Ch'i: Scaling Microkernel Capabilities in Cache-Incoherent Systems

Yuxin Ren
The George Washington University
Washington, DC, USA
ryx@gwmail.gwu.edu

Gabriel Parmer
The George Washington University
Washington, DC, USA
gparmer@gwu.edu

Dejan Milojicic
Hewlett Packard Labs
Palo Alto, CA, USA
dejan.milojicic@hpe.com

Abstract—**Hardware cache coherence limits the scalability of shared-memory multicore and multi-processors. Recently, there has been an increasing shift towards cache-incoherent architectures in many computing environments. Supercomputers can benefit from shared memory, yet must avoid the price of coherency across the system. To support co-locating multiple applications on cores, processors, and the overall system, an Operating System must manage the distributed memory resources. In this context, incoherence poses a significant challenge for OS that must manage memory access permissions across the system without compromising the performance of software.**

In this paper, we introduce the Ch'i microkernel that leverages incoherent shared memory using quiescence-based techniques to bound the extent of non-coherency. Ch'i maintains type and context consistency of its core data structures: capabilities. This enables a uniform, capability-based security model to manage system-wide memory. Using this approach, we demonstrate data-structure consistency in software in support of high-throughput, low-latency applications.

I. Introduction

Compute capacity improvements are achieved through integrating more cores and accelerators into the system, and the explosion of data volumes requires a massive amount of fast, accessible memory. The increasing compute and memory capacity lead to more heterogeneous and distributed architectures, which require greater scalability and energy efficiency. This trend drives the emergence of cache-incoherent architectures in many computing environments; from many-core chips [1], [2], [3], accelerators [4], [5], [6], rack-scale machines [7], [8], [9], RDMA systems [10], [11], [12] and data-center memory disaggregation [13], [14], [15]. These systems provide a global pool of memory which is accessible from any core in the system. Cores have private caches, but lack hardware support for system-wide cache coherency. While on-chip cache coherence may not be fully eliminated, future hardware architectures are more likely to be composed of multiple incoherent nodes, where cache coherence is provided only within a node.

Getting rid of global coherent cache greatly simplifies architecture design, provides higher energy efficiency, and enables more parallelism. However, this complicates the inter-core collaboration, due to the need for explicit software support for synchronization over shared memory. This synchronization often involves the intentional write-back or invalidation of cache-lines to synchronize cache contents with memory. Cache-incoherence is particularly challenging for operating systems as data-structures for tracking globally accessible resources can be complex. Despite the complexity of cache-incoherent memory, operating systems are necessary to provide system-wide, uniform access control. This is essential to isolate clients, contain errors within fault domains, and manage resources such as the overall memory pool. To fully utilize the global pool of shared memory for low-latency, high-throughput processing, new systems and techniques are required to orchestrate the data sharing and movement between cores. We propose that new systems should satisfy the following requirements.

- **Low latency**. CPU caches must be leveraged to enable cache-speed access to global memory, thus avoiding memory latency.
- **High scalability**. All global memory should be directly accessed, thus enabling systems and applications to scale with the amount of global memory.
- **Compatibility**. The convenience of shared memory programming should be preserved to the maximum extent possible, thus allowing conventional techniques and mechanisms to be adapted easily.

However, existing shared memory frameworks and distributed operating systems fail to achieve above requirements. A simple approach is to explicitly insert cache write-back and invalidation operations abound shared memory accesses. Unfortunately, as studied in [16], frequent cache manipulation operations trigger non-trivial overhead. Even worse, cache invalidation results in memory-level latencies. On the other hand, several alternative kernel designs treat the hardware as a distributed system and eschew memory sharing among incoherent nodes. For instance, some systems [17], [18], [19] use message passing to coordinate between nodes. Popcorn [20], K2 [21] and Kerrighed [22] use replicated kernels and migrate threads between incoherent nodes. Barrelfish [23] employs a form of two-phase commit distributed consensus to synchronize capability management. However, these distributed approaches sacrifice performance and limit the scalability due to expensive message passing. Other systems aimed at hardware resource disaggregation restrict functionality and usability (*e.g.* LegoOS [15] does not support writable shared memory across nodes).

To achieve an efficient shared memory kernel, we introduce the Ch'i microkernel that applies quiescence-based techniques – commonly used to support scalable memory revocation (*e.g.* RCU) [24] – to manage cache coherency. This approach was introduced in the context shared memory data-structures [16], and this paper leverages and extends it in a co-design with an OS. Ch'i is designed to enable common-case kernel paths to run at full speed using cached memory without cache flushing, and to fully utilize shared memory without message passing. Ch'i does this by leveraging a key observation: that the access to *stale cache contents* can be controlled in a manner similar to *stale references to data-structures* in parallel systems. It draws on work in deferred reclamation and quiescence techniques in kernels [25], [24] to accelerate the read-only fast-paths and synchronize modifications in the kernel. In a nutshell, quiescence-based synchronization tracks possible references to resources within parallel reader cores, and defers resource reuse to a safe point when no references can exist. Ch'i leverages quiescence techniques to track cache staleness instead, and flushes accessed cache-lines to drop references in the stale cache. Caches are flushed periodically, which amortizes the cost of cache flush instructions and bounds the extent of stale cache-lines appearing in any cache due to incoherency.

