Address Translation for Manycore Systems

Scott Beamer and Henry Cook
Department of Computer Science
University of California, Berkeley
{sbeamer, hcook}@cs.berkeley.edu

ABSTRACT

One of the many challenges of designing efficient manycore systems is to determine where and to what degree shared information is cached locally. In this study we specifically address efficient solutions for distributing virtual-to-physical address translations and keeping them coherent throughout a chip multiprocessor system with hundreds of cores. We evaluate multiple mechanisms in terms of their performance and overhead with the aid of software simulation. Since TLB information is invalidated rarely, we find that the mechanisms with a fast common case performed much better, and that TLB reload overhead (and not communication) was a significant factor in the performance of many benchmarks.

1. INTRODUCTION

Multiprocessor systems present a challenge for computer architects because they by definition are designed to concurrently update machine state. This leads to an implicit tension throughout the memory hierarchy between providing coherence for all replicated state versus having to synchronize on a single shared copy. One type of locally stored state critical to performance is the physical address translation and protection information for an address in virtual memory. In most modern systems this information is cached in a structure known as a Translation Lookaside Buffer. In a large multiprocessor system, if translation information for a shared memory location is not replicated, the translation information itself may become a memory hot spot [7]. However, if multiple copies of translation information exist in local TLBs, we must ensure that the copies are kept consistent with each other and with the shared page table structure.

Several solutions to this translation coherence problem have been proposed and implemented in the distributed multiprocessor domain, but prior work leaves doubt as to which ones scale most effectively with increasing core counts. We seek to find a scheme that is readily applicable to a chip manycore processor domain, specifically energy-efficient handheld device systems. Collocating all processing elements on the same chip reduces the communication overhead of coherence traffic and means that processors will share the same off-chip memory interface, both of which could have a significant impact on the TLB coherence. The dynamic hardware partitioning capabilities of our target architecture also provide a tool for limiting the scope at which coherence must be maintained. Finally, we are very concerned with mechanisms that support highly energy efficient systems, and this informs our decision as well.

Shared address translation information is always writable, even if the mapped page is read only, since it can be changed whenever page permissions are modified by hypervisor mechanisms. Furthermore, even on a system with no TLBs, during the time when a processor has read the address translation information but before it has accessed the referenced page the processor can be considered to hold an implicit copy of the page table entry. On a multiprocessor we must use some combination of locks, interrupts or atomic operations to prevent observable states where local translation information does not match actual page table entries. This isolation can be achieved by limiting which pages' status and protections can be changed, and we must do this in a way that is aware of all PTE copies implicitly or explicitly resident in processors or TLBs. Synchronization often results in serialization, whereas optimizations that enhance parallelism can result in increased coherence network traffic.

1.1 ParLab Influence

The Universal Parallel Computing Research Laboratory (ParLab) is attempting to solve the parallel computing problem with an emphasis on what would be appropriate for a mobile consumer device. This direction causes the system to be single-socket and strongly oriented towards low power. Currently the architecture is a homogeneous grid of many tiles, each of which contains: a simple processing core, a cache, and some hardware accelerators. To leverage this hardware and to make multiprogramming easier the system is designed around the concept of hadrware partitions. A partition will be a contiguous block of cores assigned to a single application, and inside it the L2 cache is shared. The OS will be deconstructed into a thin hypervisor that can run on any core. These features combined cause an application to look monolithic within a partition.

1.2 Summary

In this paper we present the scalability, performance, and the energy efficiency of several TLB coherence schemes. Some of these attributes are evaluated purely in terms of TLB performance, while others require a view of the overhead asso-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

ciated with local data replication, and some will require a holistic view of the the system that includes cache performance and dynamically interleaved memory accesses. We evaluate the performance and energy efficiency of these designs with full system simulations based runs of the PARSEC benchmarks [1] and synthetic system overheads.

We find that TLB entry count and system page size have the most significant impact on performance of the shootdown scheme. Having cores share access to a single TLB is highly effective in reducing TLB misses, and the reduction in delay cause by TLB reloads when using hierarchical TLBs is also significant. While TLB shootdown requires many orders of magnitude more messages than the validation, the added synchronization it requires does not have a significant impact on performance.

2. RELATED WORK

There has been much previous work that explores the impact of TLB characteristics and designs on both uniprocessor and multiprocessor systems. These studies have focused on TLB reload mechanisms, memory hierarchy placement, and coherence mechanisms. A wide range of metrics have been used to gauge the impact of translation design decisions on overall system performance.

Jacob et al. [3] perform a comparative study of several TLB refill mechanisms and page table organizations. They find that system performance is sensitive to TLB and cache size and that interrupts become a increasingly significant source of memory system overhead as more instructions are executed concurrently. They find that purely hardware managed or purely software managed address translation schemes are the least dependent on the cost of interrupts as caches grow larger.

On a miss, hardware-managed TLBs freeze the processor pipeline but don't flush it. If cores can execute instructions out of order, they may even continue to execute instructions during the miss. The worst-case cache effect is a few lines of data evicted [4], but the page table organization is defined permanently by the hardware mechanisms. Hardware managed schemes do not place any limitations on the types of caches that can be used in the memory hierarchy. Our schemes assumes software refill but should work with a hardware refill scheme.

Jacob et al. [5] also propose and evaluate a purely software scheme that obviates the need for TLBs entirely. This scheme is dependent on using entirely virtual caches, and assumes an addressing scheme that removes homonyms or synonyms. ASIDs and protection bits are added to the cache to improve performance. The system takes an interrupt on every L2 miss, and the OS handles the translation itself. They found that for this scheme the VM system becomes much more sensitive to the cache performance. The interrupt overhead decreases as cache resources increase, rather than remaining constant. In another study that supports these design choices, Qiu et al. [8] find that due to the cache filtering effect and the relative ineffectualness of TLBs, a TLB that is placed after a large virtual cache might be eliminated entirely without affecting system performance significantly.

Software-managed hardware TLBs are the third option in this space. On a TLB miss, software-managed TLBs raise an interrupt, and trap into a software routine for walking the page table. This allows the page table structure to be defined by software, giving the system greater flexibility. However, the table walking code must already be in the instruction cache on a TLB miss or the miss penalty will be significantly greater than that encountered in a hardware-managed system. If precise interrupts are supported by the processor then missing in the TLB will result in a pipeline flush, which is especially damaging to complex core performance. Nagle et al. [6] find that the flexibility outweighs the higher miss cost.

