Selective GPU Caches to Eliminate CPU–GPU HW Cache Coherence

Neha Agarwal†, David Nellans‡, Eiman Ebrahimi‡, Thomas F. Wenisch†,
John Danskin‡, Stephen W. Keckler‡

‡ NVIDIA and †University of Michigan
{dnellans,eebrahimi,jdanskin,skeckler}@nvidia.com, {nehaag,twenisch}@umich.edu

ABSTRACT

Cache coherence is ubiquitous in shared memory multiprocessors because it provides a simple, high performance memory abstraction to programmers. Recent work suggests extending hardware cache coherence between CPUs and GPUs to help support programming models with tightly coordinated sharing between CPU and GPU threads. However, implementing hardware cache coherence is particularly challenging in systems with discrete CPUs and GPUs that may not be produced by a single vendor. Instead, we propose, *selective caching*, wherein we disallow GPU caching of any memory that would require coherence updates to propagate between the CPU and GPU, thereby decoupling the GPU from vendor-specific CPU coherence protocols. We propose several architectural improvements to offset the performance penalty of selective caching: aggressive request coalescing, CPU-side coherent caching for GPU-uncacheable requests, and a CPU–GPU interconnect optimization to support variable-size transfers. Moreover, current GPU workloads access many read-only memory pages; we exploit this property to allow promiscuous GPU caching of these pages, relying on page-level protection, rather than hardware cache coherence, to ensure correctness. These optimizations bring a selective caching GPU implementation to within 93% of a hardware cache-coherent implementation without the need to integrate CPUs and GPUs under a single hardware coherence protocol.

1. INTRODUCTION

Technology trends indicate an increasing number of systems designed with CPUs, accelerators, and GPUs coupled via high-speed links. Such systems are likely to introduce unified shared CPU-GPU memory with shared page tables. In fact, some systems already feature such implementations [4]. Introducing globally visible shared memory improves programmer productivity by eliminating explicit copies and memory management overheads. Whereas this abstraction can be supported using software-only page-level protection mechanisms [26, 45], hardware cache coherence can improve performance by allowing concurrent, fine-grained access to memory by both CPU and GPU. If the CPU and GPU have separate physical memories, page migration may also be used to optimize page placement for latency or bandwidth by using both near and far memory [1, 13, 16, 42].

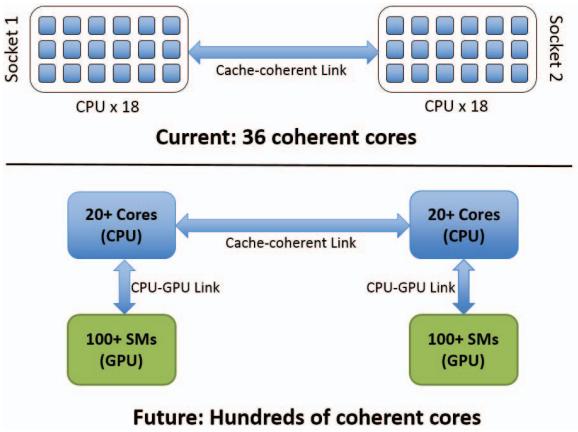

Figure 1: Number of coherent caches in future two socket CPU-only vs CPU–GPU systems.

Some CPU–GPU systems will be tightly integrated into a system on chip (SoC) making on-chip hardware coherence a natural fit, possibly even by sharing a portion of the on-chip cache hierarchy [15, 22, 26]. However, the largest GPU implementations consume nearly 8B transistors and have their own specialized memory systems [48]. Power and thermal constraints preclude single-die integration of such designs. Thus, many CPU–GPU systems are likely to have discrete CPUs and GPUs connected via dedicated off-chip interconnects like NVLINK (NVIDIA), CAPI (IBM), HT (AMD), and QPI (INTEL) or implemented as multi-chip modules [10, 29, 30, 31, 46]. The availability of these high speed off-chip interconnects has led both academic groups and vendors like NVIDIA to investigate how future GPUs may integrate into existing OS-controlled unified shared memory regimes used by CPUs [1, 2, 52, 53].

Current CPUs have up to 18 cores per socket [32] but GPUs are expected to have hundreds of streaming multiprocessors (SMs) each with its own cache(s) within the next few years. Hence, extending traditional hardware cache-coherency into a multi-chip CPU–GPU memory system re-

This research was supported in part by the United States Department of Energy. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

quires coherence messages to be exchanged not just within the GPU but over the CPU–GPU interconnect. Keeping these hundreds of caches coherent with a traditional HW coherence protocol, as shown in Figure 1, potentially requires large state and interconnect bandwidth [33,36]. Some recent proposals call for heterogeneous race-free (HRF) GPU programming models, which allow relaxed or scoped memory consistency to reduce the frequency or hide the latency of enforcing coherence [22]. Others argue that scopes substantially increase programmer burden, and instead propose a data-race-free programming model with coherence based on reader-initiated invalidation [59]. However, irrespective of memory ordering requirements, such approaches either require software to initiate flushes at synchronization points or system-wide hardware cache coherence mechanisms. We show that Selective Caching can achieve performance rivaling more complex CPU-GPU cache coherence protocols. Techniques like region coherence [54] seek to scale coherence protocols for heterogeneous systems, but require pervasive changes throughout the CPU and GPU memory systems. Such approaches also incur highly coordinated design and verification effort by both CPU and GPU vendors [24] that is challenging when multiple vendors wish to integrate existing CPU and GPU designs in a timely manner.