All Ch'i data-structures use the shared memory pool, thus access to ranges of memory is determined uniformly across all nodes in the system. Ch'i is based on the Composite OS [25] and is centered around a capability-based [26] access control model of system resources. Ensuring that capability tables and page-tables leverage shared memory and efficient software coherence, a resource manager applies a set of system-wide, uniform access-control mechanisms. This capability-based isolation can be used to strongly partition the system – nodes and memory – among clients. Isolated managers can mediate and manage resources, with a global view on system resource demand despite non-coherent caches. Furthermore, resource managers execute on *all nodes* as they also maintain our software coherency, thus they can be directly invoked on each node. This avoids distributed coordination (RPC between nodes, consensus, etc...), and leverages efficient, low-latency inter-process communication (IPC).

The contributions of this paper include:

- the design and implementation of the Ch'i microkernel that uses quiescence to effectively coordinate the safe and efficient sharing of kernel capabilities across incoherent nodes,
- the evaluation of this system with a simple key-value (`memcached`) service.

This paper is organized as follows. §II introduces the background of incoherent cache architecture, quiescence-based technique and capability-based systems. §III and §IV present the design and implementation of Ch'i. §V evaluates Ch'i using microbenchmarks and `memcached` service. §VI, §VII and §VIII discusses related and future work and concludes.

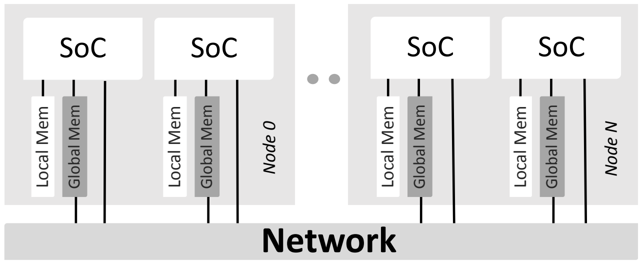

Fig. 1. Cache-incoherent system with many nodes connected to global non-coherent memory and local non-shared memory. Each node is a cache coherency domain.

II. Background

A. Incoherent Cache Architecture

Figure 1 presents a typical incoherent cache architecture. Every core has a private memory, which is not accessible from other cores. A global memory is attached to and shared by all the cores. A small number of cores are grouped into a node. Hardware cache coherency is provided only within a node. Without inter-node cache coherence, software has to orchestrate the data sharing among nodes.

To explicitly force coherency, Ch'i assumes instructions to invalidate and write-back specific cache-lines are available. Additionally, we assume the integration of atomic instructions into the memory fabric. These assumptions are, for instance, validated by GRU [27] and the industry-standard libfabric [9], [28].

B. Quiescence-based Technique

Quiescence-based techniques [29], [30], [31], especially RCU [24] have been widely used inside Linux kernel. These techniques enable read-only operations to execute in parallel with update operations, and require no explicit synchronization on the read path. On the other hand, update operations make a private copy of the data structure portion being modified. Once the private copy has the intended changes, the writer atomically replaces the old data structure portion with the private copy. To prevent interference on parallel readers, the old data structure portion cannot be reclaimed until all ongoing readers finish their operations. The deferred reclamation complicates the semantics of read operations, since readers are possible to get old data structure objects while new ones are already present in the data structure. Such stale access inspires the key observation of Ch'i: staleness introduced by incoherent-cache can be controlled and managed by leveraging quiescence-based techniques.

C. Capability-based Management

Capability-based access control [26], [32] is widely used in modern microkernels [25], [33], [34], [35]. Harnessing and modifying hardware and kernel resources are required to provide a capability that designates access rights to the resource. Table I summarizes the common resources managed by capabilities. While we use Composite [25] as a reference, the usage of capabilities is similar to other microkernels.

TABLE I
COMMON RESOURCE TYPES IN MICROKERNELS

Execution Resources[a]	
Threads	Execution context and invocation stack that tracks a sequential flow of control
Sync Invocation EP[b]	Call an associated component with an IPC
Receive EP	Block a thread waiting for an event (for example, an interrupt or send)
Async Send EP	Send an event to activate a receiver
Temporal capability	Capabilities that provide access control of the execution time
Memory Resources	
Frame	Untyped physical memory
Virtual Memory	User-level accessible memory
Access Control Resources	
Capability Table Chunk (lvl)	A chunk at a specific level in a capability table radix trie
Page Table Chunk (lvl)	A chunk at a specific level in a page table
Component	An aggregate of capability and page tables

[a]Examples are taken from Composite [25]
[b]EP stands for endpoint

Capabilities are tracked in capability tables, which are often organized in tree-based data-structures [34], [36]. The ability to *modify* capability tables is protected in a manner consistent with the rest of the kernel resources: only a capability to a capability table enables operations to modify its contents.

Capability management. Every system call will result in at least a capability table lookup to validate a proper capability resides in the capability table. Some system calls require modification to a capability table, such as capability table construction with assembling different levels into the tree (*cons* and *decons*), capability creation (*activate*), revocation (*deactivate*) and delegation (*copy*). Thus, providing a shared kernel capability management infrastructure is challenging but critical to efficiently leverage microkernels in incoherent cache architectures.

Memory management. Another important type of capability table is the page table, which tracks memory resources and mappings in the system. While kernel memory management is moved to user-level pioneered by seL4 [34], the kernel has to guarantee *type safety*, which is that a single memory frame cannot be accessed concurrently as two different types. For example, a frame cannot be both used as a kernel thread structure, *and* as a virtual memory mapping in a user-level application at the same time. Hence, memory retyping, map, and unmap operations provided by the kernel are core operations to provide a mapping of a frame of memory to any set of nodes, thus must be scalable and fast.