Black et al [**?**] propose a method for maintaining PTE coherence in a multiprocessor system termed 'TLB Shootdown'. This method, discussed in greater detail in the following section, is centered around synchronizing processor accesses to the page table, locking out other processors during PTE updates, and broadcasting directives to force all processors to flush outdated data. Adds element of synchronization. This technique is commonly employed in modern SMP operating systems (i.e. SMP-Linux); the OS is responsible for managing the shootdown process on TLB fills. We evaluate the performance and overhead of this technique in our study.

Teller et al. [9] propose 'Read-Locked TLBs' as a straighforward approach to TLB consistency in the NYU Ultra-Computer. Each processor maintains a read lock on any PTEs currently in its TLB. Unsafe modifications are only allowed to unlocked PTEs, and so the OS tries to avoid cases requiring unsafe modifications while also preventing deadlock. Unlike TLB shootdown, this scheme has a low network overhead. However, the mechanism for deadlock avoidance implicitly ties the sizes of TLBs to the size of memory partitions [9]. TLBs do not have to flushed on page evictions [10].

A second scheme proposed by Teller et al. is termed 'Validation' [9]. Rather than ensuring that TLB information is always valid, this scheme allows TLB entries to become stale, and then detect accesses that have used stale translation information. The memory controller must do additional computation to check stamps on every access [2]. The benefit of this scheme is the low overhead: modifying processors operate independently and with constant overhead, and since fixing stale data only requires one extra round trip, each processor can independently but simultaneously ensure the consistency of a distinct page/entry. This mechanism is one of the ones we evaluate in this study.

3. TLB COHERENCE SCHEMES

While similar types of coherence problems have been rigorously studied in the case of general purpose caches, some special properties of TLBs may offer opportunities for more efficient solutions. Employing optimizations required to achieve good performance in a general purpose cache hierarchy is wasteful if some of these mechanisms will be under-utilized. Furthermore, the events both causing and resulting from TLB miss may be significantly different from the actions that must be taken in the event of cache miss on the same system. Chip multiprocessor solutions also may differ from the previously investigated multiprocessor TLB coherence solutions because of the differences between distributed shared memory and a local memory with multiple memory controllers.

The translation information stored in TLBs has severeal attributes which make solving the TLB coherence problem easier than solving the cache coherence problem (Teller 1990). By leveraging these traits we can create a more efficient coherence scheme. For example, the number of references to

address translations per modification of that translation is very large. Address translation information can reside anywhere on the path from a processor to memory, and some schemes also make use of "optimistic recovery" where incorrect translations are corrected when they reach memory. Finally, there is lots of flexibility in picking which PTE to modify unsafely (i.e. which page to evict).

Based on our study of related work, and analysis of the needs of the target manycore system, we select four TLB configurations and associated coherence schemes to simulate. These schemes are intended to be representative of four important points in the design space of potential solutions.

3.1 TLB Shootdown

Each processor has its own private TLB, which is consulted in parallel with a virtually indexed, physically tagged L1 cache. The L2 cache is physically indexed and tagged. We assume that this TLB is reloaded by software. On a PTE update, the modifying processor locks the page table (preventing other processors from accessing it), and flushes its own TLB to writeback the changed pages. Simultaneously, the modifying processor broadcasts a message to all other processors that are using the same page table. This message causes an interrupt on each processor and causes it to flush its own TLB. The original processor waits for confirmation messages from all other processors that they have completed their TLB flushes, and then finally unlocks the page table. TLB coherence is maintained by the page table locks, broadcast messages and mandatory synchronization.

3.2 Validation

All caches in this system are virtually tagged and indexed. When a miss in the virtual L2 cache is reported to a processor, the processor consults its private TLB for the translation of the missing virtual address. In other words, the TLB is only consulted on L2 cache misses. The TLB does not only contain the translation information for a given page, but also a timestamp (encoded as a generation count) that is set when the TLB is filled. Generation counts associated with a given page of physical memory are also stored in a generation count table maintained at the appropriate memory controller.

When a processor makes a memory access using translation information from its TLB it also sends the memory controller the generation count of the translation it used. The memory controller checks the generation count of a potential access before it processes the request. If the generation count is out of date, the memory controller fixes the request, and sends up-to-date translation information back to the TLB. The system must incorporate a mechanism for handling generation count wraparound – Teller et al. [10] suggests the use of counters or epochs.

On PTE updates, coherence is not explicitly maintained. Old copies of the modified PTE are allowed to remain in their private TLBs, and no messages are sent to flush the TLBs. However, the generation count associated with the physical address is updated by a message from the processor making the modification. Any memory accesses made using out-of-date translation information will be detected when the generation counts are compared at the memory controller. TLB coherence is maintained by the atomicity of the update to the generation count and the fact that all future accesses made with stale information will be detected at the memory controller.

By having a virtual cache, the translation process is pushed back so it is taken out of the critical path which could improve both performance and energy efficiency. This will reduce the access time to the L1 cache and the L2 cache because no translation will need to be done along the way. A system very concerned about access latency could take an eager approach and pass the address to the TLB in parallel so when it misses in the L2 the correct page is already selected. Having the TLB farther from the core reduces the access latency and area restrictions, allowing for a bigger TLB which could also improve performance. Having a virtual cache could even save power because the costly associative lookup will only be performed on cache misses rather than on every access, so well designed circuits could save energy in this way.

3.3 Shared TLBs

On a chip multiprocessor, the potential exists for multiple processors to share a single TLB. This solution has not been considered in previous work because it does not make sense in a multiprocessor context. Assuming all processors are using the same address space, and potentially running the same code, overall performance may benefit from constructive interference. We evaluate a system where groups of four cores share local TLBs. Coherence is maintained using the TLB shootdown method described above, but the amount of coherence traffic required is reduced because the number of private entities is quartered. Hopefully if processors are working on the same code, the performance benefit will come from cores constructively interfering with eachother. If the execution model always has all cores that share a TLB running the same application, the cache could be virtually cached. In such a system, synchronization is maintained by messages passed between the the shared TLB hardware, but all processors will still be interrupted while the shared TLBs flush pages that have been modified.