In the past, NVIDIA has investigated extending hardware cache-coherence mechanisms to multi-chip CPU–GPU memory systems. Due to the significant challenges associated with building such systems, in this work, we architect a GPU *selective caching* mechanism. This mechanism provides the conceptual simplicity of CPU–GPU hardware cache coherence and maintains a high level of GPU performance, but does not actually implement hardware cache coherence within the GPU, or between the CPU and GPU. In our proposed selective caching GPU, the GPU does not cache data that resides in CPU physical memory, nor does it cache data that resides in the GPU memory that is actively in-use by the CPU on-chip caches. This approach is orthogonal to the memory consistency model and leverages the latency-tolerant nature of GPU architectures combined with upcoming low-latency and high-bandwidth interconnects to enable the benefits of shared memory. To evaluate the performance of such a GPU, we measure ourselves against a theoretical hardware cache-coherent CPU–GPU system that, while high performance, is impractical to implement.

In this work, we make the following contributions:

1. We propose GPU selective caching, which enables a CPU–GPU system that provides a unified shared memory without requiring hardware cache-coherence protocols within the GPU or between CPU and GPU caches.

2. We identify that much of the improvement from GPU caches is due to coalescing memory accesses that are spatially contiguous. Leveraging aggressive request coalescing, GPUs can achieve much of the performance benefit of caching, without caches.

3. We propose a small on-die CPU cache to handle uncached GPU requests that are issued at sub-cache line granularity. This cache helps both shield the CPU memory system from the bandwidth hungry GPU and supports improved CPU–GPU interconnect efficiency by implementing variable-sized transfer granularity.

4. We demonstrate that a large fraction of GPU-accessed data is read-only. Allowing the GPU to cache this data and relying on page protection mechanisms rather than hardware coherence to ensure correctness closes the performance gap between a selective caching and hardware cache-coherent GPU for many applications.

Workload	L1 Hit Rate (%)	L2 Hit Rate (%)
backprop	62.4	70.0
bfs	19.6	58.6
btree	81.8	61.8
cns	47.0	55.2
comd	62.5	97.1
kmeans	5.6	29.5
minife	46.7	20.4
mummer	60.0	30.0
needle	7.0	55.7
pathfinder	42.4	23.0
srad_v1	46.9	25.9
xsbench	30.7	63.0
Arith Mean	44.4	51.6

Table 1: GPU L1 and L2 cache hit rates (average).

2. MOTIVATION AND BACKGROUND

Heterogeneous CPU–GPU systems have been widely adopted by the high performance computing community and are becoming increasingly common in other computing paradigms. High performance GPUs have developed into stand-alone PCIe-attached accelerators requiring explicit memory management by the programmer to control data transfers into the GPU's high-bandwidth locally attached memory. As GPUs have evolved, the onus of explicit memory management has been addressed by providing a unified shared memory address space between the GPU and CPU [26,45]. Whereas a single unified virtual address space improves programmer productivity, discrete GPU and CPU systems still have separate locally attached physical memories, optimized for bandwidth and latency respectively.

Managing the physical location of data, and guaranteeing that reads access the most up-to-date copies of data in a unified shared memory can be done through the use of page level migration and protection. Such mechanisms move data at the OS page granularity between physical memories [45]. With the advent of non-PCIe high-bandwidth, low-latency CPU–GPU interconnects, the possibility of performing cache-line, rather than OS-page-granularity, accesses becomes feasible. Without OS page protection mechanisms to support correctness guarantees, however, the responsibility of coherence has typically fallen on hardware cache-coherence implementations.

As programming models supporting transparent CPU–GPU sharing become more prevalent and sharing becomes more fine-grain and frequent, the performance gap between

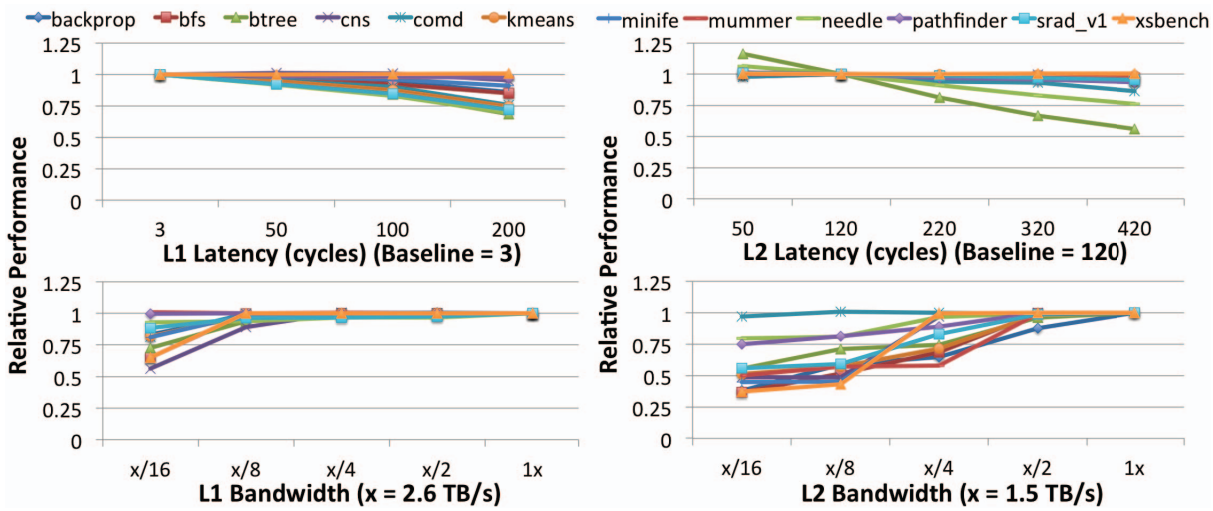

Figure 2: GPU performance sensitivity to L1 and L2 latency and bandwidth changes.

page-level coherence and fine-grained hardware cache-coherent access will grow [1,2,40]. On-chip caches, and thus HW cache coherence, are widely used in CPUs because they provide substantial memory bandwidth and latency improvements [41]. Building scalable, high-performance cache coherence requires a holistic system that strikes a balance between directory storage overhead, cache probe bandwidth, and application characteristics [8, 24, 33, 36, 54, 55, 58]. Although relaxed or scoped consistency models allow coherence operations to be re-ordered or deferred, hiding latency, they do not obviate the need for HW cache coherence. However, supporting a CPU-like HW coherence model in large GPUs, where many applications do not require coherence, is a tax on GPU designers. Similarly, requiring CPUs to relax or change their HW coherence implementations or implement instructions enabling software management of the cache hierarchy adds significant system complexity.

