VIRTUAL HIERARCHIES

ABUNDANT CORES PER CHIP WILL ENCOURAGE A GREATER USE OF SPACE SHARING, WHERE WORK STAYS ON A GROUP OF CORES FOR LONG TIME INTERVALS. VIRTUAL HIERARCHIES CAN IMPROVE PERFORMANCE AND PERFORMANCE ISOLATION OF SPACE-SHARED WORKLOADS, WHILE STILL SUPPORTING GLOBALLY SHARED MEMORY TO FACILITATE DYNAMIC PARTITIONING AND CONTENT-BASED PAGE SHARING.

Michael R. Marty
Mark D. Hill
University of Wisconsin—Madison

Memory system hierarchies are fundamental to computing systems. They have long improved performance because most programs temporally concentrate accesses to code and data. However, emerging many-core chip multiprocessors (CMPs) provide a new computing landscape. Rather than just time-sharing jobs on one or a few cores, we expect abundant cores will encourage a greater use of space sharing, where single-threaded or multithreaded jobs are simultaneously assigned to separate groups of cores for long time intervals.

To optimize for space-shared workloads, we propose using *virtual hierarchies* to overlay a coherence and caching hierarchy onto a physical system. Unlike a fixed physical hierarchy, a virtual hierarchy can adapt to fit how the work is space shared for improved performance and performance isolation. Moreover, a virtual hierarchy might make tiled (repeatable) architectures more compelling by offering the latency and bandwidth characteristics of a physical hierarchy without actually building one.

Virtual hierarchies

Many of today's many-core CMPs use a physical hierarchy of two or more cache levels that statically determine where cache blocks allocate, how many copies of the block coexist, and how the block is found for satisfying a read request or invalidation for a write request. For example, the eight cores in the Sun Niagara processor share an L2 cache. On the other hand, cores in the AMD Barcelona processor have private L1 and L2 caches but share an L3 cache. Like many design choices, hardwired hierarchies don't adapt to the optimal arrangement for a given workload. Nevertheless, the more abundant resources of future many-core CMPs will make adaptable cache hierarchies more valuable.

A virtual hierarchy is a cache hierarchy that can adapt to fit the workload or mix of workloads. The hierarchy's first level locates data blocks close to the cores needing them for faster access, establishes a shared-cache domain, and establishes a point of coherence for faster communication. When a miss leaves a tile, it first attempts to locate the block (or sharers) within the first level. In the common case, this can greatly reduce access latency when the resources in the first level are assigned close to one another. The first level can also provide isolation between independent workloads. If the miss can't be serviced in the first level, the first-level controller invokes the hierarchy's second level.

0272-1732/08/$20.00 © 2008 IEEE Published by the IEEE Computer Society

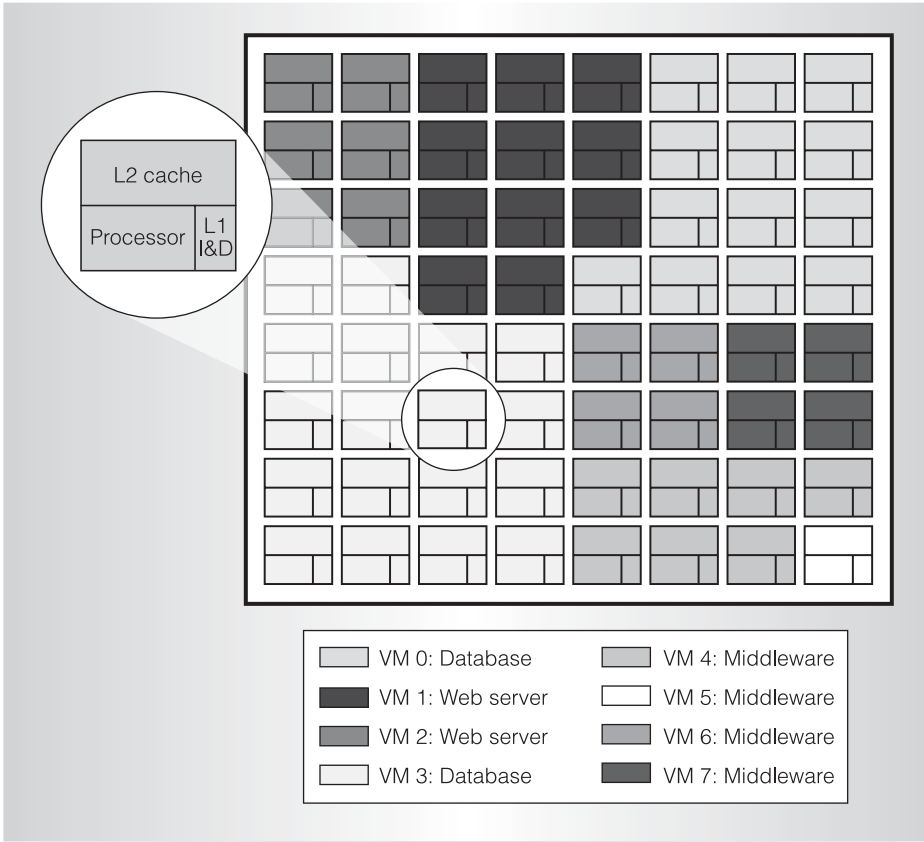


Figure 1. Tiled chip multiprocessors (CMP) running consolidated servers.

We argue for virtual hierarchies with a case study of a 64-tile flat CMP running several consolidated multithreaded workloads. As Figure 1 illustrates, we use space sharing to assign each workload to several adjacent cores. We tentatively assume that each workload runs in its own *virtual machine* (VM). However, space sharing equally applies within a single operating system.

