 cache block has 1 bit of state. The first state bit in L2, *Shared (S)*, keeps track of whether the block has been touched by multiple cores. The second state bit, *Written (W)*, keeps track of whether the block has been written to. The third state bit, *Exclusivity Level (EL)*, which is also the one state bit in the L1, denotes which cache has exclusive access to this block. The exclusivity level bit may only be set in one cache in the entire system, be it the L2 or one of the L1s. We therefore also often refer to it as the *EL Token*. The storage requirement for SWEL (3 bits per L2 block and 1 bit per L1 block) does not depend on the number of sharers or L1 caches (unlike the MESI directory protocol); it is based only on the number of cache blocks. These 4 bits are used to represent 5 distinct states in the collapsed state diagram shown in Figure 2(a). We next walk through various events in detail. For now, we will assume that the L1 cache is write-through and the L1-L2 hierarchy is inclusive.

3.1.2 Initial Access

When a data block is first read by a CPU, the block is brought into the L2 and the corresponding L1, matching the Private Read state in the diagram. The EL state bit in the L1 is set to denote

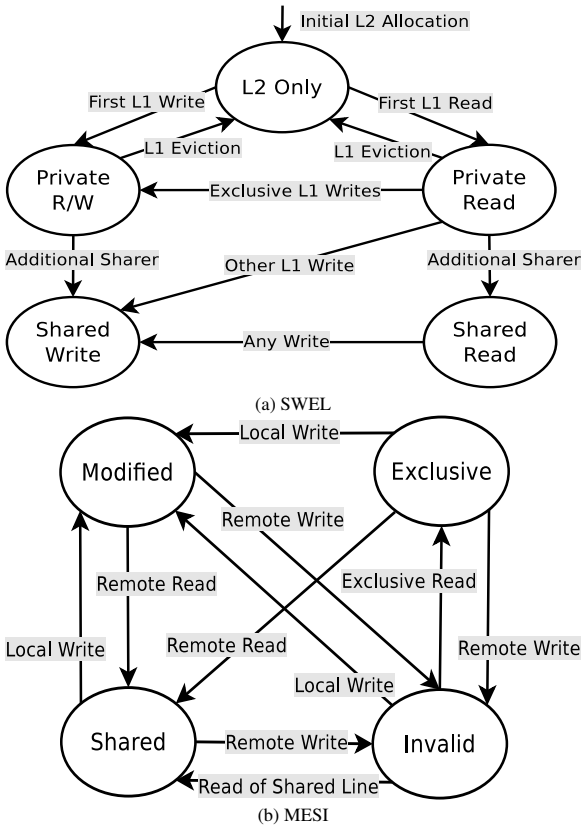


Figure 2: Collapsed State Machine Diagram for the SWEL and MESI Protocols

that the block is exclusive to the L1. The block in L2 is set as non-shared, non-written, and not exclusive to the L2. If this block is evicted from the L1 while in this state, it sends a message back to the L2 giving up its EL bit, matching the L2 Only state in the diagram. The state in the L2 is now non-shared, non-written and exclusive to the L2. If read again by a CPU, the block will re-enter the Private Read state.

3.1.3 Writes

When a block is first written by a CPU, assuming it was either in the L2 Only state or the Private Read state, it will enter the Private R/W state. If this initial write resulted in an L2 miss, then the block will enter this state directly. A message is sent as part of the write-through policy to the L2 so that it may update its state to set the W bit. The W bit in the L2 is "sticky", and will not change until the block is evicted from the L2. This message also contains a bit that is set if this writing CPU has the EL token for that block. This is so the L2 knows to not transition to a shared state. From the Private R/W state, an eviction of this block will send a message to the L2 giving back its EL bit and it will go back to the L2 Only state. Private data spends all of its time in these three states: L2 Only, Private Read and Private R/W.

3.1.4 Determining the Shared State

If a cache reads or writes a block and neither the cache nor the L2 have the corresponding EL token, then the L2 must set its S bit and enter either the Shared Read or Shared Written state. Once a block has its S bit set in the L2, that bit can never be changed unless the block is evicted from the L2. Since this is a sticky state, the EL token ceases to hold much meaning so it is unimportant for the EL token to be reclaimed from the L1 that holds it at this time. Additional shared readers beyond the first do not have any additional impact on the L2's state.

3.1.5 L1 Invalidation and L2 Relegation

Shared Read blocks are still able to be cached in the L1s, but Shared R/W blocks must never be. When a block first enters the Shared R/W state, all of the L1s must invalidate their copies of that data, and the EL bit must be sent back to the L2. The invalidation is done via a broadcast bus discussed later. Since this is a relatively uncommon event, we do not expect the bus to saturate even for high core counts. Future accesses of that data result in an L1 miss and are serviced by the L2 only. Data that is uncacheable at a higher level has no need for cache coherence. Once a block enters the Shared R/W state, there is no way for it to ever leave that state. The Shared R/W state can be reached from 3 different states. First, a Shared Read block can be written to, causing the data to be shared, read and now written. Second, a Private R/W block can be accessed by another CPU (either read or written), causing the data to be read, written and now shared. Finally, a Private Read block can be written to by another CPU, causing the block to be read and now written and shared.