D. Challenges with Incoherent Cache

Operations	Node 0	Node 1
activate	$*A = C$	
deactivate	$*A = NULL$	
lookup		$*A \rightarrow C$
activate	$*B = C$	
lookup		$*A \rightarrow C$

This figure shows a sequence of capability operations performed on different nodes, demonstrating the possible issues caused by incoherent cache, even if a resource's memory is never modified. First, node 0 *activates* a resource – which converts a frame of memory into a kernel resource such as a capability table, C. At this point, node 1 might access that capability table, bringing its memory into cache. When node 0 then *deactivates* A (setting it to $NULL$), this deactivation is not guaranteed to be visible on node 1 which can still *lookup* C in its cache. At this point, the incoherent caches can break the access control policies guaranteed by traditional capability mechanism. For instance, if the previously deactivated resource (*e.g.* a memory frame) is reallocated (*activated* again), but as a *different type of resource* (*e.g.* a thread), this will lead to the erroneous situation where Node 0 is using the memory as a thread, and Node 1 is treating it as a capability table. The core problem is that resources that can be accessed in other node's caches should not be reallocated.

To summarise, modifications to kernel data-structures, especially on capability tables can lead to stale cache-lines, resource access failures, and capability inconsistencies. Thus it is non-trivial to adapt microkernels to cache-incoherent architectures. This paper introduces Ch'i microkernel, the first OS we know of to execute on non-coherent, shared memory. It utilizes quiescence-base techniques to avoid expensive cache manipulation operations on the read-path. It utilizes deferred reclamation which is reimagined in the incoherent memory and OS context to avoid the problem outlined above. Kernel data-structures are immutable except via synchronized atomic instructions paired with the quiescence mechanism.

III. DESIGN

The design of Ch'i allows the conventional single-image system abstractions and enables its most common kernel operations to be performed with no explicit cache operations, despite kernel data-structures residency in non-coherent memory.

A. Design Goals

Cache incoherency complicates the consistency guarantees for capability and kernel resource modifications. However, the kernel must hide such complexity efficiently, thus an application or system service will not be limited by any kernel compatibility, security and performance issues. Specifically, Ch'i focuses on following goals:

G1 *Minimize extra complexities brought into the kernel.* Minimality is the key principle of a microkernel. Thus any mechanisms dealing with incoherent cache should not bloat kernel complexity and its trusted computing base.

G2 *Provide a uniform, global view of system resource to user-level services and applications.* To efficiently manage and use all available system resources, it is essential for user-level services and applications to get consistent and global information from the kernel.

G3 *Predictable kernel performance.* All kernel cost, such as operation latency, memory consumption and cache staleness, introduced by reasoning about incoherent cache must be controlled in a predictable way, and should not blow up with the number of incoherent nodes or the amount of global memory.

G4 *Aggressive optimization of common kernel operations.* The kernel itself should not become the bottleneck to limit application performance. Thus all its common operations need to be heavily optimized, such as avoiding memory access latencies, cache flush instructions, and message passing cost.

B. Delayed Cache Coherency via Quiescence

Ch'i makes the observation that *stale cache-lines* in incoherent cache can be treated similarly to the parallel *references* that are implicitly tracked by quiescence techniques. Both are references to stale objects that have since been made *unreachable* within the data-structure. Ch'i eliminates capability and resource inconsistency by guaranteeing that they are not reused while any stale cache contents exist in any node (**G2**). Thus, cache incoherency is tolerated, but Ch'i guarantees cache-coherency only when quiescence is achieved, which leads to a notion of delayed cache coherency.

Delayed cache coherency complicates the semantics of capability and resource access in Ch'i. Specifically, accesses to kernel resource are allowed windows of inconsistency until stale cache-lines are updated by software. Thus, it introduces accesses to stale resources. For example, a node with a stale cache-line referencing a capability table, can harness a capability to access a resource even after that capability has been revoked on another node. The same problem already exists in current operating systems [25], [24], but it results from parallel reference. Similar to existing systems, the correctness is guaranteed as long as (1) a revoked resource is not reused while it is still possible for other nodes to access it, and (2) the time window of stale access is bounded (**G3**).

Achieving cache quiescence. Ch'i achieves cache quiescence when all nodes have removed all stale cache-lines from their caches since the kernel data-structures were updated, and the kernel determines when *every* node has done so. Any capabilities or resources deactivated *before* that time may then be reused. To determine this ordering, Ch'i tracks the time when each node has finished flushing stale entries from their cache, and the time a capability is updated. Ch'i ensures that capabilities, and the kernel resources they reference are not reused until quiescence has been achieved when no stale cache-lines containing them exist. Please note, while outside the scope of this work, parallel references are also considered when trying to reuse capability tables or resource. This can be solved by employing existing mechanisms based on RCU [24] or bounded worst case kernel execution time [25].

User-level management of kernel cache coherence. To maintain kernel minimality, Ch'i does not actively execute complete cache flushes inside the kernel (**G1**). The kernel is non-preemptive, therefore doing so would be prohibitive. Instead, it delegates orchestrating the kernel cache flushes to user-level managers, who use kernel APIs to periodically flush kernel cache-lines to achieve kernel cache quiescence, thus coherency. At first glance, orchestrating the cache coherency of the kernel in user-level is a little scary – an inversion of control. However, Ch'i *guarantees kernel safety* as it always validates that quiescence has been achieved before allowing modifications to any kernel objects. Thus the integrity of kernel data-structures is independent of user-level orchestration of cache quiescence. In fact, the user-level management of kernel resources is widely accepted design philosophy of microkernels. Previous research use the user-level management of different kernel resources, such as schedulers [37], kernel memory [38], [34], interrupt management [39], context switches [40], and distributed capability delegation [23].