Conventional caching and coherence schemes for tiled architectures don't optimize for space-shared workloads. A global broadcast for all coherence across a large number of tiles is expensive and time consuming. And using a global directory in DRAM or SRAM forces many memory accesses to unnecessarily cross the chip, failing to minimize memory access time or isolate performance between workloads or VMs. Statically distributing the directory among tiles can do much better, provided operating systems or hypervisors carefully map virtual pages to physical frames within the workload's assigned tiles. However, requiring the hypervisor and OS to manage cache layout complicates memory allocation, resource reassignment, and scheduling, and it might limit sharing opportunities.

For our case study, we recommend a two-level virtual coherence and caching hierarchy that harmonizes with the assignment of tiles to VMs. Figure 2 illustrates a logical view of such a virtual hierarchy. Each VM operates in its own, isolated first level of the hierarchy, which minimizes both miss access time and performance interference with other workloads or VMs. Moreover, the shared resources of cache capacity, interconnect links, miss-status handling registers (MSHRs), and more are mostly isolated between VMs. The second level maintains globally shared memory. This facilitates dynamically repartitioning resources without costly cache flushes. Furthermore, maintaining globally shared memory min-

imizes changes to existing system software and allows virtualization features such as content-based page sharing.[1]

Virtual hierarchy implementation

We implemented a virtual hierarchy with a two-level coherence protocol. The two protocols operate like a two-level protocol on a physical hierarchy but have two key differences. First, the virtual hierarchy isn't tied to a physical hierarchy that might or might not match the VM assignment. Second, the virtual hierarchy can change dynamically when VM assignment changes.

First-level protocol

We first developed a first-level directory protocol for intra-VM coherence. When a memory reference misses in a tile, it is directed to a home tile that either contains the block with appropriate permission, a directory entry (in an L2 tag) pertaining to the block, or has no information. The latter case implies the block isn't present in this VM. When a block isn't present or the directory entry contains insufficient coherence permissions—for example, when the new request seeks to modify the block and it finds only a shared copy—the request is issued to the directory at the second level.

A surprising challenge for an intra-VM first-level protocol is finding the home tile (that is local to the VM) for any given memory block address. For a system with a physical hierarchy, the home tile is usually determined by a simple interleaving of fixed power-of-two tiles in the local part of the hierarchy. For an intra-VM protocol, the home tile is a function of two properties: which tiles belong to a VM, and how many tiles belong to a VM. Moreover, dynamic VM reassignment can change both. Also, it isn't desirable to require all VM sizes to be a power of two.

To this end, we support dynamic home tiles in VMs of arbitrary sizes using a simple table lookup that must be performed before a miss leaves a tile (but may be overlapped with L2 cache access). As Figure 3 illustrates, each of the 64 tiles includes a table with 64 six-bit entries indexed by the block number's six least-significant bits. The figure further shows table values set to distribute requests approximately evenly among the three home tiles in this VM (p12, p13, and p14). At VM (or process) reassignment, a hypervisor (or OS) would set tables. (Alternatively, a dynamic home tile could be selected on a per-page basis, either by explicitly encoding a home tile location into the page table or using automatic hardware-based predictors. We don't explore these approaches in this work.)

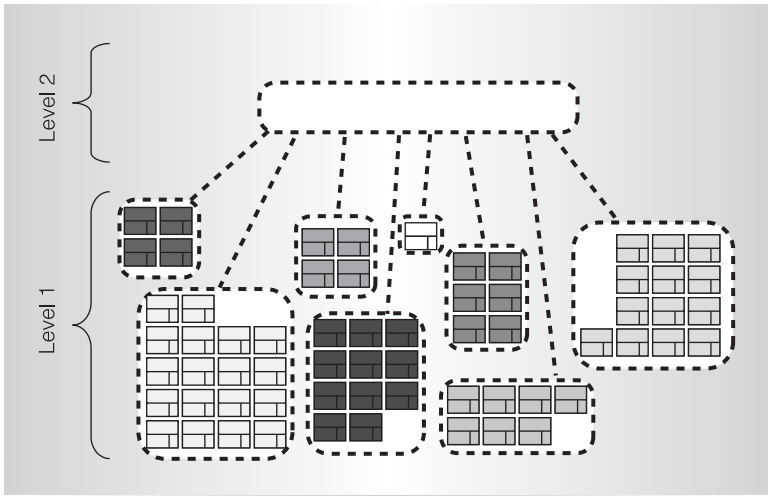


Figure 2. A logical view of a virtual hierarchy that harmonizes with workload assignments.

Similar to a tile naming the dynamic home is allowing an intra-VM (level-one) directory entry to name the tiles in its current VM (for example, which tiles could share a block). In general, the current VM could be as large as all 64 tiles, but it can contain any subset of the tiles. We use the simple solution of having each intra-VM directory entry include a 64-bit vector for naming any of the tiles as sharers. This solution can be wasteful if VMs are small, because only bits for tiles in the current VM will ever be set. Of course, more compact representations are possible at a cost of additional bandwidth or complexity.