3.1.6 Absence of Transient States

Transient states are usually introduced in conventional directory protocols when a request arrives at the directory and the directory must contact a third party to resolve the request. Such scenarios never happen in the SWEL protocol as most transactions typically only involve one L1 and the L2. The L2 is typically contacted to simply set either the Shared or Written bit and these operations can happen atomically. The only potentially tricky scenario is when a block is about to be categorized as "shared + written". This transition happens via an invalidation on a shared broadcast bus, an operation that is atomic. The block is immediately relegated to L2 without the need to maintain a transient state. Therefore, the bus is strategically used to handle this uncommon but tricky case, and we derive the corresponding benefit of a snooping-based protocol here (snooping-based protocols do not typically have transient states).

3.1.7 Requirements for Sequential Consistency

A sequentially consistent (SC) implementation requires that each core preserve program order and that each write happens atomically in some program-interleaved order [9]. In a traditional invalidation-based directory MESI protocol, the atomic-write condition can be simplified to state, "a cached value cannot be propagated to other readers until the previous write to that block has invalidated all other cached copies" [9]. This condition is trivially met by the SWEL protocol – when the above situation arises, the block is relegated to L2 after a broadcast that invalidates all other cached copies.

As a performance optimization, processors can issue reads speculatively, but then have to re-issue the read at the time of instruction commit to verify that the result is the same as an SC implementation. Such an optimization would apply to the SWEL protocol just as it would to the baseline MESI protocol.

Now consider the speculative issue of writes before prior operations have finished. Note that writes are usually not on the critical path, so their issue can be delayed until the time of commit. Since a slow write operation can hold up the instruction commit process, they are typically placed away in a write buffer. This can give rise to a situation where a thread has multiple outstanding writes in a write buffer. This must be handled carefully so as to not violate

SC. In fact, a write can issue only if the previous write has been made "visible" to its directory. This is best explained with an example.

Consider a baseline MESI protocol and thread $T1$ that first writes to A and then to B. At the same time, thread $T2$ first reads B and then A. If the read to B returns the new value, as written by $T1$, an SC implementation requires that the read to A should also return its new value. If $T1$'s write request for A is stuck in network traffic and hasn't reached the directory, but the write to B has completed, there is a possibility that thread $T2$ may move on with the new value of B, but a stale value of A. Hence, $T1$ is not allowed to issue its write to B until it has at least received an Ack from the directory for its attempt to write to A. This Ack typically contains a speculative copy of the cache block and the number of sharers that will be invalidated and that will be sending additional Acks to the writer.

In exactly the same vein, in the SWEL protocol, we must be sure that previous writes are "visible" before starting a new write. If the write must be propagated to the L2 copy of that block, the L2 must send back an Ack so the core can proceed with its next write. This is a new message that must be added to the protocol to help preserve SC. The Ack is required only when dealing with shared+written blocks in L2 or when issuing the first write to a private block (more on this shortly). The Ack should have a minimal impact on performance if the write buffers are large enough.

3.2 Optimizations to SWEL

The basic SWEL protocol is all that is necessary to maintain coherence correctly, although there are some low-overhead optimizations that improve its performance and power profile considerably.

3.2.1 Write Back

SWEL requires that all L1 writes be written through to the L2 so that the L2 state can be updated as soon as a write happens, just in case the cache block enters the Shared R/W state and requires broadcast invalidation. Writing through into the NUCA cache (and receiving an Ack) can significantly increase on-chip communication demands. For this reason, we add one more bit to the L1 cache indicating whether that block has been written by that CPU yet, which will be used to reduce the amount of write-through that happens in the cache hierarchy. This optimization keeps the storage requirement of the SWEL protocol the same in L2 (3 bits per cache block) and increases it to 2 bits per cache block in each L1.

When a CPU first does a write to a cache block, it will not have the write back bit set for that block and must send a message to the L2 so that the L2's W bit can be set for that block. When this message reaches the L2, one of two things will happen depending on its current state. Either the write was to a shared block, which would cause a broadcast invalidate, or the write was to a private block, with the writer holding the original EL token. In the former case the operation is identical to the originally described SWEL protocol. After this initial write message, all subsequent writes to that cache block in the L1 can be treated as write-back. The write-back happens when this block is evicted from the L1 for capacity issues, as in the normal case of write-back L1 caches, or in the event of a broadcast invalidate initiated by the SWEL protocol.

3.2.2 Reconstituted SWEL - RSWEL

The basic SWEL protocol evicts and effectively banishes Shared R/W data from the L1s in order to reduce the number of coherence operations required to maintain coherence. This causes the L1s to have a 0% hit rate when accessessing those cache blocks, and forces all requests to that data to go to the larger, slower shared L2 from the time the blocks are banished. It is easy to imagine this causing very low performance in programs where data is shared and written early on in execution, but from then on is only read. We propose improving SWEL's performance in this case with the Reconstituted SWEL optimization (RSWEL). This allows banished data to be cacheable in the L1s again after it has been banished, effectively allowing the re-characterization of Shared R/W data to be private or read-only data again, and improving the latency to access those blocks. After a cache block is reconstituted, it may be broadcast invalidated again, and in turn it may be reconstituted again. The RSWEL optimization aims to allow for optimum placement of data in the cache hierarchy at any given time.