Kernel interface for cache quiescence. Ch'i exposes an interface to user-level managers so that they can trigger the periodic kernel cache flush. However, the kernel cannot export a simple system call to remove *all* stale kernel cache-lines, the non-preemptible kernel would suffer an unpredictable execution overhead. Instead, Ch'i adds a new cache flush system call, which flushes a constant number of stale kernel cache-lines (**G3**). Once this operation has been performed enough times to flush the region of incoherent memory, quiescence is achieved, and the kernel records that previously deactivated resources can be re-used. Similar to other resource management, the functionality of flushing kernel cache lines is accessed only through a privileged per-node *hardware capability*. §IV-A details how Ch'i identifies and flushes stale cache-lines efficiently.

C. Common Kernel Capability Operations

Read-only operations. Most common kernel capability operations are read-only, as they just perform capability table lookups to harness kernel resources. Synchronizing read access to kernel structures uses quiescence, avoiding heavyweight cache invalidation and synchronization operations, such as reference counts or locks (**G4**). The performance implications of this are significant: objects are loaded exclusively from CPU cache, which is usually an order of magnitude faster than memory. On the other hand, stale versions of objects may be retrieved. Thus, modifications are required to respect concurrent readers by delaying resource reclamation until quiescence is achieved.

Mutating operations. Instead of explicitly coordinating between nodes to perform mutating operations (*e.g.* using two-phase commit), capability table modifications are made directly, and quiescence is used to bound the scope of inter-node inconsistency. This complicates the efficient coordination between modifications. To cope with such difficulty, Ch'i designs capability modifications in the following ways. First, synchronization between multiple nodes (and cores on that node) is performed by architecture-provided atomic instructions (fetch and add, and compare and swap). Second, all updated objects are written back to memory immediately using cache write-back instructions. Last, any given kernel operation attempts to limit modifications to a single cache-line. In Ch'i, it reduces the number of explicit cache write-back operations and the atomic transactions they generated.

Capability activation. A capability is added to a capability table slot when activated. All capabilities fit into a single cache-line. When capabilities are activated, those larger than a word require multiple stores that are ideally written back to memory in a single transaction. Ch'i reserves a slot using an atomic instruction, populates the rest of the slot, and makes the capability accessible by setting the type.

Capability deactivation. When deactivating a capability, its type is reset to a quiescing value, and a representation of the time of deactivation is stored into the slot's header. Subsequent activations must assess if quiescence has been achieved based on the recorded time before the kernel allows the operation.

Memory retyping. A retype table (a special capability table) tracks the type and number of references to a physical page. When memory is retyped into a raw frame, it is ensured that no stale references exist in cache. References are modified using atomic instructions, thus are consistent between nodes. Kernel memory is added into a capability table and typed as a kernel data-structure by *activating* the memory. This operation must also await quiescence both of the memory, and of the capability table slot it is being activated in to ensure that stale references are dropped. The same reasoning is applied to add user memory into its page table.

D. Generality and extensibility of Ch'i

Ch'i benefits from the scalable quiescence-based techniques. The integration of quiescence into the system harnesses the read-only fast-paths, the simple capability table data-structures (multi-level trie), and the carefully designed atomic modifications. While Ch'i is implemented as a microkernel, many other kernels such as Linux heavily rely on RCU, thus are possible to adapt to Ch'i's mechanisms to address incoherent cache. Further, the inter-node capability system of Ch'i paired with its single-image enable user-level components to define their own consistency models and management policies, using any necessary cache operations and global system view (**G2**).

IV. Implementation

The Ch'i prototype is implemented based on Composite [25], a scalable, real-time microkernel. The lock-free nature and the real-time execution time of the non-preemptive Composite kernel ease the synchronization and parallel reference tracking in Ch'i. Thus, we are able to focus on adding the requisite cache operations and support for cache quiescence.

A. Cache Quiescence

User-level managers are responsible for periodic kernel cache flush and quiescence. Analogous to different granularities of timer ticks in conventional systems, choosing the frequency of cache quiescence represents a system trade-off between performance and staleness. As no one frequency is best for all systems, Ch'i implements the policy in user-level, allowing its redefinition on a per-system basis.

Flushing accessed kernel cache-lines. Cache flushes are aggregated across many kernel operations by batching and pipelining cache flush instructions. To reduce cache flush cost, Ch'i limits the scope of the flushes from all cache-lines holding kernel data-structures (potentially 100s of GiB) to those cache-lines that are *actually accessed*, thus are *possibly* stale.

The kernel must identify and track which cache-lines have been accessed, thus need to be flushed. Toward this, Ch'i uses the *accessed* bits in the page-table entries for the kernel virtual memory to determine which virtual pages have been accessed and may have stale cache-lines. Thus, the corresponding flushes are accurate to a page. Each node tracks accesses separately as a set of page-table entries for the cache-incoherent memory by maintaining a separate set of kernel page-tables per-node.

This approach has a number of complications. First, page-table walks (on TLB miss) have the impact of causing *accesses* (loads) on page-table nodes. However, on x86, these accesses do *not* set the accessed bit corresponding to that page-table node (as the access is not through the kernel's virtual memory mappings). Thus, the accesses to the page-table nodes by the hardware page-table walker are not tracked, by default. Ch'i handles this case by maintaining which pages are used as page-tables in the retype table, and we make the pessimistic assumption that they all need flushing. Second, we find that as long as a page's translation is in the TLB, an unset accessed bit will never be set (presumably delaying setting the bit to a TLB eviction or flush). This is particularly relevant for kernel pages as they are not flushed from the TLB upon a context switch (they are marked as *global*). Ch'i handles this by explicitly flushing global bits (by deactivating them in `cr4`, flushing, then reactivating them) at the beginning of each flush period. This issue is not documented in Intel's architecture manuals, so behavior may change in the future. Regardless, Ch'i's techniques are conservative and won't cause incorrect behavior if the architecture changes, however there may be opportunities for performance improvement.