Figure 4 illustrates how the first level protocol in our virtual hierarchy enables localized sharing within the VM to meet our goals of minimizing average memory access time and mitigating the performance impact of one VM on another. The requestor also sends a completion message

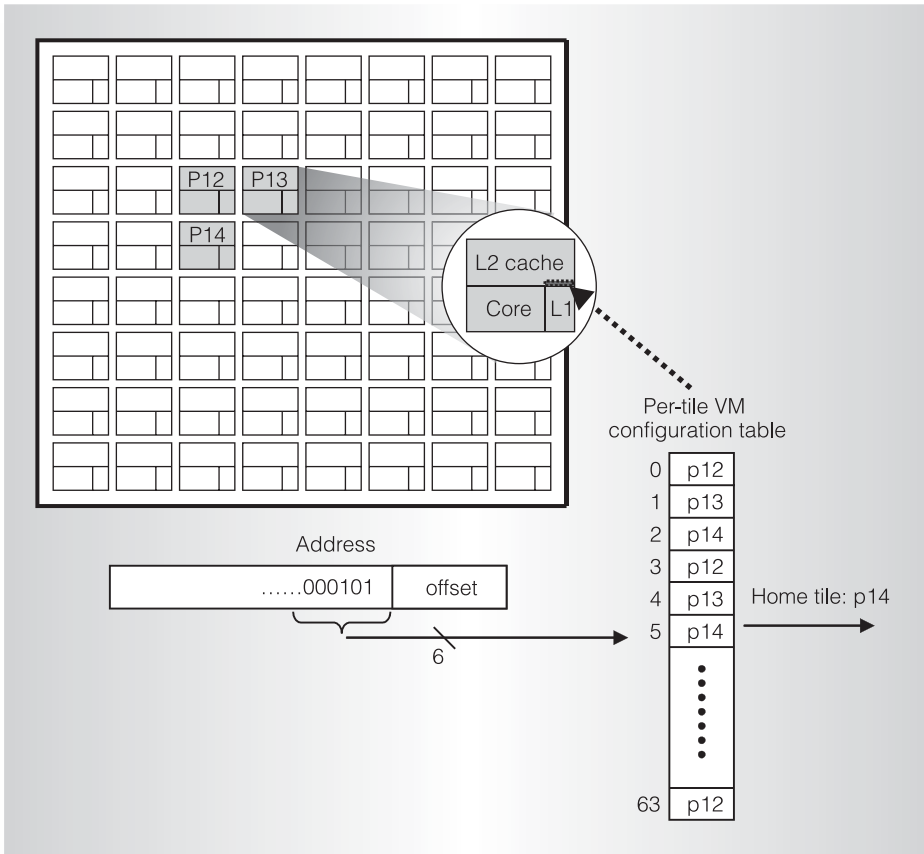


Figure 3. Tiles use a configuration table to select dynamic homes within the VM.

to the dynamic home tile upon finishing a request (not shown in Figure 4) to facilitate our race-handling scheme, which we describe in the next section.

Second-level protocols

We specify two virtual hierarchy protocol classes that differ in how they perform level-two coherence. Protocol VH_A uses a directory logically in memory. It's a virtualized version of a two-level directory protocol in a physical hierarchy with two key differences.

First, a level-two directory entry must name subsets of level-one directories. This is straightforward in a physical hierarchy where the names and numbers of level-one directories are hardwired into the hardware. In a virtual hierarchy, any tile can function as a first-level directory for any block. Therefore, we use a full bit-vector to name any tile as a possible first-level directory.

Second, a key challenge in two-level directory protocols is resolving races, a matter not well discussed in the literature. Protocol VH_A manages races within protocols and between protocols with blocking directories.[2] However in a two-level directory protocol, a naive implementation of blocking can lead to a deadlock because of dependences between blocked first-level directories. The protocol can avoid a deadlock by always handling second-level messages at first-level directories, but this might cause an explosion of states when a second-level message can interrupt any pending first-level operation. We instead establish a smaller set of *safe states*, which includes all stable states and a subset of the transient states. A safe state can immediately handle any second-level message, otherwise the second-level message must wait until a safe state is reached. First-level requests either complete or cause the first-level

directory to enter a safe state before issuing a second-level request. Safe states reduce the state-space explosion of the protocol at a cost of some concurrency between first- and second-level actions. (More details on resolving races are available elsewhere.[7])

Figure 5 shows how the second level of coherence allows for inter-VM sharing due to VM migration, reconfiguration, or page sharing between VMs.

We also propose Protocol VH_B to reduce the space requirement of the in-memory directory to a single bit per block by relying on a broadcast for second-level coherence. It reduces resource overhead, facilitates optimizations, and only uses broadcast for rarer second-level coherence. VH_B augments cached copies of blocks with a token count[3] to achieve two primary advantages. First, tiles without copies of a block don't need to acknowledge requests (one cache responds in the common case[4]). Second, the single per-block bit in memory can be thought of as representing presence of either all of the tokens or no tokens.[5] This property enables the single bit-per-block directory state. Moreover, a broadcast combined with token counting can reduce complexity[6] and enable orthogonal optimizations.