Since the goal of the SWEL protocol is to reduce coherence operations by forcing data to reside at a centralized location, it is not necessarily the goal of the RSWEL optimization to have data constantly bouncing between L1s and L2 like a directory protocol would have it do. Instead the RSWEL optimization only reconstitutes data after a period of time, and only if it is unlikely to cause more coherence operations in the near future. For this reason we add a 2-bit saturating counter to the L2 state which is initialized with a value of 2 when a cache block is first broadcast invalidated, and which is incremented each time the block is written while in its banished state. The counter is decremented with some period N, and when it reaches 0, the cache block can be reconstituted on its next read. The RSWEL protocol with a period N=0 will behave similarly to a directory protocol, where data can be returned to the L1s soon after it is broadcast invalidated, and a period N=infinity will behave identically to the regular SWEL protocol.

3.3 Dynamically Tuned RSWEL

The RSWEL optimization assumes a fixed reconstitution period for the entire program execution. This does not take into account transitions in program phases, which may prefer one reconstitution period or another. To solve this problem, we also introduce a Dynamically Tuned version of RSWEL, which seeks to find the locally optimal reconstitution period N for each program phase.

This works by analyzing the performance of the current epoch of execution, and comparing that with the previous epoch's analysis. If sufficient change is noticed, then we consider a program phase change to have occurred, and we then explore several different values of N to see which yields the locally highest performance. After this exploration is completed, the N with the highest performance is used until another program phase change is detected.

The details of our implementation of this scheme into the framework of the RSWEL protocol are as follows. To detect program phase changes, we use the metric of average memory latency, and when this varies by 5%, we consider a program phase change to have taken place. During exploration, we try to maximize L1 hit rates, as this metric most closely correlates with performance and can be measured on a local scale. Our epoch length is 10 kilocycles, and we use the reconstitution period timers of N = 10, 50, 100, 500 and 1,000 cycles.

Average memory operation latency is an appropriate metric to detect program phase changes because it stays relatively the same for a given sharing pattern, but then changes sharply when a new sharing pattern is introduced, which is indicative of a program phase change. We kept the N-cycle timer in the range of 10-1,000 because the highest performing timer values for our benchmark suite are frequently within this range, and rarely outside it, as can be seen later in Figure 6a.

3.4 SWEL/RSWEL Communication Implementation

Up to this point we have talked about on-chip communication from a very high-level perspective, but now we will describe the methods of communication in the SWEL design. SWEL combines a point-to-point packet-switched network with a global broadcast bus. The point-to-point network is used for handling cache misses of remote data, write back, and write through messages. The global broadcast bus is used exclusively for sending out cache block invalidation messages.

The point-to-point network serves fewer functions in SWEL than it does in a MESI protocol. In MESI, the network serves all of the same functions as in SWEL, but it also handles indirection to find the latest copy of data, and point-to-point invalidation. If the amount of traffic generated by MESI's indirection and invalidation is more than the amount of traffic generated by SWEL's write address messages and compulsory L1 misses of Shared R/W data, then some energy could be saved in the network by SWEL. However, we expect this to likely not be the case as the MESI coherence messages are short, but the compulsory L2 accesses in SWEL require larger data transfers.

The global broadcast bus serves a single function in SWEL, and we believe this bus will scale better than buses are generally believed to scale because of its low utilization rate. Buses perform poorly when they are highly utilized. However, the bus in SWEL is used only for broadcast invalidates, and these occur infrequently, only when a block's classification changes. When a MESI protocol repeatedly performs point-to-point invalidates of the same cache block, SWEL performs only one broadcast invalidate of that cache block ever. The RSWEL protocol might perform multiple broadcast invalidates of the same block, but for the right reconstitute period N it will do it far less often than a MESI protocol would perform point-to-point invalidates.

4. SYNTHETIC BENCHMARK COMPARISON

4.1 Best Case Performance - MESI and SWEL

Since coherence operations are only required to bring the system back into a coherent state after it leaves one, none are required if the system never leaves a coherent state in the first place. When running highly parallel programs with little or no sharing, or running different programs concurrently, very few or no coherence operations will be required. In the case of MESI, no sharer bits will be set, so when writes occur, no invalidation messages will be sent. In the case of SWEL, the shared bit in the L2 state is never set, so when writes occur, there are never any broadcast invalidates. As a result, in the case where all data is processor private, SWEL and MESI will perform identically. As stated earlier, we believe these types of programs to be the norm in future large-scale parallel systems. For these benchmark classes, SWEL is able to achieve the same level of performance as MESI with a much lower storage overhead and design/verification effort.

4.2 Worst Case Performance - MESI

MESI is at its worst in producer-consumer type scenarios. In these programs, one processor writes to a cache block, invalidating all other copies of that block in the system. Later, another processor reads that block, requiring indirection to get the most up to date copy of that block from where it was modified by the first processor. Repeated producing and consuming of the same block repeats this process.

SWEL handles this situation much more elegantly. When one processor writes a block and another reads it, the block permanently enters the shared + written state, so the data is only cacheable in the L2. From then on, all reads and writes to this block have the same latency, which is on average lower than the cost of MESI's indirection and invalidation. Our test of a simple two thread producer consumer program showed SWEL to perform 33% better than MESI in this ideal case.