The Ch'i kernel never accesses user-level virtual memory, so only kernel cache-lines are flushed by the kernel. User-level applications need to define their own logic for coherency (see [16]), use unshared memory, or depend on other components that abstract those details. Additionally, core-local resources (*e.g.* threads) will only ever be accessed on their node, also decreasing the number of cache-lines that could possibly require flushing.

Using kernel cache flush interface. Ch'i adds a new system call (via capability) to flush the cache of a constant number of kernel memory pages. The system call returns a value to indicate if the entire kernel region has been flushed or not. Once this operation has been performed enough times to flush the region of incoherent kernel memory, quiescence is signaled by publishing the time of the last finished operation. As the all cache flush are local to the calling core, no synchronization is needed during its execution.

B. Capability-table Bootstrapping

The boot sequence of a node starts with the creation of capability structures in local memory that reference only resources in that memory. Next, a super-block-like structure in the global cache-incoherent memory is referenced that contains the root capability tables and page-tables for each node. All resources in the cache-incoherent memory are rooted in these capability tables (*i.e.* guaranteed to be managed/accessible through at least one node's capability tables).

When the system is first booted, a single node has capabilities not only to all of the cache-incoherent memory, but also nested references to the other node's capability tables. Most nodes, then, don't have access to global memory until they are explicitly delegated that access by the boot node. This enables the access control policies to be rooted in the early boot process, and to be controlled by the small trusted computing base of the kernel and initial component.

As kernel memory page tables are in local memory, every node has a separate page table and the associated kernel's address spaces. However, Ch'i assumes that global memory is mapped at the same virtual addresses in each kernel's address spaces. This is easy to implement in Ch'i due to its uniform, single-image design. The current prototype uses KVM/Qemu virtualization support to expose a PCI device that is backed by a shared, memory-mapped region on the host. Multiple nodes are brought up in Ch'i by booting multiple VMs, each sharing the same virtual device.

C. Kernel Capability Operations

Read-path lazy coherency. Each capability table and page table in Ch'i is implemented as a radix trie. Looking up a slot can find stale cache-lines in trie-internal pages and in the slots. This introduces extra complexity due to the hardware page walker. Stale page table entries result in a page-fault. Ch'i updates the page-fault handler to perform the cache-line flushes, and retry the faulting instruction. Cache quiescence ensures that capability table slots cannot be reused until stale mappings no longer exist.

Resource retyping. To maintain type safety, each frame is treated globally as exactly one of the three types (user-level memory, physical frame, or kernel memory). A retype table contains entries for each physical frame within the incoherent memory. The entries include the type of the frame, and a count of the number of mappings if it is virtual memory. For simplicity, Ch'i currently only keeps a single table shared across all nodes. When memory is retyped from or back to physical frame, its type in the entry is atomically modified using atomic instructions (`cas`). The reference count (due to *map/unmap*) is also updated by atomic operations.

Reference and TLB quiescence. Kernel capability slots and resources must not be reused while referenced by another node. The most difficult resource to reasoning about is virtual memory mappings, which must also consider references in TLB. To retype memory that was previously user-accessible to a frame, Ch'i must ensure no references to it exist in page tables, stale cache and TLB. TLB achieves quiescence after every core flushes its TLB during a context switch.

V. Evaluation

All experiments are run on Intel x86 i5-6400 processor running at 2.70GHz with a 6 MB last-level cache and 4 cores. All Linux experiments use Ubuntu 14.04, kernel 4.2.0-27. We execute Ch'i using QEMU with KVM and hardware virtualization extensions enabled.

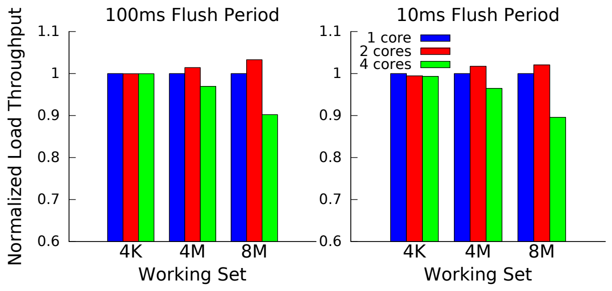

Fig. 2. The normalized load throughput with cache flush

A. Accuracy of Experimental Setup

QEMU can expose a memory mapped file as a PCI device to the VM which can directly access the memory through an aperture in physical memory. A system with shared memory between nodes is emulated by using the same file to back the global memory of multiple VMs. This section assesses the fidelity of such emulation.

Cache operations on modification. To effectively emulate non-coherent memory, all modifications to shared structures are prefixed with cache-line invalidations, and postfixed with a write-back and invalidation. This forces those modifications to cause accurate cache operation overheads and memory latencies.

Impact of invalidations with coherency. Different VMs will share the global memory, thus will express both the benefits and overheads of cache-coherency. Invalidating cache-lines on one core will invalidate those in another core. On the other hand, a load instruction on a core might "pre-fetch" a cache-line into the shared cache for another core, possibly *increasing* performance.

To understand the impact of these competing behaviors, Figure 2 depicts the load throughput normalized to a single core's performance, under an increasing working set in the presence of quiescence-based invalidations. We see that on two cores, there is a slight performance *benefit* (around 3%). On four cores, we find that for large working sets the cross-core impact of invalidations is the dominant factor (10% overhead). Also, an increasing number of cores leads to increasing inaccuracy due to the cache-coherency of our system. Motivated by these factors, we perform our Ch'i experiments with 4 cores so as to *pessimistically* (on our infrastructure) assess the system's performance.