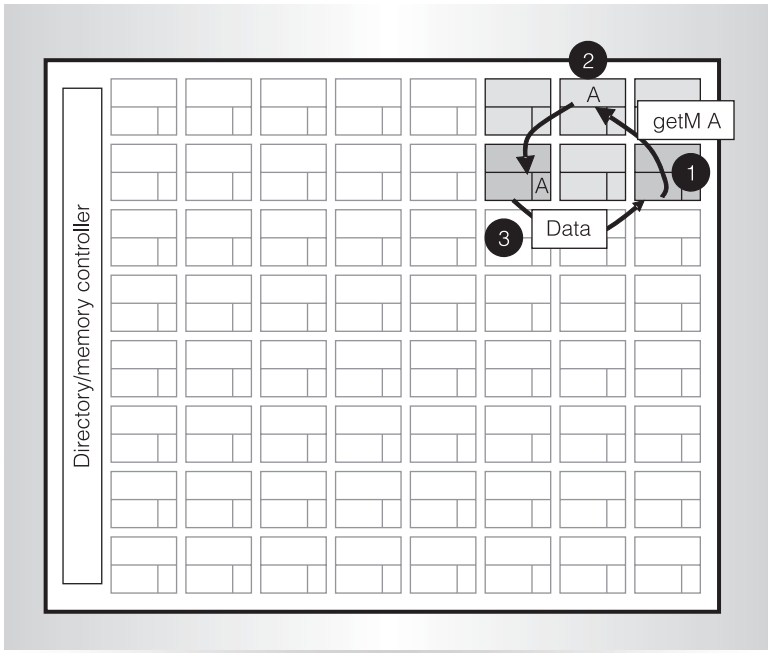


Figure 4. First level of coherence in a virtual hierarchy enables fast and isolated intra-VM coherence: A processor issues a request to the dynamic home tile that is local to the VM (1). A directory tag is found and the request redirects to the owning tile with dirty data (2). The owner responds to the requestor (3).

Discussion

Both VH_A and VH_B create a two-level virtual hierarchy that can adapt to space-shared workloads. When applied to consolidated server workloads in VMs, the virtual hierarchy minimizes memory access time, minimizes inter-VM performance interference, facilitates VM reassignment, and supports inter-VM sharing.

A virtual hierarchy can also optimize space-sharing workloads within a single operating system because global cache coherence is maintained in hardware. On the other hand, in a virtualized environment, designers might consider using a virtual hierarchy's first-level of coherence without a backing second-level coherence protocol. This option, VH_{null}, could still accomplish many of our goals with sufficient hypervisor support. Nonetheless, we see many reasons why designers might prefer VH_A and VH_B's two coherence levels.

First, VH_{null} impacts dynamic VM reassignment as each reconfiguration or rescheduling of VM resources requires the hypervisor to explicitly flush all the caches of the tiles affected and to atomically update the VM configuration tables. VH_A and VH_B, on the other hand, avoid this complexity by implicitly migrating blocks to their new homes on demand and don't require atomic updates of VM configuration tables. Second, VH_{null} might support read-only content-based page sharing among VMs, but on a miss, it would obtain the data from off-chip DRAM. VH_A and VH_B improve the latency and reduce off-chip bandwidth demands of these misses by often finding the data on chip. Third, VH_{null} adds complexity to the hypervisor because some of its memory accesses must either explicitly bypass caches or access modified data in the cache of any VM's tile. VH_A and VH_B allow the hypervisor to use caching transparently. Fourth, VH_A and VH_B's second level of coherence allows

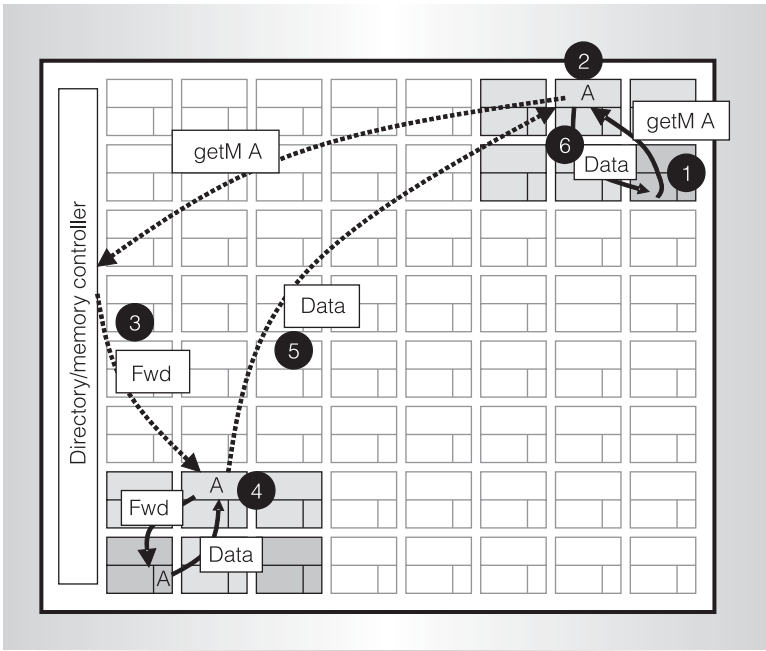


Figure 5. VH_A's second-level coherence (dashed lines) facilitates tile reassignment and content-based page sharing. The request issues to the home tile serving as the level-one directory and finds insufficient permission within the VM (1). A second-level request issues to the global level-two directory (2). Coherence messages forward to other level-one directories (3), which then in turn handle intra-VM actions (4) and send an Ack or data on behalf of the entire level-one directory (5). Finally, the originating level-one directory finishes the request on behalf of the requestor (6). (Completion messages not shown.)

subtle optimizations at the first level that aren't easily done with VH_{null}.

Evaluation

We evaluated our virtual-hierarchy protocols with a full-system simulation using Virtutech Simics (www.simics.com), extended with the GEMS toolset.[8] GEMS provides a detailed memory system timing model, which accounts for all protocol messages and state transitions.