4.3 Worst Case Performance - SWEL

SWEL is at its worst when data is shared and written early in program execution, causing it to only be cacheable in the L2, and then rarely (if ever) written again but repeatedly read. SWEL is forced to have a 0% L1 hitrate on this block, incurring the cost of an L2 access each time which typically exceeds L1 latency by 3-5x. This can happen because of program structure or because of thread migration to an alternate processor in the system due to load balancing. If a thread migrates in a SWEL system, then all of its private written data will be mis-characterized as Shared R/W.

In this case, MESI handles itself much more efficiently. After the block goes through its initial point to point invalidate early in program execution, it is free to be cached at the L1 level again benefiting from the low L1 access latency. Our test of a simple two thread program specifically showing off this behavior showed MESI performs 62% better than SWEL. It is important to keep in mind that the RSWEL optimization allowing reconstitution of shared+written blocks back into L1 completely overcomes this weakness. Thus, we point out this worst case primarily to put our results for the baseline SWEL protocol into context.

5. METHODOLOGY

5.1 Hardware Implementation

In order to test the impact of our new coherence protocol, we modeled 3 separate protocol implementations using the Virtutech Simics [18] full system simulator version 3.0.x. We model a Sunfire 6500 machine architecture modified to support a 16-way CMP processor implementation of in-order ultraSPARC III cores with system parameters as shown in Table 1. For all systems modeled, our processing cores are modeled as in-order with 32 KB 2-way L1 instruction and data caches with a 3 cycle latency. All cores share a 16 MB SNUCA L2 with 1 MB local banks having a bank latency of 10 cycles. The L2 implements a simple first touch L2 page coloring scheme [8, 15] to achieve locality within this L2 and to reduce routing traffic and average latency to remote bank accesses. Off-chip memory accesses have a static 300 cycle latency to approximate best-case multi-core memory timings as seen by Brown and Tullsen [7].

Our on-chip interconnect network is modeled as a Manhattan routed interconnect utilizing 2 cycle wire delay between routers and 3 cycle delays per pipelined router. All memory and wire delay timings were obtained from the recent CACTI 6.5 update to CACTI 6.0 [20], with a target frequency of 3 GHz at 32 nm, assuming a 1 cm^2 chip size. Energy numbers for the various communication components can be found in Table 1.

5.2 Benchmarks

For all benchmarks, we chose to use a working set input size that allowed us to complete the entire simulation for the parallel region of interest. By simulating all benchmarks and implementations for a constant amount of work, we can use throughput as our measure of performance. The fewer cycles required to complete this defined region of interest, the higher the performance. This eliminates the

Core and Communication Parameters

ISA	UltraSPARC III ISA		
CMP size and Core Freq.	16-core, 3.0 GHz	Network	Dimension-order Routed Grid
L1 I-cache	32KB/4-way, private, 3-cycle	Router Latency	3-cycle
L1 D-cache	32KB/4-way, private, 3-cycle	Link Latency	2-cycle
L2 Cache Organization	16x 1MB banks/8-way, shared	Bus Arbitration Latency	12-cycle
L2 Latency	10-cycle + network	Bus Transmission Latency	14-cycle
L1 and L2 Cache block size	64 Bytes	Flit Size	64 bits
DRAM latency	300 cycles		

Energy Characteristics

Router Energy	$1.39\text{x}10^{-10} J$
Link Energy (64 wide)	$1.57\text{x}10^{-11} J$
Bus Arbitration Energy	$9.85\text{x}10^{-13} J$
Bus Wire Energy (64 wide)	$1.25\text{x}10^{-10} J$

Table 1: Simulator parameters.

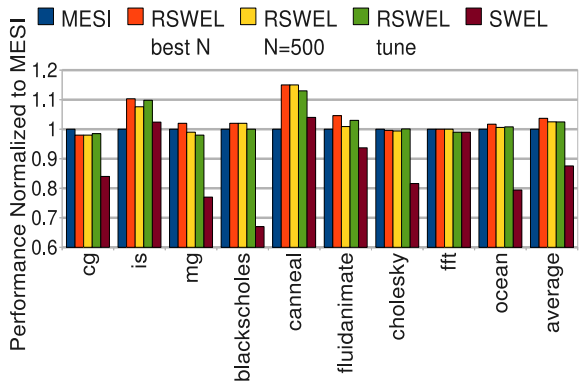


Figure 3: Benchmark performance

effects of differential spinning [3] that can artificially boost IPC as a performance metric for multi-threaded simulations. Our simulation length ranged from 250 million cycles to over 10 billion cycles per core, with an average of 2.5 billion cycles per core.

6. EXPERIMENTAL RESULTS

The synthetic workloads shown in Section 4 show two extremes of how parallel programs might behave. Real workloads should always fall within these two bounds in terms of behavior and therefore performance. This section will focus on results from benchmarks intended to be representative of real parallel workloads. We are focusing our evaluation on a workload that will stress the coherence sub-system and identify the scenarios where SWEL and its variants may lead to unusual performance losses. Note again that SWEL is intended to be a low-overhead low-effort coherence protocol for future common-case workloads (outlined in Section 1) that will require little coherence. For such workloads (for example, a multi-programmed workload), there will be almost no performance difference between SWEL and MESI, and SWEL will be a clear winner overall. Our hope is that SWEL will have acceptable performance for benchmarks that frequently invoke the coherence protocol, and even lead to performance gains in a few cases (for example, producer-consumer sharing). The benchmarks we use come from the PARSEC [5], Splash2 [22] and NAS Parallel [4] benchmark suites.