Prior work has shown that due to their many threaded design, GPUs are insensitive to off-package memory latency but very sensitive to off-chip memory bandwidth [1, 2]. Table 1 shows the L1 and L2 cache hit rates across a variety of workloads from the Rodinia and United States Department of Energy application suites [9,67]. These low hit rates cause GPUs to also be fairly insensitive to small changes in L1 and L2 cache latency and bandwidth, as shown in Figure 2. This lack of sensitivity raises the question whether GPUs need to uniformly employ on-chip caching of all off-chip memory to achieve good performance. If GPUs do not need or can selectively employ on-chip caching, then CPU–GPU systems can be built that present a unified, coherent shared-memory address space to the CPU, while not requiring a HW cache-coherence implementation within the GPU.

Avoiding hardware cache coherence benefits GPUs by decoupling them from the coherence protocol implemented within the CPU complex, enables simplified GPU designs, and improves compatibility across future systems. It also reduces the scaling load on the existing CPU coherence and directory structures by eliminating the potential addition of hundreds of caches, all of which may be sharing data. However, selective caching does not come without a cost. Some portions of the global memory space will become uncacheable within the GPU and bypassing on-chip caches can place additional load on limited off-chip memory resources. In the following sections, we show that by leveraging memory request coalescing, small CPU-side caches, improved interconnect efficiency, and promiscuous read-only caching, selective caching GPUs can perform nearly as well as HW cache-coherent CPU–GPU systems.

3. GPU SELECTIVE CACHING

Historically, GPUs have not required hardware cache coherence because their programming model did not provide a coherent address space between threads running on separate SMs [47]. CPUs however, support hardware cache coherence because it is heavily relied upon by both system and application programmers to ensure correctness in multi-threaded programs. Existing GPU programming models do not guarantee data correctness when CPU and GPU accesses interleave on the same memory location while the GPU is executing. One way to provide such guarantees is to enforce CPU–GPU hardware cache coherence, albeit with significant implementation complexity as previously discussed.

Alternatively, if the GPU does not cache any data that is concurrently cached by the CPU, no hardware coherence messages need to be exchanged between the CPU and GPU, yet data correctness is still guaranteed. This approach also decouples the, now private, coherence protocol decisions in CPU and GPU partitions, facilitating multi-vendor system integration. We now discuss how CPU–GPU memory can provide this single shared memory abstraction without implementing hardware cache coherence. We then propose several micro-architectural enhancements to enable selective caching to perform nearly as well as hardware cache coherence, while maintaining the programmability benefits of hardware cache coherence.

3.1 Naive Selective Caching

As shown in Figure 3, three simple principles enable the GPU to support a CPU-visible shared memory by implementing selective caching. First, the CPU is always allowed

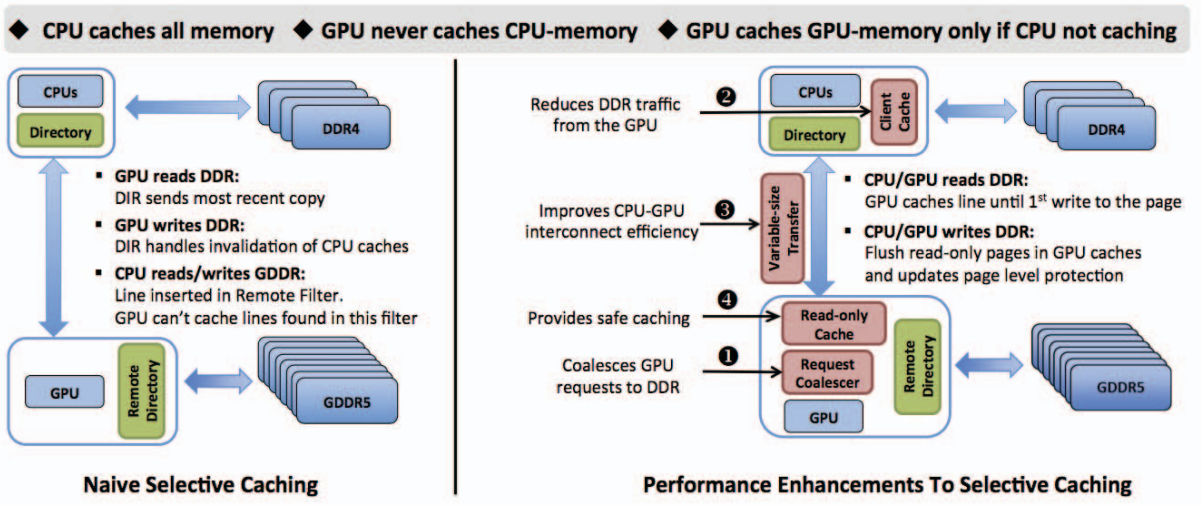

Figure 3: Overview of naive selective caching implementation and optional performance enhancements. Selective caching GPUs maintain memory coherence with the CPU while not requiring hardware cache coherence within the GPU domain.

to cache any data in the system regardless of whether that data is physically located in the memory attached to the GPU or the CPU. Second, the GPU is never allowed to cache data that resides within the CPU memory. Finally, the GPU may cache data from its own local memory if and only if the CPU is not also caching a copy of this data.