Target system

The target system we simulated is a tiled 64-core CMP similar to Figure 1. Each core consists of a two-issue in-order SPARC processor with 64-Kbyte L1 instruction and data caches. Each tile also includes a 1-Mbyte L2 bank. The 2D 8 × 8 grid interconnect consists of 16-byte links. We modeled the total latency per link as five cycles, which includes the wire and routing delay. Messages adaptively route in a packet-switched interconnect. DRAM, with a modeled access latency of 275 cycles, attaches directly to the CMP via eight memory controllers along the edges of the CMP. The physical memory size depends on the configuration simulated, ranging from 16 to 64 Gbytes. We set the memory bandwidth artificially high to isolate interference between VMs—actual systems can use memory controllers with fair queueing.[9]

Workloads

We simulated consolidated commercial-grade server workloads, where each workload is considered a virtual machine and runs its own instance of Solaris 9. We used an online transaction processing workload (OLTP), static Web-serving workloads (Apache and Zeus), and a Java middleware workload (SpecJBB). (See related work for a description of all workload configurations.[10])

Our results show workloads of the same type and size consolidate (homogenous) and workloads of different types and sizes consolidate (heterogeneous). These homogenous configurations let us report overall runtime after completing some number of transactions because all units of work are equivalent—that is, all VMs complete the same type of transaction. For heterogeneous consolidation, the simulator ran for a chosen number of cycles, and we counted the number of transactions completed for each VM. We then reported the cycles per transaction (CPT) for each VM. To account for nondeterminism in multithreaded workloads, we pseudorandomly perturbed simulations and calculated runtime error bars to 95 percent confidence.[11] Table 1 lists the configurations.

We approximated a virtualized environment by interleaving the execution and memory references of multiple functional simulators onto a single, detailed memory timing model. Thus, we didn't simulate the execution or scheduling effects of a true hypervisor and instead statically scheduled workloads to adjacent cores.

Protocols

We implemented our protocols using writeback L1 caches, and L2 cache banks

Table 1. Server consolidation configurations.

Configuration	Description
OLTP 8x8p	Eight eight-processor OLTP VMs
Apache 8x8p	Eight eight-processor Apache VMs
Zeus 8x8p	Eight eight-processor Zeus VMs
JBB 8x8p	Eight eight-processor SpecJBB VMs
Mixed	Two eight-processor Apache VMs, two 16-processor OLTP VMs, one eight-processor JBB VM, and two four-processor JBB VMs

that don't require inclusion with the tile's local L1. VH_A implemented the virtual hierarchy with the two-level directory protocol we previously described. The L2 bank at the dynamic home tile holds the only L2 copy of a cache block, and each L2 tag contains a bit-vector of sharers.

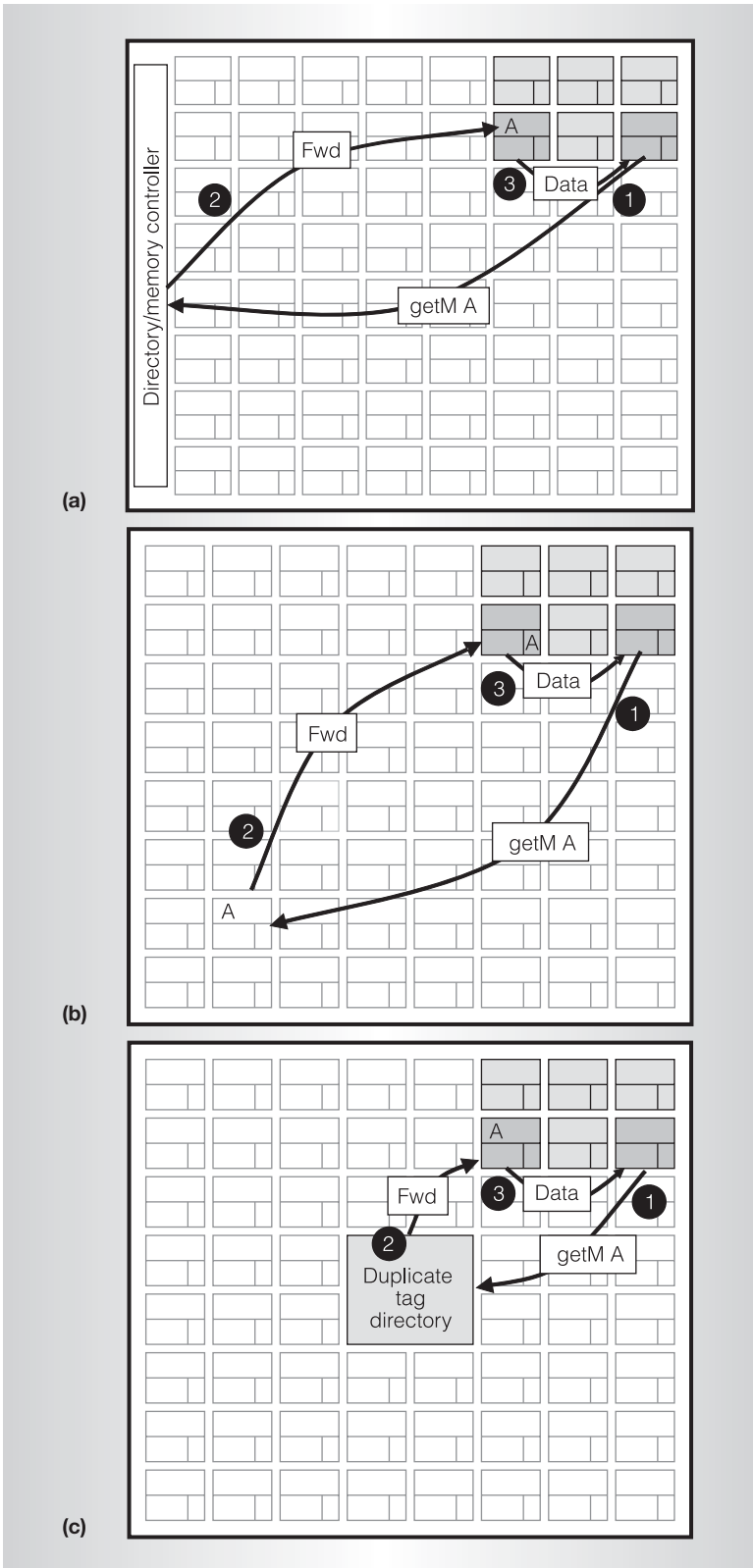
VH_B implemented the virtual hierarchy with the same first-level protocol as VH_A, but it used the token-based broadcast for second-level coherence. VH_B also included additional optimizations that we didn't implement for VH_A. One optimization placed nonshared private data in a tile's local L2 bank without allocating a tag at the dynamic home tile to store unneeded directory state. This improved access latency for private data. To distinguish between private and shared data, we initially treated a clean miss from memory as private data and denoted it with an extra bit in the L1 tag. The first sharing miss locates the block with the second-level global broadcast, which then clears the bit and allocates a tag at the dynamic home tile. Other subtle optimizations VH_B implemented include allowing the victimization of an L2 tag without invalidating active sharers and returning memory data directly to the requestor instead of first to the dynamic home tile.