In determining the value of the SWEL and RSWEL protocols, we look at five main metrics: performance, L1 miss rate per instruction, L1 eviction rate per instruction, network load, and energy requirement.

6.1 Application Throughput

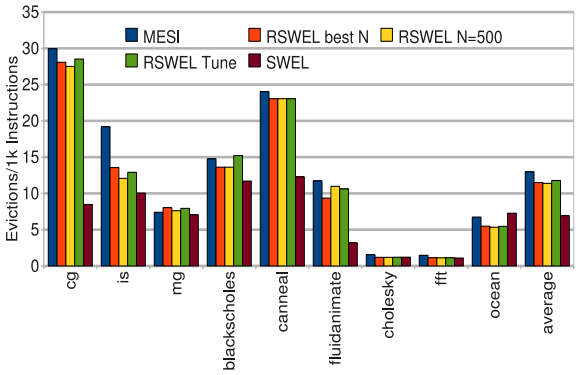
Performance is measured by the total number of cycles for a program to complete rather than instructions per cycle, because in our SIMICS simulation environment, it is possible for the IPC to change drastically due to excess waiting on locks, depending on the behavior of a particular benchmark. As can be seen in Figure 3, RSWEL is consistently competitive with MESI, sometimes surpassing its performance, and is never too far behind MESI. SWEL also compares favorably with MESI in some benchmarks, but its worst case is much worse than RSWEL's. We include N = 500 as a datapoint because we found this to be the best all-around static value of N.

SWEL is able to outperform MESI in both the Canneal and IS benchmarks. Canneal employs fine-grained data sharing with high degrees of sharing and data exchange. IS is a simple bucket sort program, which exhibits producer-consumer sharing. In all cases, RSWEL outperforms SWEL, frequently by a large amount. The Dynamically Tuned RSWEL algorithm (referred to as RSWEL Tune in the figures) is consistently a strong performer, but does not match the best N. At its worst, RSWEL Tune is 2% worse than MESI, and at its best it is 13% better performing, with an average of 2.5%. The RSWEL protocols perform especially favorably in the Canneal, IS, and Fluidanimate benchmarks.

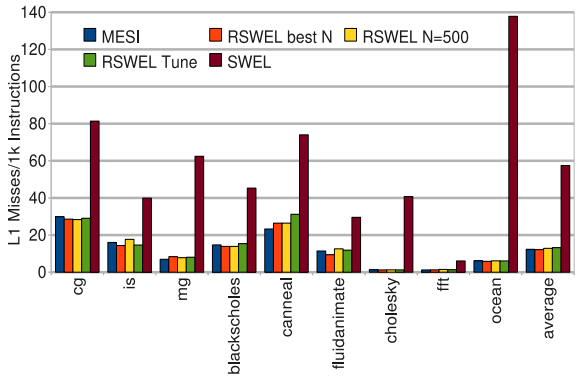
6.2 L1 Evictions

Effective L1 cache capacity is increased by SWEL and RSWEL, as is evidenced by the lower number of L1 evictions required by those protocols, as shown in Figure 4a. Every time a new block is allocated into the L1, it must evict an existing block. Under MESI, when a block gets invalidated due to coherence, this block is automatically selected for eviction rather than evicting a live block. Under MESI, if that invalidated block is accessed a second time, it will evict yet another block in the L1. These Shared R/W blocks have a negative effect on the effective capacity of the L1 caches due to thrashing. On the other hand, the longer the Shared R/W block stays out of the L1 the longer that capacity can be used for other live blocks.

SWEL does this well by keeping Shared R/W data out of the L1s indefinitely. In all benchmarks, SWEL has the fewest L1 cache evictions, meaning that more of the L1 cache is available to private data for a larger percentage of program execution, and data is not constantly being thrashed between L1 and L2 caches. The IS benchmark shows this behavior well. RSWEL and SWEL have noticeably fewer L1 evictions than MESI in this case. IS has a high demand for coherence operations and this results in a high rate of cache block thrashing between L1 and L2. If RSWEL uses the right reconstitution period it has a good opportunity to increase the effective cache size.