3.4 Hierarchical TLBs

Another enhancement that may improve the performance of local TLBs is to include a second level of TLBs in the system. These L2 TLBs serve the same role as the L2 cache included in most modern memory hierarcies – capture a larger working set, keep relevant data on chip for a longer time, and improve the reload time of the L1 TLB. We assume that hardware is in charge of refilling the L1 TLB, but that L2 TLB reloads are handled by software. An L2 TLB may shared by several L1 TLBs. There are multiple possible coherence schemes possible both between levels and among TLBs of the same level. In this study, we evaluate a system where each processing element has its own L1 and L2 TLBs. The L2 TLB is inclusive, and coherence is maintained between the L2 TLBs via shootdown, with both the L1 and L2 TLBs being flushed on shootdown events.

4. METHODOLOGY

4.1 Simulator

We use the Virtutech Simics ISA simulator [11] to explore the design space of possible TLB solutions. Simics is capable of running full, unmodified operating systems and unmodified binaries. We run Solaris 10 on simulated Ultra-

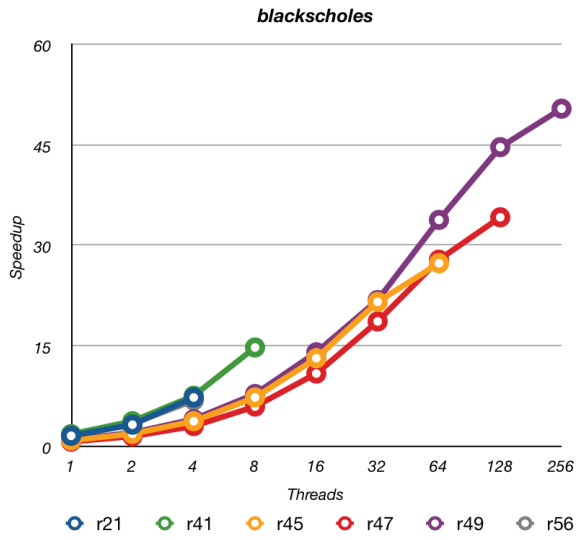

Figure 1: ▬

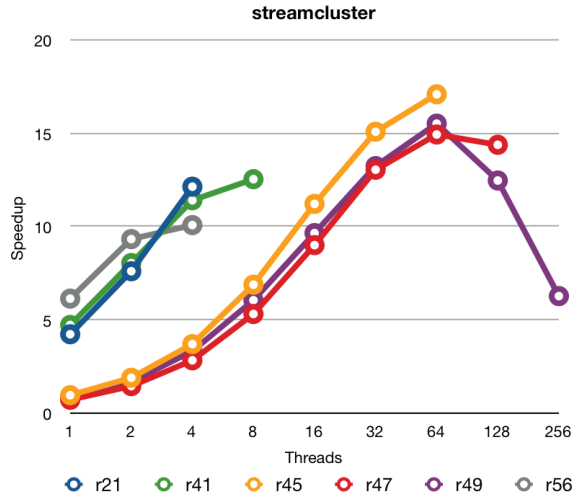

Figure 2: ▬

SPARC III processors, and we create machines with up to 128 processors.

Simics is fast enough that even when running simulations with over 100 processors we still are able to use the largest input set included in the PARSEC benchmark. However, Simics achieves this speed by making every instruction occur atomically in a single cycle. In order to add realism to the simulation, Simics can be enhanced with device modules that add delay cycles to the simulation according to effect the instructions have on the memory hierarchy. We attach modules that simulate caches, various types and configurations of TLBs, and the interface to off-chip memory.

4.2 Benchmarks

PARSEC. Descriptions of the three specific ones we are using. Working set size, synchronization overhead, observed results on real hardware. We chose a set of three benchmarks from the Princeton Application Repository for Shared

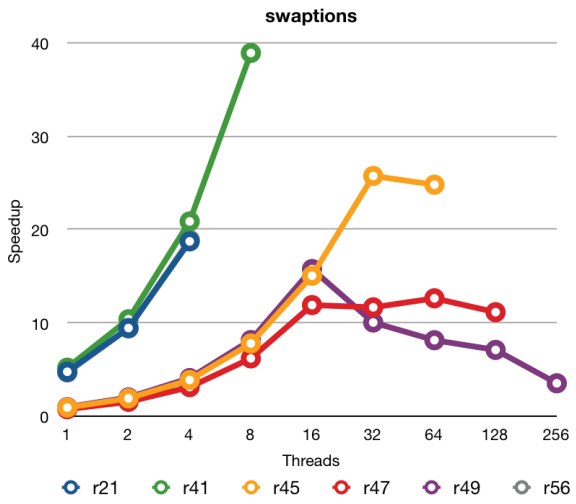

Figure 3: ▬

Memory Computers (PARSEC) **??**. These applications and kernels are appropriate for our research objectives because they are current, parallel, and represent emerging workloads. Another benefit of this suite is that each benchmark has 6 workloads, of vastly different sizes to allow the user to pick one with a working set size and runtime appropriate for their platform or software simulator. To understand the benchmarks we ran them with the 'native' workload on a sampling of current and prototype CMP's to see how well they scaled. All of them obtained good speedups on conventional x86 CMP's, but only some of them continued to get improvement from systems with much higher thread counts. This process included porting them to compile and run under Solaris and on SPARC processors. The benchmarks are implemented mostly in C, using POSIX Threads or OpenMP. The three selected for this study were pthreads based. When running them on the software simulator, we had to scale down to 'simlarge' to have reasonable simulation runtimes. When running the benchmarks on actual hardware we found the best performance was achieved when the number of threads equaled the number of cores, so we did the same for the simulations.

4.2.1 *Blackscholes*

Blackscholes is an analytical PDE solver used to compute the prices of a portfolio of European options. It uses the Black-Scholes partial differential equation which is of the form

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

. It obtains good parallel speedup even at high thread counts as shown in Figure 1. This makes sense since the application is compute-intensive, and only has 8 barriers during the runtime. With simlarge its working set is essentially 2MB for 8 threads, and it will continue to grow as the number of threads is increased.

4.2.2 *Streamcluster*

Streamcluster is a kernel from a data-mining application that is able to partition groups of points around a predetermined number of centers. As a metric for the quality of the

clustering it uses the sum of the squared distances. Depending on the dimensionality it can be very compute bound for high dimensions or memory bound for low dimension. This kernel has the most barriers of any in the suite by two orders of magnitude (129,600 for the 'simlarge' dataset). It also obtains decent parallel speedup, but after a point it saturates and increased thread counts actually degrade performance as shown in Figure 2. The working set for simlarge is about 16MB, and its access patterns exhibit a great deal of spatial locality. Its memory accesses are strongly biased towards reads by a ratio of roughly 150 to 1.