We compared VH_A and VH_B against several alternative directory protocols (see the example in Figure 6). In all three protocols, a sharing miss within a VM results in a global indirection to either a directory in DRAM (DRAM-DIR), an interleaved directory based on memory address (STATIC-BANK-DIR), or a directory implemented as a duplicate tag store (TAG-DIR).

DRAM-DIR uses a full bit-vector directory implemented in DRAM. The directory considers the cache hierarchy in each tile private, but we limited the number of copies of a block that can be stored in L2 banks to increase overall cache capacity. To do so, we implemented a policy that uses a simple heuristic based on the block's coherence state. In our MOESI (Modified, Owned, Exclusive, Shared, Invalid) implementation, an L1 replacement victim in state M, O, or E always allocates in the local L2 bank. However, a block in the S-state won't allocate in the local bank because it's likely that another tile holds the corresponding O-block. If the O-block replaces to the directory, then the subsequent requestor will become the new owner. To reduce indirection overhead, we also implemented a generous 1-Mbyte directory cache at each of the eight memory/directory controllers.

STATIC-BANK-DIR implements a MESI (Modified, Exclusive, Shared, Invalid) protocol with directory state stored in the L2 tags of a statically assigned home tile for each block.[12,13] Home tiles interleave by the lowest six bits of the page frame address. L1 caches always replace the block to the static home tile. Thus, logically there is a single shared L2 cache interleaved across all tiles.

TAG-DIR implements a protocol with a directory state stored in an exact duplicate tag store[14–16] located in the middle of the chip. We charge three cycles to access the 1,024-way content-addressable memory (CAM) required to access all tags of possible cache locations—that is, 64 tiles with 16-way associative L2 caches. TAG-DIR implements the same heuristic as DRAM-DIR to increase the overall cache capacity.



Results

Figure 7 shows the normalized runtime for the consolidated server workloads running multiple VMs of the same workload type (homogenous consolidation). DRAM-DIR performed the worst because of long indirection latencies. Latencies were especially penalized by misses to the on-chip directory caches, which achieved a hit rate of 43 to 65 percent. VH_B performed 22 to 42 percent better than STATIC-BANK-DIR and 4 to 46 percent better than TAG-DIR (with its arguably unimplementable duplicate tag directory). VH_A performed similarly to VH_B and outperformed the baseline directory protocols. Nonetheless, VH_B's private data optimization gave a 6 to 8 percent edge over VH_A for three of the workloads.

Figure 8 shows the breakdown of memory system stall cycles. The bars show the normalized amount of cycles spent on servicing off-chip misses, hits in the local L2 bank, hits in remote L2 banks, and hits in remote L1 caches. The figure indicates that off-chip misses contributed to the majority of time spent in the memory system. However, the figure shows how VH_A and VH_B significantly reduced the cycles spent on sharing misses serviced by remote L1 and L2 caches. VH_A averaged 45 cycles for such a miss in an Apache workload compared with 102 cycles for TAG-DIR.

Figure 9 shows the normalized cycles per transaction (CPT) of each VM running the mixed configuration. We compared CPT within VMs because the units of work differ between workload type and VM size. VH_B offered the best overall performance by showing the lowest CPT for the majority of VMs. STATIC-BANK-DIR slightly out-

←

Figure 6. An intra-VM sharing miss in our baseline directory protocols: directory in DRAM (DRAM-DIR) (a), an interleaved directory based on memory address (STATIC-BANK-DIR) (b), or a directory implemented as a duplicate tag store (TAG-DIR) (c).

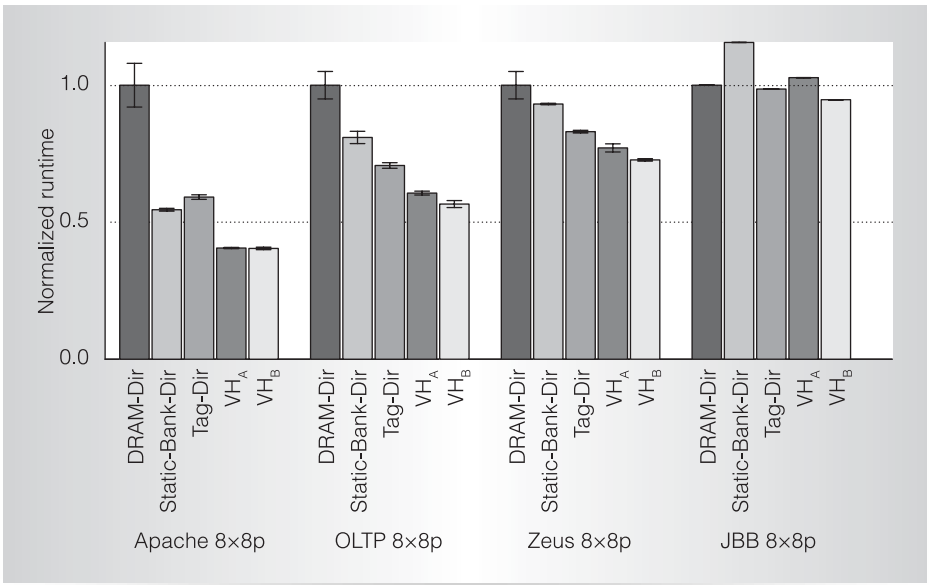


Figure 7. Normalized runtime for homogenous consolidation.