When the CPU is known to be caching a line that is homed in GPU memory and the GPU requests this line, the request must be routed to the CPU where the requested data is serviced from the CPU cache, rather than the GPU memory. Similarly, if the GPU is caching a line that the CPU requests, then this line must be flushed from the GPU caches when the request is received by the GPU memory controller. By disallowing caching of memory in use by the CPU, the GPU cannot violate the CPU hardware coherence model.

The primary microarchitectural structure needed by the GPU to implement selective caching is the *remote directory*. The remote directory block shown in Figure 3 tracks approximately, but conservatively, the cache lines homed in GPU memory that are presently cached at the CPU. When the CPU requests a line from GPU memory, its cache block address is entered into the remote directory. If the address was not already present, the GPU probes and discards the line from all GPU caches, as in a conventional invalidation-based coherence protocol. Once a cache block is added to the GPU remote directory, it becomes un-cacheable within the GPU; future GPU accesses to the line will be serviced from the CPU cache.

To limit hardware cost, we implement the remote directory as a cuckoo filter (a space efficient version of a counting bloom filter) that never reports false negatives but may report false positives [7, 18]. Thus, the remote directory may erroneously, but conservatively, indicate that a line is cached at the CPU that has never been requested, but will accurately reference all lines that have actually been requested. False positives in the remote directory generate a spurious request to the CPU, which must respond with a negative acknowledgement (NACK) should the line not be present in the CPU cache. This request will then be serviced from the GPU memory system. Similarly, if the CPU has cached a line homed in GPU memory (causing a remote directory insertion) and has since evicted it, the CPU may also NACK a GPU request, causing the request to return to the GPU memory for fulfillment.

Because entries are inserted but never pruned from our remote directory, we must track if the directory becomes full or reaches a pre-selected high-water mark. If it becomes full, our implementation forces the CPU to flush all cache lines homed in GPU memory and then resets the remote directory. This limited cache flush operation does not flush any lines homed in CPU memory, the vast majority of the system's memory capacity. In our design, the flush is performed by triggering a software daemon to call the Linux `cacheflush` trap.

The remote directory is sized to track CPU caching of up to 8MB of GPU memory, which when fully occupied requires just 64KB of on-chip storage to achieve a false positive rate of 3%. In the workloads we evaluate, the remote directory remains largely empty, and neither the capacity nor false positive rate have a significant impact on GPU performance. If workloads emerge that heavily utilize concurrent CPU–GPU threads, the size and performance of this structure will need to be re-evaluated. However if `cacheflush` trapping should become excessive due to an undersized remote directory, page-migration of CPU–GPU shared pages out of GPU memory and into CPU memory can also be employed to reduce pressure on the GPU remote directory.

3.2 Improving Selective Caching Performance

Caches have consistently been shown to provide significant performance gains thanks to improved bandwidth and latency. As such, naively bypassing the GPU caches based on the mechanisms described in Section 3.1 should be expected to hurt performance. In this subsection, we describe three architectural improvements that mitigate the impact of selectively bypassing the GPU caches and provide perfor-

mance approaching a system with hardware cache coherence.

3.2.1 Cacheless Request Coalescing

The first optimization we make to our naive selective caching design is to implement aggressive miss status handling register (MSHR) request coalescing for requests sent to CPU memory, labeled ❶ in Figure 3. MSHR request coalescing can significantly reduce the request traffic to CPU memory without violating coherency guarantees. Request coalescing works by promoting the granularity of an individual load request (that may be as small as 64 bits) to a larger granularity (typically 128B cache lines) before issuing the request to the memory system. While this larger request is in-flight, if other requests are made within the same 128B block, then these requests can simply be attached to the pending request list in the corresponding MSHR and no new request is issued to the memory system.

To maintain correctness in a selective caching system, this same coalescing scheme can be utilized, but data that is returned to the coalesced requests for which no pending request is found must be discarded immediately. Discarding data in this way is similar to self-invalidating coherence protocols, which attempt to minimize invalidation traffic in CC-NUMA systems [38,39]. Whereas most MSHR implementations allocate their storage in the cache into which the pending request will be inserted, our cache-less request coalescing must have local storage to latch the returned data. This storage overhead is negligible compared to the aggregate size of the on-chip caches that are no longer needed with selective caching.

Table 2 shows the fraction of GPU memory requests that can be coalesced by matching them to pre-existing in-flight memory requests. We call request coalescing that happens within a single SM *L1 coalescing* and coalescing across SMs *L1+L2 coalescing*. On average, 35% of memory requests can be serviced via cacheless request coalescing. While a 35% hit rate may seem low when compared to conventional CPU caches, we observe that capturing spatial request locality via request coalescing provides the majority of the benefit of the L1 caches (44.4% hit rate) found in a hardware cache-coherent GPU, as shown in Table 1.

Workload	L1 Coalescing	L1+L2 Coalescing
backprop	54.2	60.0
bfs	15.8	17.6
btree	69.4	82.4
cns	24.8	28.1
comd	45.7	53.8
kmeans	0.0	0.0
minife	29.0	32.6
mummer	41.9	51.1
needle	0.1	1.8
pathfinder	41.4	45.8
srad_v1	30.8	34.2
xsbench	15.6	18.0
Average	30.7	35.4

Table 2: Percentage of memory accesses that can be coalesced into existing in-flight memory requests, when using L1 (intra-SM) coalescing, and L1 + L2 (inter-SM) coalescing.

3.2.2 CPU-side Client Cache

Although memory request coalescing provides hit rates approaching that of conventional GPU L1 caches, it still falls short as it cannot capture temporal locality. Selective caching prohibits the GPU from locally caching lines that are potentially shared with the CPU but it does not preclude the GPU from remotely accessing coherent CPU caches. We exploit this opportunity to propose a *CPU-side GPU client cache* (label ❷ in Figure 3), which takes advantage of temporal locality not exploited by MSHR coalescing.