4.2.3 Swaptions

Swaptions is a Monte Carlo simulation kernel from a program that prices a portfolio of swaptions. The basic working set of the application is 512KB, and it stays constant with respect to thread count. Its percentage of data shared across many or all of the threads is the second most in the entire suite, and it protects its shared data with locks accesses, which on cache coherent shared memory could result in many invalidations. This can be seen in our studies on real hardware where it obtains decent parallel speedups until the thread count reaches 16 or 32, where it saturates(Figure 3). Looking closer at these results it saturates earlier and sharper on the systems capable of more threads because the probability of invalidations increases and they have a greater latency between caches.

4.3 Simulated System Architecture

Simics provides a functional execution engine that assumes atomic instructions that complete in a single cycle. We enhance this simulation with the inclusion of `g-cache` timing modules that simulate MESI coherent caches with configurable parameters. We also add our own custom built TLB modules that store address translation information and add reload or coherence-related communication delay to instructions who miss in the TLB or access out-of-date data. Simics provides its own functional translations, so our module only serves to provide accurate timing for TLB events. We also include some specialized simulation devices for routing memory transactions between TLBs and caches, a stalling device that mimics the delay incurred by off-chip memory accesses, and various timing and statistics gathering devices. All these devices are implemented in C, and the scripts that connect them are written in Python.

4.4 Simulated Machine Parameters

The specific TLB configurations are parameterized as described in Table 1. All cache size parameters and latencies remain constant across all TLB configurations, though they may be virtual or physical depending on the specific TLB scheme being considered. Each processing element has its own private data and instruction cache, each of which is a 16KB 4–way set associative with a one cycle access latency. All cores share a banked L2 that is 512KB, 16–way set associative, with a 10 cycle access latency. We assume an average 100 cycle memory access latency.

4.5 Synthetic OS interference

While umodified Solaris is capable of running on systems with hundreds of processors, it is not able to provide all the functionality required for some of the potential TLB schemes. Furthermore, we are conducting this study to evaluate performance under a theoretical hypervisor, not the original operating system. For both of these reasons, we choose to interject page table entry modifications into the system synthetically. This allows us to fully explore the sensitivity of a given TLB scheme to the type and frequency of modifications that are being made without specifying the precise mechanism at fault. We select a random core to stall and run the hypervisor code that updates the shared page table. We select the entry to modify according to one of three policies:

Parameter	Value
Shootdown	
TLB hit latency	1 cycle
TLB reload latency	2000 cycles
Core-to-core latency	10 cycles
Baseline size	64 entries
Baseline associativity	4
Number	1 per core
Validation	
TLB hit latency	1 cycle
TLB reload latency	2000 cycles
Timestamp check	5 cycles
Baseline size	64 entries
Baseline associativity	4
Number	1 per L2 bank
Shared	
TLB hit latency	2 cycles
TLB reload latency	2000 cycles
Core-to-core latency	10 cycles
Baseline size	256 entries
Baseline associativity	4
Number	1 per 4 cores
Hierarchical	
L1 TLB hit latency	1 cycles
L2 TLB hit latency	5 cycles
L2 TLB reload latency	2000 cycles
Core-to-core latency	10 cycles
Baseline L2 size	512 entries
Baseline L2 associativity	4
Number	1 per L1 TLB

Table 1: Summary of TLB configuration information for all architectures.

Best case policy. The page table entry associated with the page whose permissions are being modified is not cached in any TLBs.

Random policy. The page table entry associated with the page whose permissions are being modified is a random entry in a random TLB.

Worst case policy. The page table entry associated with the page whose permissions are being modified is the entry which is cached in the most TLBs.

In a classic OS, when a page is evicted it is usually a good idea to not pick a page in the TLB (like our best case). TLB schemes like shootdown will broadcast the invalidation regardless, so we wanted to model this overhead. With a thin distributed hypervisor, a partition may not be as aware of what pages in memory are in other partitions TLB's so its choice in page may seem random. To stress our TLB schemes we used the worst case policy to see how much impact page selection could have.

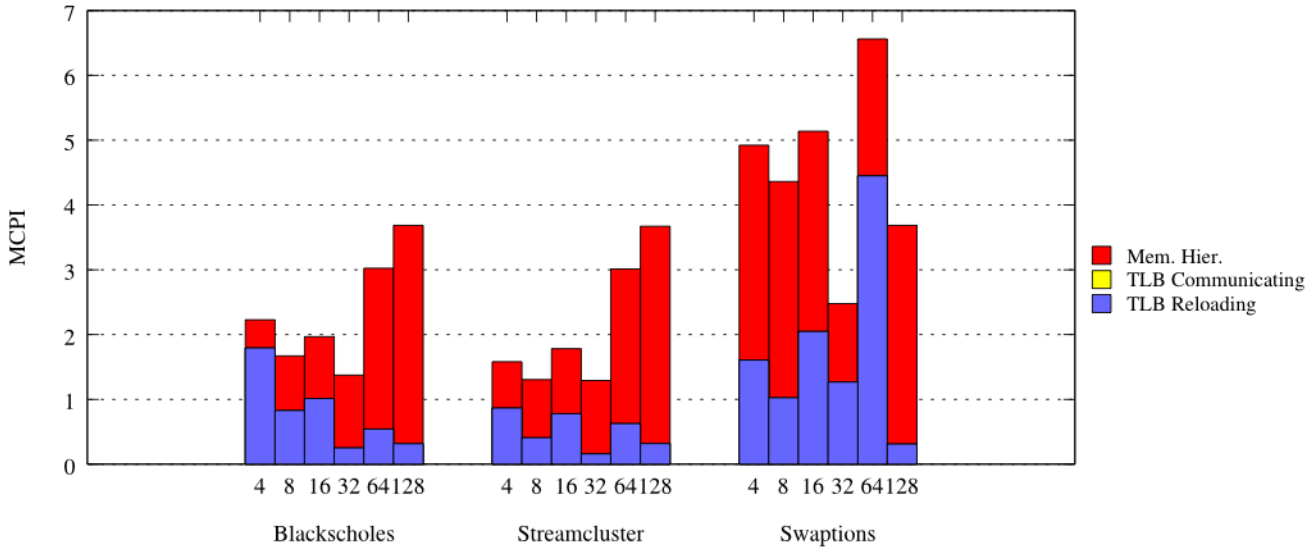

Figure 4: MCPI results for all benchmarks on a 128 entry, shootdown-based coherence TLB system