performed VH_B for the OLTP VMs in the mixed configuration. This is because the OLTP working set is large and the STATIC-BANK-DIR protocol allows one VM to utilize the cache resources of other VMs. However, where STATIC-BANK-DIR slightly improved the performance of the OLTP VMs, it made the JBB VMs perform more poorly because of the interference. On the other hand, VH_B isolated the cache resources between VMs, thereby offering good overall performance.

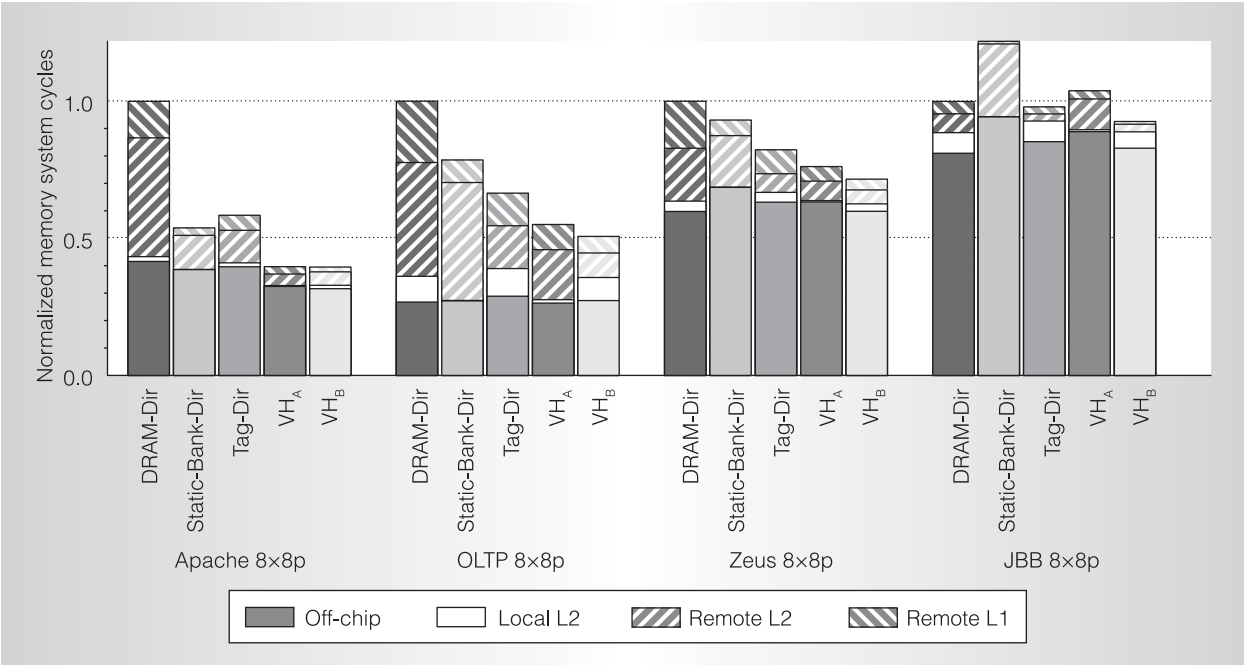


Figure 8. Normalized memory stall cycles, by type, for homogenous consolidation.

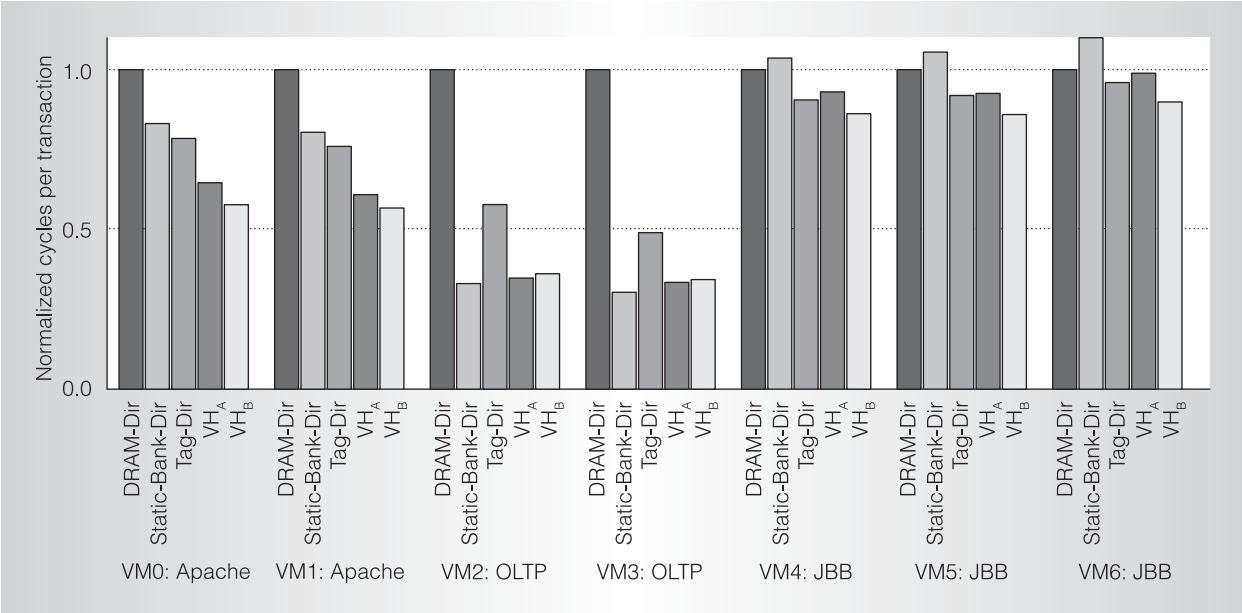


Figure 9. Normalized cycles per transaction (CPT) for each virtual machine running the mixed configuration.