To access CPU memory, the GPU must already send a request to the CPU memory controller to access the line. If request coalescing has failed to capture re-use of a cache line, then multiple requests for the same line will be sent to the CPU memory controller causing superfluous transfers across the DRAM pins, wasting precious bandwidth. To reduce this DRAM pressure, we introduce a small client cache at the CPU memory controller to service these GPU requests, thereby shielding the DDR memory system from repeated requests for the same line. Our proposed GPU client cache participates in the CPU coherence protocol much like any other coherent cache on the CPU die, however lines are allocated in this cache only upon request by an off-chip processor, such as the GPU.

This single new cache does not introduce the coherence and interconnect scaling challenges of GPU-side caches, but still provides some latency and bandwidth filtering advantages for GPU accesses. One might consider an alternative where GPU-requested lines are instead injected into the existing last-level cache (LLC) at the CPU. In contrast to an injection approach, our dedicated client cache avoids thrashing the CPU LLC when the GPU streams data from CPU memory (a common access pattern). By placing this client cache on the CPU-side rather than the GPU-side of the CPU–GPU interconnect, we decouple the need to extend the CPU's hardware cache coherence protocol into even one on-die GPU cache. However, because the GPU client cache is located at the CPU-side of the CPU–GPU interconnect, it provides less bandwidth than a GPU-side on-die cache. As described in Section 2 and shown in Figure 2, this bandwidth loss is typically not performance-critical.

3.2.3 Variable-size Link Transfers

Conventional memory systems access data at cache line granularity to simplify addressing and request matching logic, improve DRAM energy consumption, and exploit spatial locality within caches. Indeed, the minimum transfer size supported by DRAM is usually a cache line. Cache line-sized transfers work well when data that was not immediately needed can be inserted into an on-chip cache, but with selective caching, unrequested data transferred from CPU memory must be discarded. Hence, portions of a cache line that were transferred, but not matched to any coalesced access, result in wasted bandwidth and energy.

The effect of this data over-fetch is shown in Table 3,

Workload	Avg. Cacheline Utilization(%)
backprop	85.9
bfs	37.4
btree	78.7
cns	77.6
comd	32.6
kmeans	25.0
minife	91.6
mummer	46.0
needle	39.3
pathfinder	86.6
srad_v1	96.3
xsbench	30.3
Average	60.6

Table 3: Utilization of 128B cache line requests where the returned data must be discarded if there is no matching coalesced request.

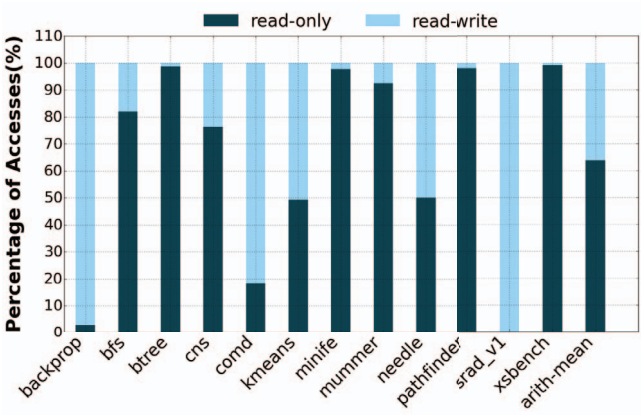

Figure 4: Fraction of 4KB OS pages that are read-only and read-write during GPU kernel execution.

where cache line utilization is the fraction of the transferred line that has a pending request when the GPU receives a cache line-sized response from CPU memory. An average cache line utilization of 60% indicates that just 77 out of 128 bytes transferred are actually used by the GPU. 51 additional bytes were transferred across the DRAM interface and CPU–GPU interconnect only to be immediately discarded.

To address this inefficiency, architects might consider reducing the transfer unit for selective caching clients from 128B down to 64 or 32 bytes. While fine-grained transfers improve transfer efficiency by omitting unrequested data, that efficiency is offset by the need for multiple small requests and packetization overhead on the interconnect. For example, in our link implementation, a transfer granularity of 32B achieves at best 66% link utilization (assuming all data is used) due to interconnect protocol overheads, while 128B transfers (again, assuming all data is used) can achieve 88% efficiency.

To maintain the benefit of request coalescing, but reduce interconnect inefficiency, we propose using *variable-size transfer units* on the CPU–GPU interconnect (labeled ❸ in Figure 3). To implement variable-size transfer units at the GPU, we allocate GPU MSHR entries at the full 128B granularity; coalescing requests as described in Section 3.2.1. However, when a request is issued across the CPU–GPU interconnect, we embed a bitmask in the request header indicating which 32B sub-blocks of the 128B cache line should be transferred on the return path. While this initial request is pending across the interconnect, if additional requests for the same 128B cache line are made by the GPU, those requests will be issued across the interconnect and their 32B sub-block mask will be merged in the GPU MSHR.

Similar to the GPU-side MSHR, variable sized transfer units require that the CPU-side client cache also maintain pending MSHR masks for requests it receives, if it can not service the requests immediately from the cache. By maintaining this mask, when the DRAM returns the 128B line, only those 32B blocks that have been requested are transferred to the GPU (again with a bitmask indicating which blocks are included). Because there may be both requests and responses in-flight simultaneously for a single 128B line, it is possible that two or more responses are required to fulfill the data requested by a single MSHR; the bitmasks included in each response facilitate this matching. Because GPUs typically perform SIMD lane-level request coalescing within an SM, 32B requests happen to be the minimum and most frequently sized request issued to the GPU memory system. As a result, we do not investigate supporting link transfer sizes smaller than 32 bytes, which would require microarchitectural changes within the GPU SM.