Impact of shared cache. An additional impact of our test environment is that the last-level shared cache is contested by the multiple VMs. We ensure that all comparison cases use the same workload, thus are equally impacted by the reduced, and contested shared cache. Though this doesn't emulate a system without shared caches, our intent is to equally impact all compared systems.

Testbed scale. Our testing machine has only 4 cores. While small core count does not depict popular modern systems, higher core count will amplify the inaccuracy mentioned above, making it harder to justify the experiment results. On the other hand, all read operations in Ch'i do not involve

TABLE II
KERNEL OPERATION BENCHMARK

Operation	Local Memory	Global Memory	Linux
Round-trip IPC	*1392* **1405** [a] *725* **732** [b]	*1393* **1404**	*8955* **9026**
Thread dispatch	*260* **266**	*265* **269**	*2148* **2217**
async snd + recv	*586* **591**	*599* **605**	
Memory Map	*341* **393**	*820* **1540**	*831* **883**
Memory Unmap	*450* **456**	*1454* **2437**	*2692* **2783**
Cap Activation	*400* **410**	*916* **1530**	
Cap Deactivation	*373* **381**	*1054* **1705**	

[a]All numbers are cycles and are presented as *Average Cost* **99th Percentile Cost**
[b]This result is measured on bare-metal.

cache operations, and the cache operations involved in other kernel operations are local to the cache-coherent domain. Thus their overhead are independent of core count and do not prevent Ch'i from scaling to much larger machines. The only metric increasing with the core count is the amount of time to achieve cache quiescence. This is because our current prototype requires all the cores to flush stale kernel cache lines (§III-B). Optimization of cache quiescence detection and evaluating on larger machines are left as future work (§VII). In short, we believe our experiments focusing on a small number of cores fairly evaluate the current prototype.

B. Microbenchmarks

Delayed cache coherency enables efficient kernel capability operations without cache invalidations. This section evaluates the costs of various kernel operations especially with respect to the overheads introduced by explicit cache operations. All operations are executed 10 million times, and we report both the average cost and 99th percentile cost.

Discussion. Table II shows the results on both local and global memory. We compare to similar operations in Linux. QEMU/KVM adds significant overhead on some operations, most notably round-trip IPC. To show this, we include the IPC cost run on bare-metal. For other operations, there is little difference between QEMU and bare-metal.

Because of delayed cache coherency, the incoherent global memory has negligible impact on read-only kernel operations. Their costs are similar to local memory. However for mutating kernel operations, such as capability table modifications, cache operations dominate the overhead. Even so, their performance is competitive with Linux.

C. Key-Value Server

To evaluate the use of Ch'i with real world applications, we use `memcached` [41]. In its simplest form, `memcached` provides `get` and `put` requests for a cache of key-value pairs stored in a concurrent hash table. As Ch'i does not directly provide a user-level mechanism to handle incoherent cache, we apply BI [16] to `memcached`. BI is a library which extends RCU to support cache-incoherent systems.

To compare against distributed approaches, we apply three techniques to `memcached`. (1) `2PC` – all data is replicated across servers. Any server can handle any `get` request locally.

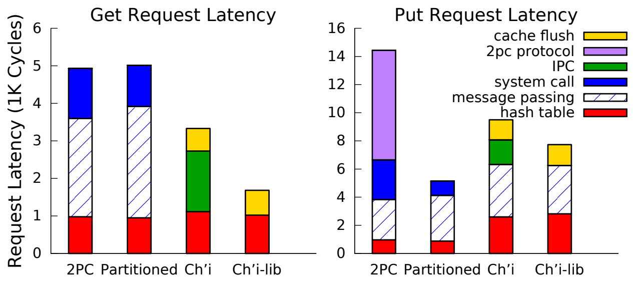

Fig. 3. `memcached` request latency (smaller is better).

`put` requests update all replicas using the Two Phase Commit (2PC) protocol. Barrelfish [23] uses this mechanism for kernel capability management. (2) `Partitioned` – data is partitioned across servers. Clients randomly choose a server and send both `get` and `put` requests to that server using global-memory-based RPC. A similar message passing approach is used in LegoOS [15] to coordinate incoherent nodes. (3) Ch'i – BI-based `memcached` on Ch'i. A client uses local IPC to invoke the `memcached` service, which processes `get` requests locally and sends `put` requests to the corresponding server node based on consistent hashing. A protection boundary between the client and `memcached` is not necessary if they trust each other. (4) `Ch'i-lib` – the client uses `memcached` as a library and it calls its functions without IPC. This shows the trade-off between performance and protection.

While these variants are not apple-to-apple comparison of `memcached` implementation, they represent typical approaches to handle incoherent-cache, which is the focus of this paper. Furthermore, due to different system models, BI-based `memcached` cannot be ported to other systems. For example, thanks to its uniform, single-image model, only Ch'i supports local invocation to global service, which enables `get` requests to be processed locally in BI-based `memcached`. Additionally, BI has its own performance trade-offs. As shown later, `put` requests in BI-based `memcached` are slow as BI requires more kernel operations. Thus performance improvements on Ch'i are not solely from BI.

Experiment set-up. We use two clients and two `memcached` servers, each of them is running on a dedicated core. Ch'i achieves periodic cache quiescence 30 times per second. To avoid QEMU negatively impacting other systems, we only run Ch'i within QEMU, while other experiments are executed directly on bare-metal with Linux. We use YCSB [42] to generate the trace, which contains 10 million requests with 16B keys and 32B values.