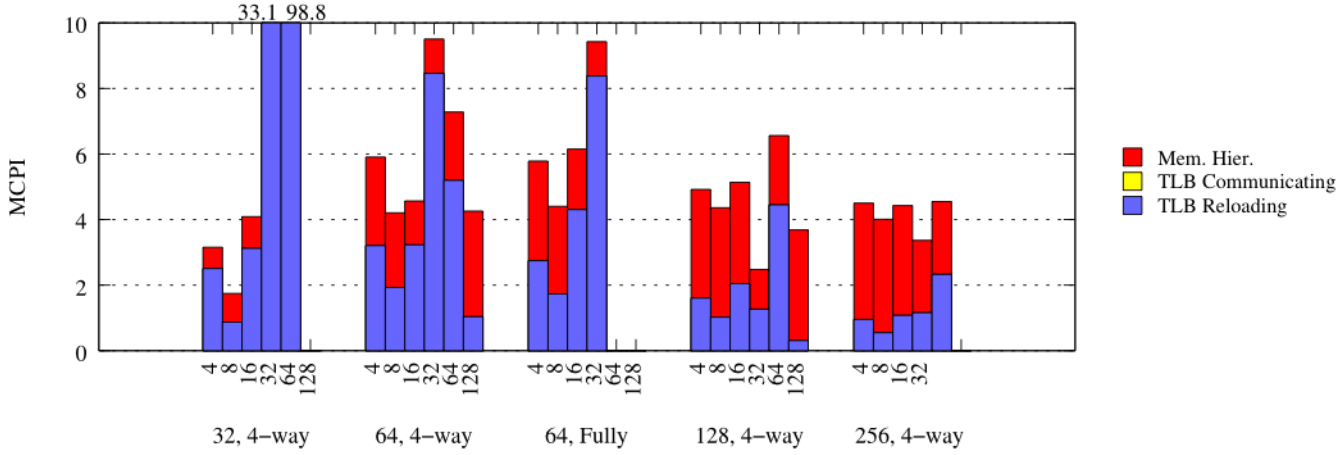

Figure 5: Effect of private TLB size on MCPI of shootdown-based TLBs

PTE modification rates are varied between once every ten million cycles and once every hundred thousand cycles. These rates correspond to theoretical unsafe modifications to page table entries, which might for example occur when pages change from resident to non-resident status. The invalidation rate was also simulated at a high (and unrealistic) rate to fully stress the translation system and to discover at what invalidation rates performance is significantly impacted. Thrashing will not only be detrimental to performance, but will also be energy wasteful. A well tuned application and system should invalidate rarely to achieve good performance and energy efficiency.

Our studies considered only one running application at a time, but this is reasonable given our focus is on performance within a hardware partition. We attempt to generically approximate the effects of hypervisor activity, novel OS mechanisms, or programs running in other partitions using our synthetic invalidation scheme.

4.6 Infrastructure Development Costs

This research made use of data collected from over 340 simulations taking about 900 total hours of simulation, spread out over weeks on a cluster. To coordinate this many runs and to efficiently batch them out, we developed 600 lines of configuration and automation scripts. We also wrote over 2500 lines of code to create the required Simics modules. Care was taken in the design of these simulator devices to keep things modular and built around a standard interface, allowing us to change the simulated architecture. This let us set some of the architectural features as parameters that could be set at simulation time, and change modules without recompiling.

We choose to use Solaris to be able to scale to higher core counts (generic Linux is capped at 32 processors), and to simulate the SunFire 6800 Platform to get many cores. This target model allowed us to change a few parameters to simulate that many cores. Getting Solaris 10 running was also non-trivial as it had to be installed inside the simulation itself. Even running in the least accurate simulation mode, Simics is still many times slower than original hardware –

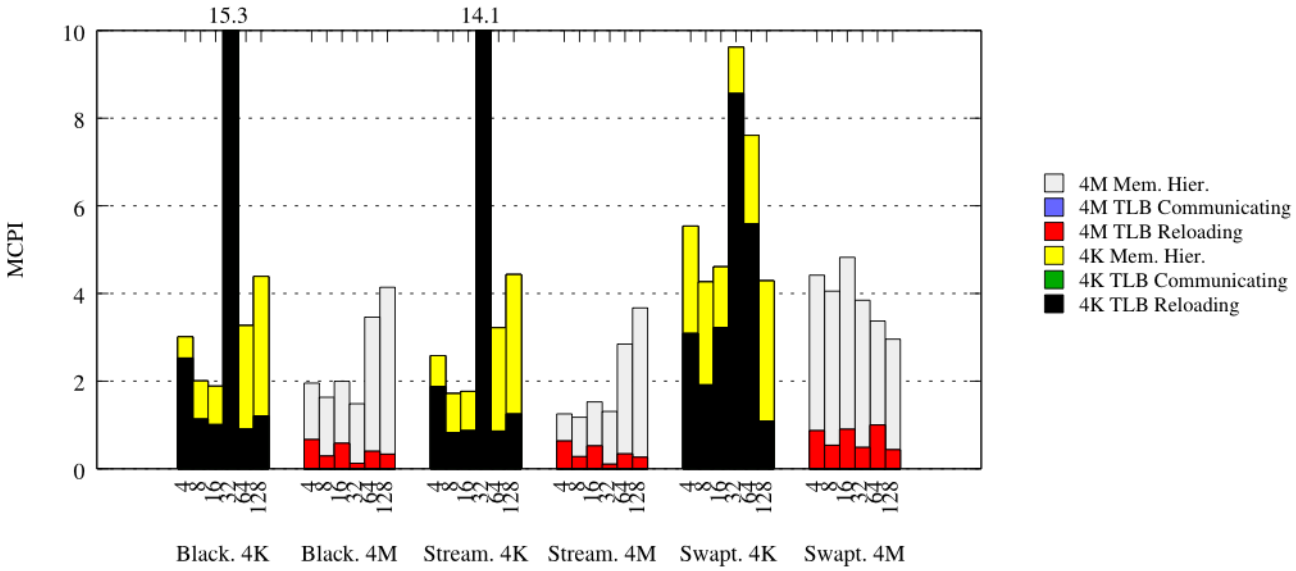

Figure 6: Effect of 4MB and 4KB pages on MCPI of shootdown-based TLBs

installing the OS took days. Getting a newly installed OS configured inside a simulated environment and talking to the outside world was a nontrivial undertaking.