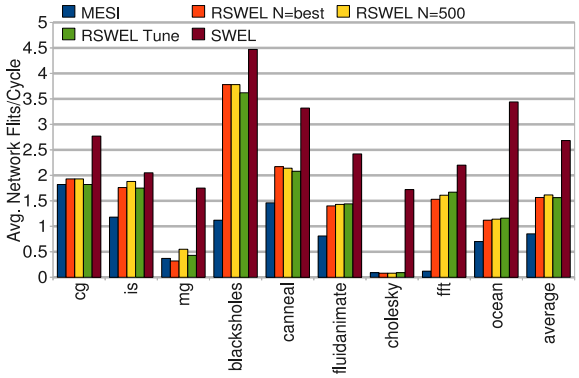


a. L1 evictions per kilo-instruction

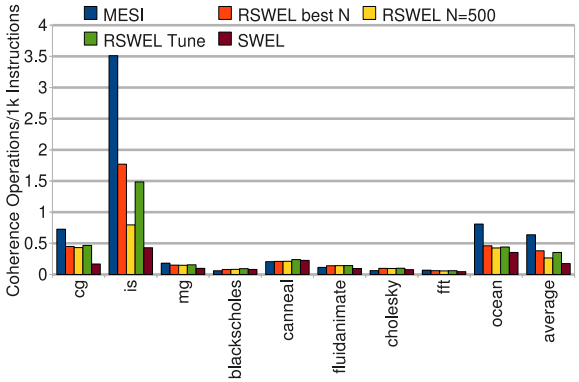


b. L1 misses per kilo-instruction

Figure 4: L1 cache performance comparison of coherence protocols



a. Average flits in the grid network at any given time



b. Operations needed to maintain coherence

Figure 5: Network utilization comparison of coherence protocols

6.3 L1 Miss Rate

The closer data can be found to where computation is performed, the higher the performance. This is why performance is tied so closely to L1 hit rates. Figure 4b shows the normalized number of L1 misses per instruction of each of the different tested protocols. It is immediately apparent that for some benchmarks, SWEL incurs many more L1 misses than the other protocols, causing its performance to be frequently weak compared to the other protocols. The RSWEL protocols show miss rates comparable to MESI.

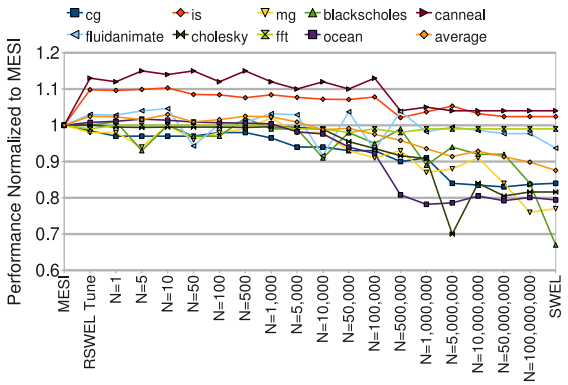
6.4 Communication Demands

- *Grid Network:* Network load represents the number of flits that pass through the routers of the grid interconnect network. An address message is one flit long and only contributes one to the network load for each router it passes through. For example, if an address message makes two hops, then its network impact is 3–one for each hop and one for the destination router which it must pass through. A data message, which is comprised of 9 flits (1 for address and 8 for data), generates a network load of 9 for each router it passes through. For example, a data message which travels one hop generates a network load of 18–nine for the router it touches before hopping to the destination, and another nine for the destination router.

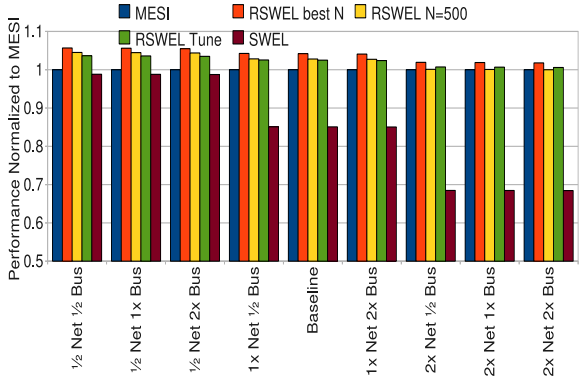
  Since SWEL and RSWEL write-through to the L2 on the first write of a cache block in L1, they will have higher network demands than MESI. MESI is able to perform write back on all of its writes, but SWEL protocols must perform write-through unless they have the write-back token. Although the write-back optimization can reduce the number of write-throughs by 50-98%, there are still many extra write-through messages in the system. Also, since SWEL's L1 hit rate is lower on average than MESI, the network is required more often to get data from the NUCA L2. In many cases, the RSWEL optimization can greatly reduce the amount of network traffic required by SWEL, but it still requires more network traffic than MESI on average (Figure 5a).

- *Broadcast Bus:*Broadcast buses are viable options when they are under-utilized. Bus utilization is the ratio of the amount of the time the bus is being charged to communicate, to the total execution time. In our experiments, the greatest bus utilization rate we observed was 5.6%, far below the accepted rate of 50% when buses start to show very poor performance. This occurred during the run of the IS benchmark, which had the greatest demand for coherence operations by far, as seen in Figure 5b.

- *Coherence Operations:*We define a coherence operation as an action that must be taken to bring an incoherent system back into coherence. In the case of directory-based MESI, there are two primary coherence operations. Point to point invalidates are required when shared data is written, causing the sharers to become out of date. L1 to L1 indirect transfers are required when one CPU holds a block in modified state and another CPU attempts a read. SWEL has only one coherence operation, in contrast. The use of the broadcast bus when a block enters the Shared R/W state atomically brings an incoherent system back into coherence. One broadcast



a. RSWEL performance when varying the reconstitution time N



b. Performance sensitivity of SWEL and RSWEL to varying network and bus latencies

Figure 6: Performance sensitivity to parameter variation

bus invalidate in SWEL can replace several point to point invalidates and L1 to L1 indirect transfers in directory-based MESI.

Figure 5b shows that for the majority of the sharing patterns in these benchmarks, relatively little effort must be expended to maintain coherence, much less than one coherence operation per thousand instructions. In the cases where greater effort was required, SWEL and RSWEL significantly reduce the number of coherence operations required.