Latency Analysis. The different approaches make different trade-offs between read and write performance. To show this, we break down request latency for each. Figure 3's left graph shows the `get` request latency. `2PC` and `partitioned` have similar `get` latency, which are dominated by message passing costs. Ch'i has lower overheads as a result of local IPC. `Ch'i-lib` reduces the latency further as it avoids IPC costs. However, `put` latency behaves differently as shown in the graph on the right in Figure 3. `2PC` has the most overhead

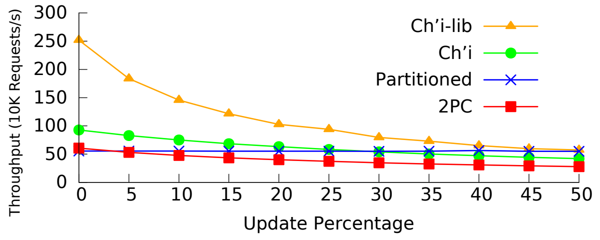

Fig. 4. `memcached` throughput.

because its consensus protocol requires exchanging multiple messages among all servers. Ch'i has more overhead than the `partitioned` approach, while it has less than 2PC. This is because there are two extra overheads on the update path. The first one comes from memory allocation which involves kernel capability table modification. The second one is the cost of invoking kernel cache quiescence.

Throughput Discussion. Figure 4 reports the `memcached` throughput with different update ratios. With low update percentages (that are common [41], [43], [44]), Ch'i has a high throughput due to cached read-path accesses. And removing IPC overhead in `Ch'i-lib` significantly improves throughput. When the update ratio increases, as expected, both Ch'i and 2PC degrade due to their high `put` overheads. `partitioned memcached` treats `get` and `put` equally, thus its throughput does not change with different `put` ratios.

VI. Related Work

Quiescence are commonly used in RCU and SMR techniques [29], [30], [31] Ch'i extends these techniques to determine if stale cache references can exist on any node. Quiescence based techniques are also utilized in shared-memory cache-coherent operating systems. There are over 6500 API calls in the Linux kernel using RCU [24]; The Composite kernel [25] uses a simple version of time-based quiescence to implement a kernel that has no lock-based synchronization, and minimizes and controls cache-line modification. However, none of these systems support cache-incoherent architectures.

§I discusses research about implementing operating systems on top of incoherent cache. Other recent research focus on handling incoherent cache within user-level services. Hare [45] is a file-system for cache-incoherent systems, and it relies on message passing to coordinate multiple nodes. libMPNode [46], an OpenMP runtime, uses distributed shared memory and thread migration to deal with incoherent cache. Atlas [47] delays cache flushes to the exit of a critical section to address incoherency for lock-based applications. BI [16] extends RCU to support cache-incoherent systems. LazyPIM [48] is a hardware mechanism exploring lazy coherence for processing-in-memory. Tavarageri et al. [49] present compiler extensions to support software cache coherence.

VII. Discussion and Future Work

There is much room for future research with cache-incoherent architectures and Ch'i. Current Ch'i prototype has a couple of limitations, and we aim to improve and evaluate it on larger scale testing environments.

Fine-grained kernel cache tracking. Currently Ch'i requires all accessed kernel memory to be flushed. This is pessimistic, as not all accessed memory has been modified, and not all of it will be accessed again in the future. There are lots of opportunities to track and identify stale kernel cache lines more accurately, thus reducing the amount of cache to be flushed. For example, Ch'i can employ some form of write logs to track modified kernel objects, and only flush them. Additionally, Ch'i can exploit structural properties of kernel capabilities, such as flushing only *reachable* capabilities by walking through capability tables.

System resource partitioning. In order to achieve cache quiescence, the current prototype awaits for all the nodes to flush kernel memory from cache. This implementation has two deficiencies. First, it limits the scalability of cache quiescence. The amount of time to achieve quiescence, in the worst case, will increase linearly with more nodes. Second, it weakens the isolation among nodes. If one node fails to flush its kernel's cache, it will delay or prevent other nodes from reclaiming kernel objects. To alleviate these problems, we can leverage Ch'i capability to partition the system resource across different nodes. For instance, we can delegate a subset of cores and global memory only to high-criticality tasks, and make sure no other tasks can access these resource via capability confinement. In this way, the kernel only needs to wait for the delegated cores to flush cache, instead of all the nodes, thus ensuring isolation and timely memory reuse for each client.

Security concerns. User-level management of kernel cache flush can introduce a potential denial-of-service attack. For example, if the manager is compromised, it can never flush kernel cache, which prevents reclaiming unused kernel object. If unreclaimed kernel objects get accumulated, they will eventually exhaust global memory. We see two solutions to this. First, the kernel could instead use the privileged instruction `wbinvd` to flush the entire CPU cache. The kernel could enforce cache quiescence by invoking `wbinvd` when available resource are running low. However, `wbinvd` is not a desirable general solution due to its huge overhead, and it flushes *all* cache-lines including those that are from local memory and applications. Second, the user-level manager for this is a very small amount of code (< 200 LoC), and when combined with separation kernel support (via the temporal access control of TCaps [50]), we believe it is reasonable to integrate the user-level quiescence manger into the system's TCB.

VIII. Conclusions

This paper has introduced the Ch'i system that enables cache-speed microkernel fast-paths on cache incoherent architectures, while using quiescence-based techniques to control the impact of incoherent cache to avoid data-structure inconsistencies in kernel. A single capability-based access-control mechanism is used to manage shared incoherent memory. Results show significant performance improvements over conventional distributed techniques. We believe this marks a significant step toward enabling OS management and control over future scalable cache-incoherent architectures.

REFERENCES

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2010, pp. 108–109.

[2] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE micro*, vol. 26, no. 2, pp. 10–24, 2006.