Getting PARSEC to run on both a SPARC and Solaris system also took considerable effort. A number of libraries had to be installed to meet all the dependences, and many of the build paths had to be adjusted. Some of the kernels had small bugs that had to be fixed to get them to run correctly. These ranged from syntax errors that a more tolerant compiler on another platform would ignore to poor use of scoping of data types resulting in the wrong types being used. The worst bugs arose from differences in endianness that needed to be fixed in both the input files in binary format and the arithmetic kernel that processed them.

5. RESULTS

All simulated results reported in this section were collected during a one billion cycle observation period of benchmark executions Caches and TLBs were warmed for ten million cycles before any statistics were collected. Unless otherwise stated, figures in this section are based on simulations where PTE modifications are injected every 100,000 cycles, and PTEs are selected for modification based on the worst case policy described in Section 4.5.

We report the results of our simulation studies in this section using three metrics. The first metric, which we term 'MCPI' (memory access cycles per instruction), is the average number of cycles per instruction added by interactions or events occurring anywhere in the memory hierarchy or TLBs. This includes L1 cache misses, L2 cache misses, cycles spent reloading the TLB on a miss, and cycles spent communicating TLB coherence information or synchronizing the TLBs. We break up MCPI into its various components to properly attribute increased memory access time to the party at fault. MCPI is a useful metric because it serves to normalize performance across widely varying thread counts.

The second metric we use is the normalized count of cycles spent idling on memory hierarchy events. This count includes L1 hit times and is used when comparing the very different system architectures used to implement the shootdown and validation schemes. The final metric we use is simply the raw number of hardware messages that have to be sent out to maintain TLB coherence. This metric is important because increased interconnect traffic can have a significant impact on energy efficiency.

Our initial studies are concerned with probing the performance of the straightforward shootdown-based architecture described in Section 3.1. This system has physically tagged caches and a single TLB associated with each core. Figure 4 shows the baseline MCPI of a system where each TLB has 128 entries and is 4-way set associative. Observe that `blackscholes` and `streamcluster` are similar, with increasingly poor cache performance as core count increases. `Swaptions` is different, possibly because of locking or degree of sharing.

The overhead of communication and synchronization between the TLBs is not even visible on these graphs (it is less than 0.1% of a cycle). However, we must keep in mind that there is no processor pipeline to be flushed on the interrupts. Even when the rate of PTE modification is raised by 100 times, the impact is still insignificant. We estimate that in order for the overhead of maintaining coherence to be noticeable relative to the time spent in the memory hierarchy or refilling the TLB, PTEs would have to be modified approximately once every 1000 cycles. While this may be conceivable for some exotic OS mechanism, it is unlikely. PTE write policy has little effect on shootdown-based systems because the flush requests must be broadcast to all TLBs irregardless of whether or not they actually cache the modified PTE. The overhead associated with popular PTE being evicted by the worstcase PTE modification scheme did not result in a significant TLB performance difference in any of our studies.

TLB size, in terms of the number of entries included in each private TLB, turns out to be one of the parameters with the most significant impact on MCPI. Figure 5 plots MCPI for the `swaptions` benchmark across different TLB sizes and

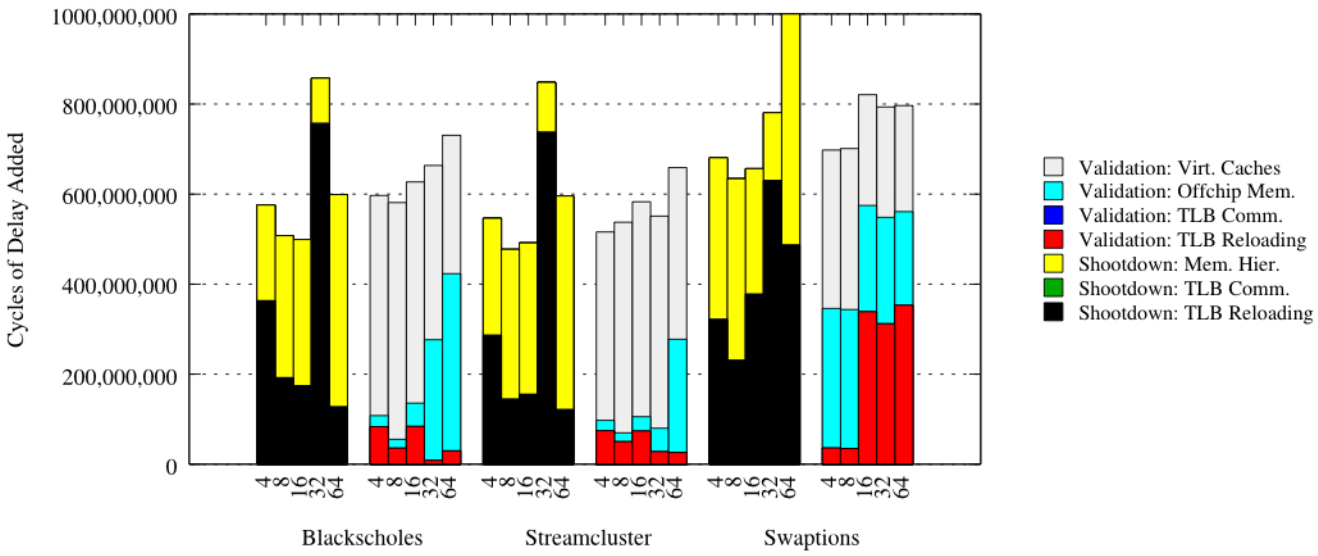

Figure 7: Comparison of validation and shootdown coherence mechanisms

associativities. Small is very bad, large is better, though cache effects seems to get worse as TLB reload effects drop. Associativity has an insignificant affect, possibly because the TLB is already big enough at 64 entries to not suffer from conflict misses.

The system page size also has a significant impact on MCPI. Unsurprisingly, large pages (4M) significantly reduce the cycle overhead associated with TLB reload events. Figure 6 illustrates this clearly for a system with 64 entry, 4-way associative TLBs. Note the emergent behavior for 32 cores, visible across all benchmarks, and eliminated by large page size. The hidden cost of large pages is fragmentation, but these simulations and our metrics do not capture this.