6.5 Reconstitution Period Variation

The amount of time a block remains in the L2 will affect the overall performance execution of a program. While we provide just a few samples throughout most of our performance graphs, it is critical to see the variance that can occur by choosing a sub-optimal N. Figure 6a shows the performance sensitivity of our workloads to various values of N. An optimal N will minimize the number of coherence operations that occur when a program enters the write phase for a cache block, but allows the block to be re-constituted to the L1 level quickly when that write-phase ends. Blackscholes shows the greatest sensitivity to the choice of the reconstitution period. Choosing the wrong N can hurt performance from 10-30%. The RSWEL Tune protocol is effective at keeping performance far clear of the worst case for every benchmark, although it doesn't ever produce ideal performance.

6.6 Communication Latency Variation

SWEL and RSWEL can inject a higher number of flits into the communication network than MESI for some workloads. As a result, network performance may be more critical to overall application throughput for SWEL than with MESI based protocols. To test this hypothesis we ran experiments, shown in Figure 6b, that vary both the absolute and relative performance of our network and broadcast bus delays. The X-axis of the graph lists the relative latency of the communication mechanisms, compared to the baseline latency parameters found in Table 1. For example ½ Net - ½ Bus indicates that both the bus and network latency parameters are half that of the baseline. 2x Net - ½ Bus indicates that we have made the bus half the latency, but the interconnect network twice as slow. The Y-axis lists the normalized average performance of all workloads. For each latency set, SWEL and RSWEL are normalized to MESI performance using those same latencies, not the baseline latencies.

The results in Figure 6b indicate that neither SWEL nor RSWEL are sensitive to changes in bus latency. This is not surprising given the extremely low bus utilization by all variations of SWEL. RSWEL also appears to not be overly sensitive to network latency; this is a function of the low N values we evaluate for optimal performance. SWEL, however, is extremely sensitive to network latency because of the higher number of flits it injects into the network compared to both MESI and RSWEL as shown in Figure 5a.

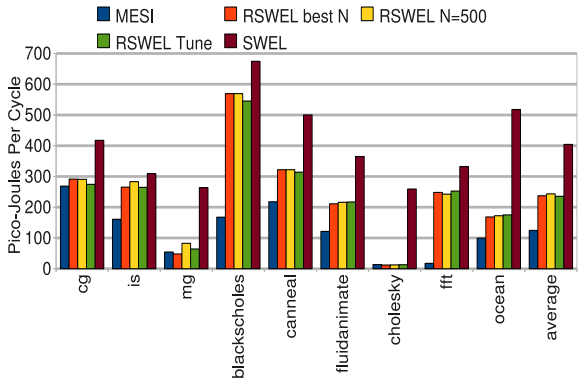


Figure 7: 16-core CPU Power Consumption, Including Network and Bus

6.7 Communication Power Comparison

Power consumption is an increasingly important metric as more processing cores are fit into a CPU die and on-chip networks grow in complexity. Figure 7 shows the power consumption of the on-chip communication systems of the various protocols. SWEL's power requirement is greatly increased due to its increased L1 miss rate; more address and especially data messages are sent across the grid network in the SWEL scheme. SWEL, at its worst, is contributing 2 W of power to the overall chip at 3 GHz and a 32 nm process. This contribution will be even lower for the workloads described in Section 1 that may have little global communication.

The RSWEL schemes perform more favorably compared to MESI, but still have higher L1 miss rates and write address messages that MESI doesn't have. The broadcast bus did not contribute very significantly to the power overhead of SWEL and RSWEL. The bus is used infrequently enough, and its per-use energy requirement low enough, that the grid network energy requirement greatly outweighs it. As can be seen in Table 1, one use of a 64-wide 5x5 router uses more energy than charging all 64 wires of the (low-swing) broadcast bus.

7. RELATED WORK

Much prior work has been done analyzing and developing cache coherence protocols. In an effort to reduce the directory storage overhead, Zebchuk et al. [24] suggest a way to improve the die area requirement of directory based coherence protocols by removing the tags in the directory and using bloom filters to represent the contents of each cache. In this scheme, writes can be problematic because they require that a new copy of the cache block be sent from a provider to the requesting writer, and because of false positives caused by the bloom filter, this provider might not still have a valid copy of the data in its cache. This happens infrequently in practice, but when it does it can be solved by invalidating all copies of the data and starting fresh with a new copy from main memory. Stenström [21] shows a way to reduce the directory state used to track sharers in directory based protocols. The SWEL protocol, in contrast, changes the directory scaling to be linear with the number of cache blocks only instead of scaling with both the number of cache blocks and the number of sharers.

Brown et al. [6] present a method to accelerate coherence for an architecture with multiple private L2 caches with a point-to-point interconnect between cores. Directory coherence is used with modifications to improve the proximity of the node from which data is fetched, thereby alleviating some of the issues of the directory indirection overhead. Acacio et al. [1, 2] explore a directory protocol that uses prediction to attempt to find the current holder of the block needed for both read and write misses. Eisley et al. [10] place a directory in each node of the network to improve routing and lower latency for coherence operations. An extra stage is added to the routing computations to direct the head flit to the location of the most up-to-date data. Hardavellas et al. [12] vary cache block replication and placement within the last-level cache to minimize the need for cache coherence and to increase effective cache capacity. They too rely on a dynamic classification of pages as either private or shared and as either data or instruction pages.