3.3 Promiscuous Read-Only Caching

Selective caching supports coherence guarantees by bypassing GPU caches when hardware cache-coherence operations could be needed. Thus far, our selective caching architecture has assumed that the GPU must avoid caching all data homed in CPU memory. We identify that we can loosen this restriction and allow GPU caching of CPU memory, but only if that data can be guaranteed to be read-only by both the CPU and GPU.

Figure 4 shows the fraction of data touched by the GPU that is read-only or both read and written, broken down at the OS page (4KB) granularity. In many workloads, we find the majority of the data touched by the GPU is read-only at the OS page level. We examine this data at page granularity because, even without hardware cache coherence, it is possible (though expensive) to guarantee correctness through OS page protection mechanisms entirely in software. Any cache may safely contain data from read-only OS pages. However, if the page is re-mapped as read-write, cached copies of the data at the GPU must be discarded, which will occur as part of the TLB shootdown process triggered by the permission change [61].

We propose that despite lacking hardware cache coherence, selective caching GPUs may choose to implement *promiscuous read-only caching of CPU-memory*, relying on such page level software coherence to provide correctness (labeled ❹ in Figure 3). To implement read-only caching, the GPU software run-time system speculatively marks pages within the application as read-only at GPU kernel launch time. It also tracks which pages may have

been marked read-only by the application itself to prevent speculation conflicts. With pages speculatively marked as read-only, when the GPU requests pages from the CPU memory, the permissions bit in the TLB entry is checked to determine if lines from this page are cacheable by the GPU despite being homed in CPU memory. Similarly, if the line resides in GPU memory but is marked as cached by the CPU in the remote directory, this line can still be cached locally because it is read-only.

If a write to a read-only page occurs at either the CPU or GPU, a protection fault is triggered. A write by the CPU invokes a fault handler on the faulting core, which marks the line as read/write at the CPU and uncacheable at the GPU. The fault handler then triggers a TLB shootdown, discarding the now stale TLB entry from all CPU and GPU TLBs. This protection fault typically incurs a 3-5us delay. The next access to this page at a GPU SM will incur a hardware page walk to refetch this PTE, typically adding < 1us to the first access to this updated page.

A faulting write at the GPU is somewhat more complex, as protection fault handlers currently do not run on a GPU SM. Instead, the GPU MMU must dispatch an interrupt to the CPU to invoke the fault handler. That SW handler then adjusts the permissions and shoots down stale TLB entries, including those at the GPU. The CPU interrupt overhead raises the total unloaded latency of the fault to 20us (as measured on NVIDIA's Maxwell generation GPUs). However, only the faulting warp is stalled: the SM can continue executing other non-faulting warps. Once the GPU receives an acknowledgement that the fault handling is complete, it will re-execute the write, incurring a TLB miss and a hardware page walk to fetch the updated PTE entry.

The many-threaded nature of the GPU allows us to largely hide the latency of these permission faults by executing other warps, thereby mitigating the performance impact of the high SW fault latency in nearly all of our workloads. Nevertheless, software page fault handlers are orders of magnitude more expensive than hardware cache-coherence messaging and may erode the benefit of promiscuous read-only caching if permission faults are frequent. We evaluate the performance of promiscuous caching under different software faulting overhead costs in Section 5.2.

4. METHODOLOGY

We evaluate selective caching via simulation on a system containing discrete CPUs and GPUs with DDR4 and GDDR5 memories attached to the CPU and GPU, respectively. Our simulation environment is based on GPGPU-Sim [5], which we modify to support a heterogeneous memory system with simulation parameters shown in Table 4. We use bandwidth-aware page placement for all simulations as it has been shown to be the best page placement strategy without requiring any application profiling or program modification [2]. In our simulated system, this page placement results in 20% of the GPU workload data being placed within the CPU-attached memory with 80% residing in the GPU-attached memory.

In our system, the CPU is connected to the GPU via a full duplex CPU–GPU interconnect. The interconnect has peak bandwidth of 90GB/s using 16B flits for both data and control messages with each data payload of up to 128B requiring a single header flit. Thus, for example, a 32B data message will require sending 1 header flit + 2 data flits = 3 flits in total. To simulate an additional interconnect hop to remote CPU memory, we model an additional fixed, pessimistic, 100-cycle interconnect latency to access the DDR4 memory from the GPU. This overhead is derived from the single additional interconnect hop latency found in SMP CPU-only designs, such as the Intel Xeon [32]. When simulating request coalescing within the GPU, we use the same number of MSHRs as the baseline configuration but allow the MSHRs to have their own local return value storage in the selective caching request coalescing case. The CPU-side GPU client cache is modeled as an 8-way set associative, write-through, no write-allocate cache with 128B line size of varying capacities shown later in Section 5.1.2. The client cache latency is 200 cycles, comprising 100 cycles of interconnect and 100 cycles of cache access latency. To support synchronization operations between CPU and GPU, we augment the GPU MSHRs to support atomic operations to data homed in either physical memory; we assume the CPU similarly can issue atomic accesses to either memory.

Simulator	GPGPU-Sim 3.x
GPU Arch	NVIDIA GTX-480 Fermi-like
GPU Cores	15 SMs @ 1.4Ghz
L1 Caches	16kB/SM, 3 cycle latency
L1 MSHRs	64 Entries/L1
L2 Caches	128kB/Channel, 120 cycle lat.
L2 MSHRs	128 Entries/L2 Slice
CPU Client Cache	512KB, 200 cycle latency
Memory System	
GPU GDDR5	8-channels, 336GB/sec aggregate
CPU DDR4	4-channels, 80GB/sec aggregate
SW Page Faults	16 concurrent per SM
DRAM Timings	RCD=RP=12, RC=40, CL=WR=12
DDR4 Burst Len.	8
CPU–GPU Interconnect	
Link Latency	100 GPU core cycles
Link Bandwidth	90 GB/s Full-Duplex
Req. Efficiency	32B=66%, 64B=80%, 128B=88%

Table 4: Parameters for experimental GPGPU based simulation environment.