[3] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase *et al.*, "Runnemede: An architecture for ubiquitous high-performance computing," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 198–209.

[4] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: a hybrid memory model for accelerators," in *Proceedings of the 37th annual international symposium on Computer architecture (ISCA'10)*, 2010, pp. 429–440.

[5] D. Johnson, M. Johnson, J. Kelm, W. Tuohy, S. Lumetta, and S. Patel, "Rigel: A 1,024-core single-chip accelerator architecture," *IEEE Micro*, vol. 31, no. 4, pp. 30–41, Jul. 2011.

[6] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 140–151, 2009.

[7] K. Asanovic, "FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, Santa Clara, CA, USA, February 2014.

[8] Intel Corporation, "Intel Rack Scale Design," Online, 2016, http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-architecture/intel-rack-scale-architecture-resources.html.

[9] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic, "Beyond Processor-centric Operating Systems," in *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause, Ittingen, Switzerland, May 18-20*, 2015.

[10] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast remote memory," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14), Seattle, WA, USA, April 2-4*, 2014.

[11] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: Distributed transactions with consistency, availability, and performance," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15), Monterey, CA, USA, October 4-7*, 2015.

[12] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "The case for rackout: Scalable data serving using rack-scale systems," in *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16), Santa Clara, CA, USA, October 5-7*, 2016.

[13] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 649–667.

[14] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont, "Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter," in *Proceedings of the Thirteenth EuroSys Conference (EuroSys'16)*, 2018, pp. 1–12.

[15] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: A disseminated, distributed OS for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[16] Y. Ren, G. Parmer, and D. Milojicic, "Bounded incoherence: A programming model for non-cache-coherent shared memory architectures," in *Proceedings of the Eleventh International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'20, 2020.

[17] S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe, "Early experience with the barrelfish os and the single-chip cloud computer." in *Proceedings of the 3rd Many-core Applications Research Community Symposium (MARC), Ettlingen, Germany, July 5-6*, 2011.

[18] S. Peter, J. Giceva, P. Shinde, G. Alonso, and T. Roscoe, "POSTER: OS design for non-cache-coherent systems," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11), Cascais, Portugal, October 23-26*, 2011.

[19] M. Hille, N. Asmussen, P. Bhatotia, and H. Härtig, "Semperos: A distributed capability system," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 709–722. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/hille

[20] A. Barbalace, B. Ravindran, and D. Katz, "Popcorn: a replicated-kernel os based on linux," in *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

[21] F. X. Lin, Z. Wang, and L. Zhong, "K2: A mobile operating system for heterogeneous coherence domains," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 14, 2014.

[22] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, D. Margery, J.-Y. Berthou, and I. D. Scherson, "Kerrighed and data parallelism: Cluster computing on single system image operating systems," in *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935)*. IEEE, 2004, pp. 277–286.

[23] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schpbach, and A. Singhania, "The Multikernel: A new OS architecture for scalable multicore systems," in *Symposium on Operating System Principles (SOSP)*, 2009.

[24] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole, "Rcu usage in the linux kernel: One decade later," *Technical report*, 2013.

[25] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[26] H. Levy, "Capability-based computer systems," 1984.

[27] M. Dreseler, T. Kissinger, T. Djürken, E. Lübke, M. Uflacker, D. Habich, H. Plattner, and W. Lehner, "Hardware-accelerated memory operations on large-scale numa systems." in *ADMS@ VLDB*, 2017, pp. 34–41.

[28] "Fabric Attached Memory Atomics libary: https://github.com/fabricattachedmemory/libfam-atomic."

[29] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, no. 12, 2007.

[30] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, 2012.

[31] Q. Wang, T. Stamler, and G. Parmer, "Parallel sections: Scaling system-level data-structures," in *Proceedings of the ACM EuroSys Conference*, 2016.

[32] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *Symposium on Operating Systems Principles*, 1999, pp. 170–185. [Online]. Available: citeseer.ist.psu.edu/shapiro99eros.html

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. Big Sky, MT, USA: ACM, Oct 2009.

[34] K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 133–150.

[35] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," in *In Proc. of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, December 2002.

[36] J. Liedtke, "On micro-kernel construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[37] J. Stoess, "Towards effective user-controlled scheduling for microkernel-based systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 59–68, 2007.

[38] A. Haeberlen and K. Elphinstone, "User-level management of kernel memory," in *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, sep 2003.

[39] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2008.

[40] P. K. Gadepalli, R. Pan, and G. Parmer, "Slite: Os support for near zero-cost, configurable scheduling," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 160–173.

[41] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 385–398. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[42] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[43] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12), London, United Kingdom, June 11-15*, 2012.

[44] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Characterizing facebook's memcached workload," *IEEE Internet Computing*, vol. 18, no. 2, 2014.

[45] C. Gruenwald, III, F. Sironi, M. F. Kaashoek, and N. Zeldovich, "Hare: A file system for non-cache-coherent multicores," in *Proceedings of the Tenth European Conference on Computer Systems (Eurosys '15)*, 2015.

[46] R. Lyerly, S.-H. Kim, and B. Ravindran, "libmpnode: An openmp runtime for parallel processing across incoherent domains," in *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'19, 2019.

[47] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, 2014.

[48] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "Lazypim: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters*, vol. 16, no. 1, 2017.

[49] S. Tavarageri, W. Kim, J. Torrellas, and P. Sadayappan, "Compiler support for software cache coherence," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, 2016.

[50] P. K. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, "Temporal capabilities: Access control for time," in *Proceedings of the 38th IEEE Real-Time Systems Symposium*, 2017.