Some approaches exist to reduce the complexity of designing coherence hardware by simply performing coherence through software techniques. Yan et al. [23] do away entirely with hardware cache coherence and instead require programmers or software profilers to distinguish between data that is shared and written, and data that is read only or private. These two classes of data are stored in separate cache hierarchies, with all Shared R/W data accesses incurring an expensive network traversal. Fensch and Cintra [11] similarly argue that hardware cache coherence is not needed and that the OS can efficiently manage the caches and keep them coherent. The L1s are kept coherent by only allowing one L1 to have a copy of any given page of memory at a time. Replication is possible, but is especially expensive in this system.

Attempting to improve performance, Huh et al. [13, 14] propose separating traditional cache coherence protocols into two parts: one to allow speculative computations on the processor and a second to enforce coherence and verify the correctness of the speculation. However, anywhere from 10-60% of these speculative executions are incorrect, making it frequently necessary to repeat the computation once the memory is brought into a coherent state. Martin et al. [19] also aim to separate the correctness and performance parts of the coherence protocol but do so by relying on token passing. There are N tokens for each cache block, and a write requires the acquisition of all N, while only one is needed to be a shared reader.

8. CONCLUSION

The class of memory accesses to private or read-only cache blocks have little need for cache coherence, and only the Shared R/W blocks require cache coherence. We devise novel cache coherence protocols, SWEL and RSWEL, that attempt to place data in their "optimal" location. Private or read-only blocks are allowed to reside in L1 caches because they do not need cache coherence operations. Written-Shared blocks are relegated to the L2 cache where they do not require coherence either and service all requests without the need for indirection. The coherence protocol is therefore more about the classifying blocks into categories, instead of being about the tracking of sharers. This leads to a much simpler and storage-efficient protocol. The penalty is that every miscategorization leads to recovery via a bus broadcast, an operation that keeps the protocol simple and that is exercised relatively infrequently. We show that RSWEL can improve performance in a few cases where read-write sharing is frequent, because of its elimination of indirection. It under-performs a MESI directory protocol when there are frequent accesses to blocks that get relegated to the L2. In terms of overall performance, MESI and RSWEL are very comparable. While RSWEL incurs fewer coherence transactions on the network, it experiences more L2 look-ups. The net result is a slight increase in network power. Our initial analysis shows that RSWEL is competitive with MESI in terms of performance and slightly worse than MESI in terms of power, while doing better than MESI in other regards (storage overhead, complexity). The argument for RSWEL is strongest when multi-cores execute workloads that rarely require coherence (for example, multiple VMs or message-passing programs). We believe that the exploration of additional optimizations for RSWEL may enable it to exploit its full potential. We believe that it is important to seriously consider the merits of a protocol that shifts the protocol burden from "tracking sharing vectors for each block" to "tracking the sharing nature for each block".

9. REFERENCES

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in a CC-NUMA Architecture. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002.

[2] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. The Use of Prediction for Accelerating Upgrade Misses in CC-NUMA Multiprocessors. In *Proceedings of PACT-11*, 2002.

[3] A. Alameldeen and D. Wood. Variability in Architectural Simulations of Multi-Threaded Workloads. In *Proceedings of HPCA-9*, March 2003.

[4] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[5] C. Benia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical report, Department of Computer Science, Princeton University, 2008.

[6] J. Brown, R. Kumar, and D. Tullsen. Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures. In *Proceedings of SPAA*, 2007.

[7] J. A. Brown and D. M. Tullsen. The Shared-Thread Multiprocessor. In *Proceedings of ICS*, 2008.

[8] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam. Compiler-Directed Page Coloring for Multiprocessors. *SIGPLAN Not.*, 31(9), 1996.

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

[10] N. Eisley, L.-S. Peh, and L. Shang. In-Network Cache Coherence. In *Proceedings of MICRO*, 2006.

[11] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *Proceedings of HPCA*, 2008.

[12] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-Optimal Block Placement And Replication In Distributed Caches. In *Proceedings of ISCA*, 2009.

[13] J. Huh, D. Burger, J. Chang, and G. S. Sohi. Speculative Incoherent Cache Protocols. *IEEE Micro*, 24(6):104–109, 2004.

[14] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence Decoupling: Making Use of Incoherence. In *Proceedings of ASPLOS*, October 2004.

[15] L. Jin and S. Cho. Better than the Two: Exceeding Private and Shared Caches via Two-Dimensional Page Coloring. In *Proceedings of CMP-MSI Workshop*, 2007.

[16] C. Kim, D. Burger, and S. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches. In *Proceedings of ASPLOS*, 2002.

[17] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of ISCA-24*, pages 241–251, June 1997.

[18] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[19] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of ISCA*, pages 182–193, June 2003.

[20] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of MICRO*, 2007.

[21] P. Stenström. A cache consistency protocol for multiprocessors with multistage networks. In *Proceedings of ISCA*, May 1989.

[22] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of ISCA*, 1995.

[23] S. Yan, X. Zhou, Y. Gao, H. Chen, S. Luo, P. Zhang, N. Cherukuri, R. Ronen, and B. Saha. Terascale Chip Multiprocessor Memory Hierarchy and Programming Model. In *Proceedings of HiPC*, December 2009.

[24] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A Tagless Coherence Directory. In *Proceedings of MICRO*, 2009.