Having completed an exploratory search of the shootdown-based design space at all core counts, we turn our attention to architectural and coherence alternatives. Figure 7 uses the overall cycles of delay added metric to compare the validation and shootdown coherence schemes. Recall that the validation scheme makes use of virtually tagged caches, whereas the shootdown mechanism system uses physical caches. In our simulations, we found that TLB-related delay is reduced by validation, but the performance overhead of the virtual caches is significant enough close the gap between the two schemes.

The other enhancements we applied to the shootdown-based machine were to introduce shared TLBs and hierarchical second level TLBs. A system where every four cores share a single 256 entry, 4-way set associative TLB are compared with the generic 64 entry private TLB architecture in Figure 8. Sharing vastly reduces delay caused by TLB reload events for all benchmarks and core counts. This can in large part be explained by many cores operating in the same address space and running the same application.

Adding hierarchical second level TLBs (in our study, these have 512 entries, 4-way set associative) behind every private TLB serves to improve MCPI by reducing the penalty associated with a TLB reload in the common case. L2 TLBs are inclusive, and maintain coherence amongst themselves while also directing their associated L1 TLB to flush as well when required. While we do not share L2 TLBs in this study to avoid confounding effects, it is likely that combined hierarchical and sharing TLBs in the same system would be a viable design point to consider. Figure 9 shows the effectiveness of the L2 TLBs at reducing the MCPI associated with TLB reload events, but also reveals a surprising phenomenon. Use of hierarchical TLBs appears to have a detrimental impact on the MCPI associated with the physical caches used in both systems. This effect is present across all benchmarks.

Our final study examines the effect of architecture and coherence mechanism on the number of messages sent by the system. This study is illustrated by Figure 10, and the benchmark used is `blackscholes`. There is a tremendous range in values, as shown by the logarithmic scale. These results were collected from the same machines used in the previous studies under two rates of PTE modification. Validation is evaluated across all three PTE modification policies (shootdown-based schemes send the same number of messages regardless of PTE modification policy). Note that under the bestcase policy, the validation scheme requires absolutely no coherence traffic. While more messages are required for harsher policies, validation still uses orders of magnitude fewer messages than shootdown. The difference becomes even more apparent when the PTE modification rate is increased by two orders of magnitude. The shootdown-based system with shared TLBs scales up at the same rate as the generic shootdown machine, but sends only a quarter as many messages.

However, while the number of messages sent by the shootdown and validation schemes are many orders of magnitude different, the actually delays caused by this communication are insignificant when compared to the other sources of delay in the system. This is mainly because PTE modification events are extremely rare relative to memory accesses. However, the energy required to send messages over the chip interconnect should not be disregarded; we need a metric that allows us to compare this communication energy with the energy consumed by, for example, larger on-chip structures. This analysis of messages sent is also useful when trying to determine which traffic types merit their own physical net-

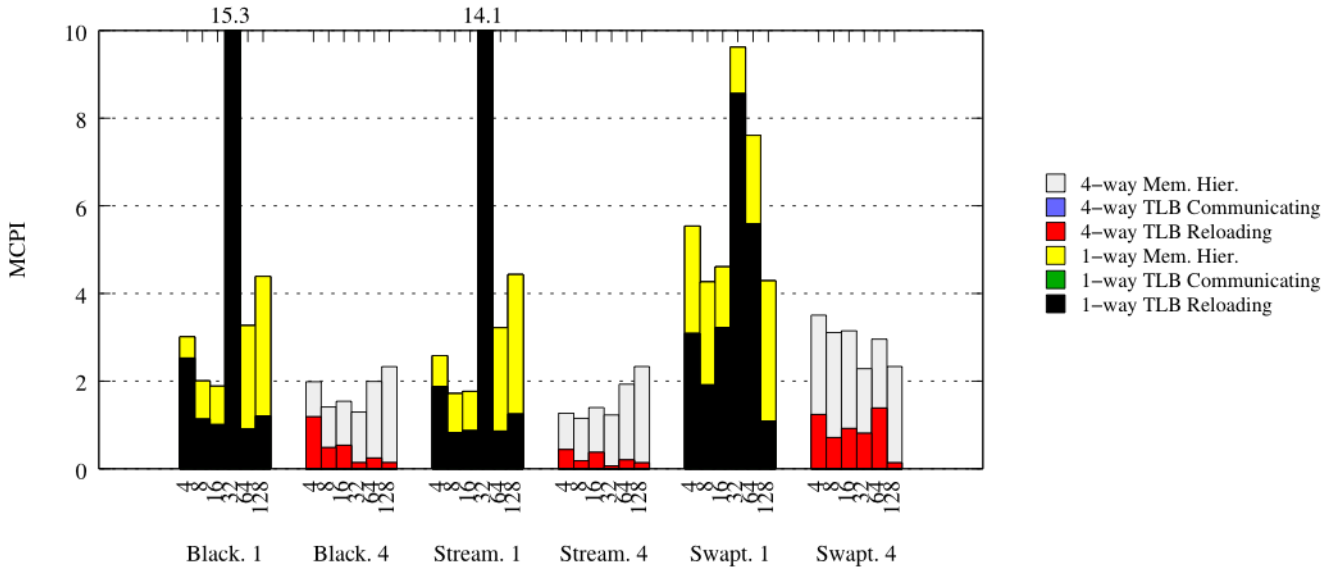

Figure 8: MCPI of shared TLBs for all benchmarks

works.

6. FUTURE WORK

This research could be furthered in several ways, but what has been accomplished has established a good foundation that gives a sense of better ways to go in the future. Of course using more applications and running simulations for more machine parameters could be instructive, but we feel the best use of time would be to test more TLB schemes. Many of the schemes presented in this paper could be composed to make hybrids that would hopefully benefit from all the included ideas. For example a combination of sharing and hierarchy with virtual caches could result in a very fast and energy efficient system. It tries to get the maximum gains from locality to improve performance and save energy by reducing the distance and amount of TLB traffic. Building a reasonable power model could be very helpful in comparing TLB schemes to give a more rigorous justification of power differences.

7. CONCLUSION

In this paper we examined several TLB architectures and compared their performance on a handful of parallel benchmarks with the help of a software simulator. We found that for performance, the rareness of TLB invalidations made the penalty for handling invalidations poorly insignificant. Larger TLBs and larger page sizes were both able to improve the performance of the shootdown scheme. The validation scheme produced orders of magnitude less messages than the shootdown scheme, but performed worse overall due to the virtual caches needing to be flushed. Sharing and adding a second level TLB both proved to improve performance by reducing the TLB refill delay. Considering these results, a recommend scheme would be a shootdown scheme with sharing and second level TLBs.