Virtual hierarchies offer a new framework for designing the vast memory hierarchy of many-core CMPs. A virtual hierarchy adapts to space-shared workloads like multiprogramming and server consolidation. Although our case study focused on consolidated server workloads in a tiled architecture, our virtual hierarchy idea can apply to other architectures and use cases. For example, our two-level virtual hierarchy readily applies to nonuniform cache arrays (NUCA) not interleaved with cores (a dance-hall organization). Our implementation can also optimize for space-shared multiprogrammed workloads in a single-OS environment with little change to system software. Extensions to our implementation include fully automatic hierarchy configuration and better mechanisms for implementing fairness and quality-of-service policies. MICRO

Acknowledgments

We thank Philip Wells for providing assistance with simulating consolidated workloads. We also thank Dan Gibson, Milo Martin, Kathleen Marty, Kathryn Minnick, Kevin Moore, Benjamin Serebrin, Andy Phelps, Karin Strauss, Yasuko Watanabe, Min Xu, the Wisconsin Multifacet group, and the Wisconsin Computer Architecture Affiliates for their comments and proofreading. Finally, we thank the Wisconsin Condor project, the UW CSL, and Virtutech for their assistance. This work is supported in part by the National Science Foundation, with grants EIA/CNS-0205286, CCR-0324878, and CNS-0551401, and by donations from Intel and Sun Microsystems. Hill has significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Intel, or Sun Microsystems.

References

1. C.A. Waldspurger, ''Memory Resource Management in VMware ESX Server,'' *Proc. 5th Symp. Operating Systems Design and Implementation* (OSDI 02), ACM Press, 2002, pp. 181-194.
2. E. Hagersten and M. Koster, ''WildFire: A Scalable Path for SMPs,'' *Proc. 5th IEEE Symp. High-Performance Computer Architecture* (HPCA 99), IEEE CS Press, 1999, pp. 172-181.
3. M.M.K. Martin, M.D. Hill, and D.A. Wood, ''Token Coherence: Decoupling Performance and Correctness,'' *Proc. 30th Ann. Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 182-193.

4. A. Gupta and W.-D. Weber, ''Cache Invalidation Patterns in Shared-Memory Multiprocessors,'' *IEEE Trans. Computers*, vol. 41, no, 7, Jul. 1992, pp. 794-810.
5. M.R. Marty and M.D. Hill, ''Coherence Ordering for Ring-Based Chip Multiprocessors,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 06), IEEE CS Press, 2006, pp. 309-320.
6. M.R. Marty et al., ''Improving Multiple-CMP Systems Using Token Coherence,'' *Proc. 11th IEEE Symp. High-Performance Computer Architecture* (HPCA 05), IEEE CS Press, 2005, pp. 328-339.
7. M.R. Marty et al., ''Virtual Hierarchies to Support Server Consolidation,'' *Proc. 34th Ann. Int'l Symp. Computer Architecture* (ISCA 07), IEEE CS Press, 2007, pp. 46-56.
8. M.M. Martin et al., ''Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset,'' *Computer Architecture News*, Sept. 2005, pp. 92-99.
9. K.J. Nesbit et al., ''Fair Queuing CMP Memory Systems,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 06), IEEE CS Press, 2006, pp. 208-222.
10. A.R. Alameldeen and D.A. Wood, ''IPC Considered Harmful for Multiprocessor Workloads,'' *IEEE Micro*, vol. 26, no. 4, Jul./Aug. 2006, pp. 8-17.
11. A.R. Alameldeen and D.A. Wood, ''Variability in Architectural Simulations of Multi-threaded Workloads,'' *Proc. 9th IEEE Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 7-18.
12. C. Kim, D. Burger, and S.W. Keckler, ''An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 211-222.
13. M. Zhang and K. Asanovic, ''Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 336-345.
14. L.A. Barroso et al., ''Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing,'' *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 282-293.
15. J. Chang and G.S. Sohi, ''Cooperative Caching for Chip Multiprocessors,'' *Proc. 33rd Ann. Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 264-276.
16. J. Huh et al., ''A NUCA Substrate for Flexible CMP Cache Sharing,'' *Proc. 19th Int'l Conf. Supercomputing* (ICS 05), ACM Press, 2005, pp. 31-40.

Michael R. Marty is a software engineer at Google. His research focuses on the memory systems of multicore processors. He obtained his PhD in computer sciences from the University of Wisconsin–Madison.

Mark D. Hill is a professor in the Computer Sciences and the Electrical and Computer Engineering Departments at the University of Wisconsin–Madison. His research interests include parallel computer system design, memory system design, and computer simulation. He earned a PhD in computer science from the University of California, Berkeley. He is an IEEE Fellow and an ACM Fellow.

Direct questions and comments about this article to Michael Marty, 1210 W. Dayton St., Madison, WI 53706; mikem @cs.wisc.edu.