To model promiscuous read-only caching, we initially mark all the pages (4kB in our system) in DDR as read-only upon GPU kernel launch. When the first write is issued to each DDR page, the ensuing protection fault invalidates the TLB entry for the page at the GPU. When the faulting memory operation is replayed, the updated PTE is loaded, indicating that the page is uncacheable. Subsequent accesses to the page are issued over the CPU–GPU interconnect. Pages marked read-write are never re-marked read-only during GPU kernel execution. Using the page placement policy described earlier in this section, the GPU is able to cache 80% of the application footprint residing in GPU memory. We vary our assumption for the remote protection fault la-

tency from 20-40us and assume support for up to 16 pending software page protection faults per SM; a seventeenth fault blocks the SM from making forward progress on any warp.

We evaluate results using the Rodinia and United States Department of Energy benchmark suites. We execute the applications under the CUDA 6.0 weak consistency memory model. While we did evaluate workloads from the Parboil [62] suite, we found that these applications have uncharacteristically high cache miss rates, hence even in the hardware cache-coherent case, most memory accesses go to the DRAM. As such, we have chosen to omit these results because they would unfairly indicate that selective caching is performance equivalent to a theoretical hardware cache-coherent GPU. In Section 5 we report GPU performance as application throughput, which is inversely proportional to workload execution time.

5. RESULTS

We evaluate the performance of selective GPU caching through iterative addition of our three proposed microarchitectural enhancements on top of naive selective caching. We then add promiscuous read-only caching and finally present a sensitivity study for scenarios where the workload footprint is too large for a performance-optimal page placement split across CPU and GPU memory.

5.1 Microarchitectural Enhancements

Figure 5 shows the baseline performance of naive selective caching compared to a hardware cache-coherent GPU. Whereas performance remains as high as 95% of the baseline for some applications, the majority of applications suffer significant degradation, with applications like `btree` and `comd` seeing nearly an order-of-magnitude slowdown. The applications that are hurt most by naive selective caching tend to be those that have a high L2 cache hit rate in a hardware cache-coherent GPU implementation like `comd` (Table 1) or those that are highly sensitive to L2 cache latency like `btree` (Figure 2). Prohibiting all GPU caching of CPU memory results in significant over-subscription of the CPU memory system, which quickly becomes the bottleneck for application forward progress, resulting in nearly a 50% performance degradation across our workload suite.

5.1.1 Cacheless Request Coalescing

Our first microarchitectural proposal is to implement cacheless request coalescing as described in Section 3.2.1. With naive selective caching relying on only the SIMD lane-level request coalescer within an SM, performance of the system degrades to just 42% of the hardware cache-coherent GPU, despite only 20% of the application data residing in CPU physical memory. Introducing request coalescing improves performance to 74% and 79% of a hardware cache-coherent GPU when using L1 coalescing and L1+L2 coalescing, respectively. This improvement comes from a drastic reduction in the total number of requests issued across the CPU–GPU interconnect and reducing pressure on the CPU memory. Surprisingly `srad_v1` shows a 5% speedup over the hardware cache-coherent GPU when using L1+L2 request coalescing. `srad_v1` has a large number of pages that are written without first being read, thus the CPU

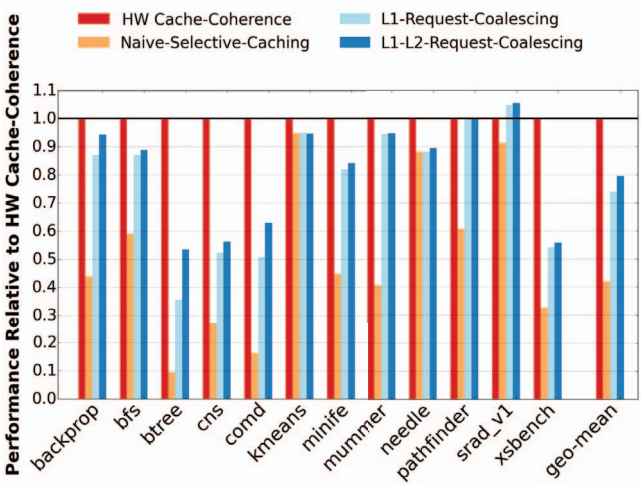

Figure 5: GPU performance under selective caching with uncoalesced requests, L1 coalesced requests, L1+L2 coalesced requests.

DRAM system benefits from the elimination of reads that are caused by the write-allocate policy in the baseline GPU's L2 cache. Because the request coalescing hit rates, shown in Table 2, lag behind the hardware cached hit rates, selective caching still places a higher load on the interconnect and CPU memory than a hardware cache-coherent GPU, which translates into the 21% performance reduction we observe when using selective caching with aggressive request coalescing.

5.1.2 CPU-side Client Cache

Whereas request coalescing captures much of the spatial locality provided by GPU L1 caches, it cannot capture any long distance temporal locality. Figure 6 shows the performance differential of adding our proposed CPU-side client cache to L1+L2 request coalescing within the selective caching GPU. This GPU client cache not only reduces traffic to CPU DRAM from the GPU, but also improves latency for requests that hit in the cache and provides additional bandwidth that the CPU–GPU interconnect may exploit. We observe that performance improvements scale with client cache size up to 512KB before returns diminish. Combining a 512KB, 8-way associative client cache with request coalescing improves performance of our selective caching GPU to within 90% of the performance of a hardware cache-coherent GPU. Note that `btree` only benefits marginally from this client cache because accessing the client cache still requires a round-trip interconnect latency of 200 cycles (Section 4). `btree` is highly sensitive to average memory access latency (Figure 2), which is not substantially improved by placing the client cache for GPU requests on the CPU die.