8. REFERENCES

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and the parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[2] D. R. Emberson. A cache coherence management technique for hypercube multiprocessors. In *International Conference on Parallel Processing*, pages 262–265, August 1987.

[3] B. Jacob and T. Mudge. A look at several memory management units, tlb-refill mechanisms and page table organization. In *ASPLOS*, 1998.

[4] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. In *IEEE Micro*, July 1998.

[5] B. Jacob and T. Mudge. Uniprocessor virtual memory without tlbs. *IEEE Transactions on Computers*, 50(5), May 2001.

[6] U. R. S. T. S. S. M. T. Nagle, D. and R. Brown. Design tradeoffs for software -managed tlbs. In *In Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 27–38, 1993.

[7] G. F. Pfister and V. A. Norton. Hot spot contention and combining multistage interconnection networks, 1985.

[8] X. Qiu and M. Dubois. Moving address translation closer to memory in distributed shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):612–623, 2005. Fellow-Michel Dubois.

[9] P. Teller, R. Kenner, and M. Snir. Tlb consistency on highly-parallel shared-memory multiprocessors. Technical Report 129, Ultracomputer Research Laboratory, October 1987.

[10] P. J. Teller. Translation-lookaside buffer consistency. *Computer*, 23(6):26–36, 1990.

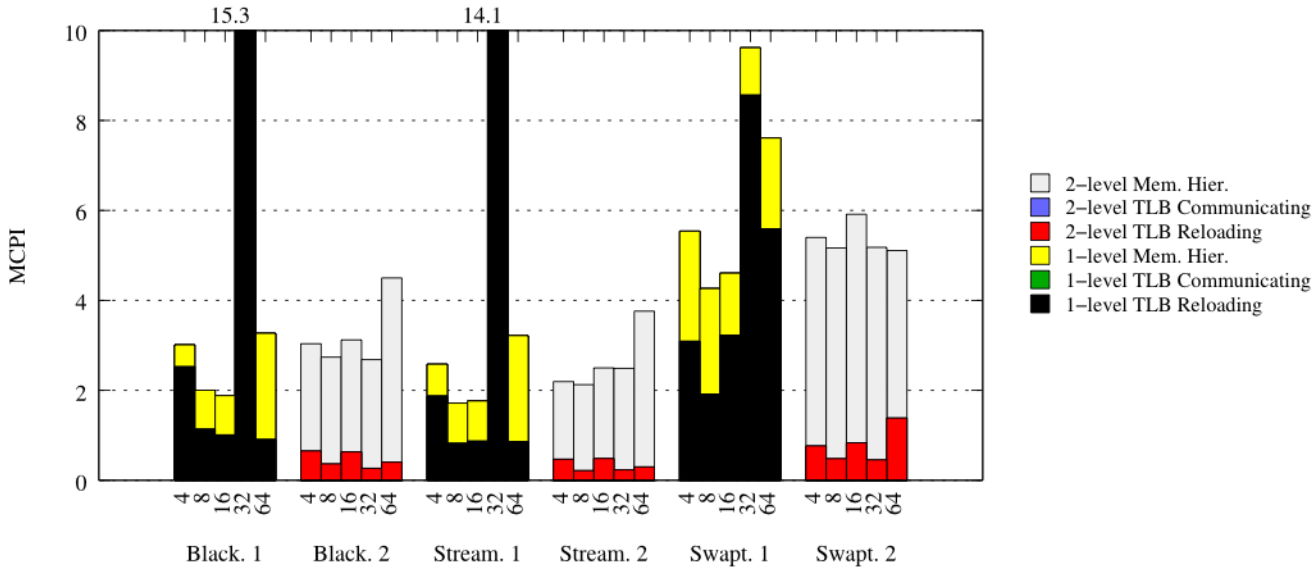

Figure 9: MCPI of hierarchical TLBs for all benchmarks

[11] Virtutech. Simics isa simulator. www.simics.net, 2008.

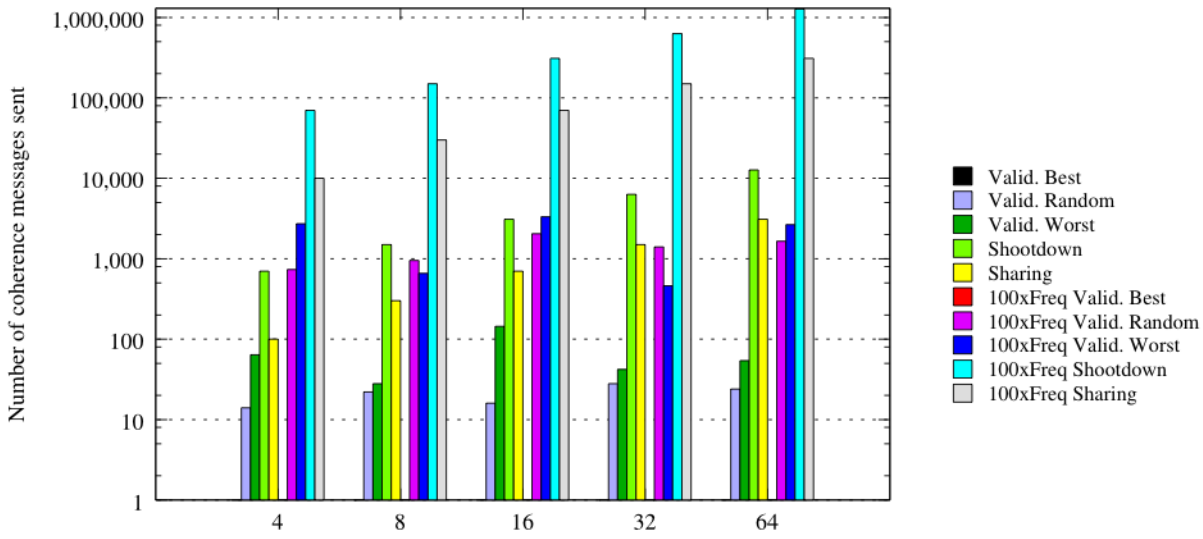

Figure 10: Coherence messages seen for different TLB schemes