The size of an on-die CPU client cache is likely out of the hands of GPU architects, and for CPU architects allocating on-die resources for an external GPU client may seem an unlikely design choice. However, this client cache constitutes only a small fraction of the total chip area of modern CPUs (0.7% in 8-core Xeon E5 [27]) and is the size of just one ad-

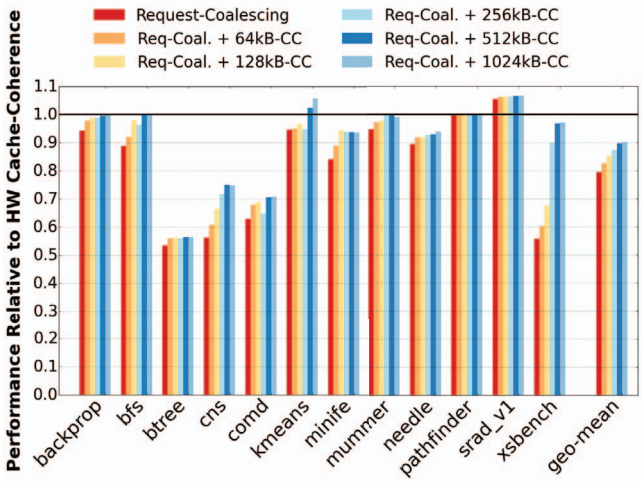

Figure 6: GPU performance with selective caching when combining request coalescing with CPU-side caching for GPU clients at 64KB–1MB cache capacities. (CC: Client-Cache)

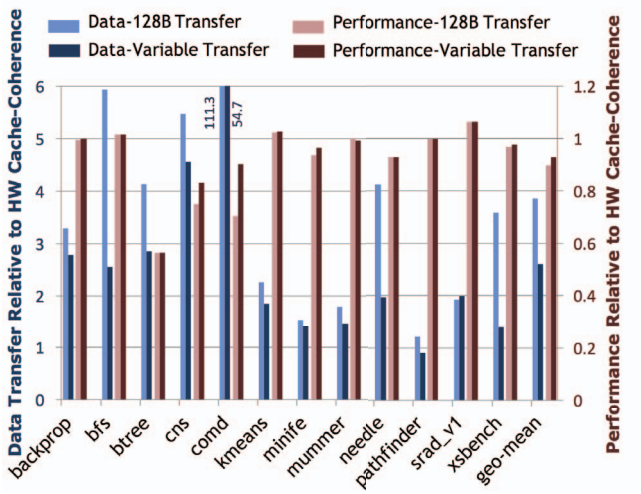

Figure 7: GPU data transferred across CPU-GPU interconnect (left y-axis) and performance (right y-axis) for 128B cache line-size link transfers and variable-size link transfers, respectively.

ditional private L2 cache within the IBM Power 8 processor. Much like processors have moved towards on-die integration of PCIe to provide improved performance with external peripherals, we believe the performance improvements due to this cache are significant enough to warrant integration. For CPU design teams, integrating such a cache into an existing design is likely easier than achieving performance by extending coherence protocols into externally developed GPUs. The GPU client cache also need not be specific to just GPU clients, other accelerators such as FPGAs or spatial architectures [50, 56] that will be integrated along-side a traditional CPU architecture will also likely benefit from such a client cache.

5.1.3 Variable-size Link Transfers

Request coalescing combined with the CPU client cache effectively reduce the pressure on the CPU DRAM by limiting the number of redundant requests that are made to CPU memory. The CPU client cache exploits temporal locality to offset data overfetch that occurs on the DRAM pins when transferring data at cache line granularity, but does not address CPU–GPU interconnect transfer inefficiency. To reduce this interconnect over-fetch, we propose variable-sized transfer units (see Section 3.2.3). The leftmost two bars for each benchmark in Figure 7 show the total traffic across the CPU–GPU interconnect when using traditional fixed 128B cache line requests and variable-sized transfers, compared to a hardware cache-coherent GPU. We see that despite request coalescing, our selective caching GPU transfers nearly 4 times the data across the CPU–GPU interconnect than the hardware cache-coherent GPU. Our variable-sized transfer implementation reduces this overhead by nearly one third to just 2.6x more interconnect traffic than the hardware cache-coherent GPU.

This reduction in interconnect bandwidth results in performance gains of just 3% on average, despite some applications like `comd` showing significant improvements. We observe that variable-sized transfers can significantly improve bandwidth utilization on the CPU–GPU interconnect but most applications remain performance-limited by the CPU memory bandwidth, not the interconnect itself. When we increase interconnect bandwidth by 1.5x without enabling variable-sized requests, we see an average performance improvement of only 1% across our benchmark suite. Variable-sized requests are not without value, however; transferring less data will save power or allow this expensive off-chip interface to be clocked at a lower frequency, but evaluating the effect of those improvements is beyond the scope of this work.

5.2 Promiscuous GPU Caching

By augmenting selective caching with request coalescing, a GPU client cache, and variable-sized transfers, we achieve performance within 93% of a hardware cache-coherent GPU. As described in Section 3.3, the GPU can be allowed to cache CPU memory that is contained within pages that are marked as read-only by the operating system. The benefit of caching data from such pages is offset by protection faults and software recovery if pages promiscuously marked as read-only and cached by the GPU are later written. Figure 8 (RO-ZeroCost) shows the upper bound on possible improvements from read-only caching for an idealized implementation that marks all pages as read-only and transitions them to read-write (and thus uncacheable) without incurring any cost when executing the required protection fault handling routine. In a few cases, this idealized implementation can outperform the hardware cache-coherent GPU because of the elimination of write allocations in the GPU caches, which tend to have little to no reuse.

We next measure the impact of protection fault cost, varying the unloaded fault latency from 20us to 40us (see Figure 8), commensurate with typical fault latencies on today's GPU implementations. While a fault is outstanding, the faulting warp and any other warp that accesses the same

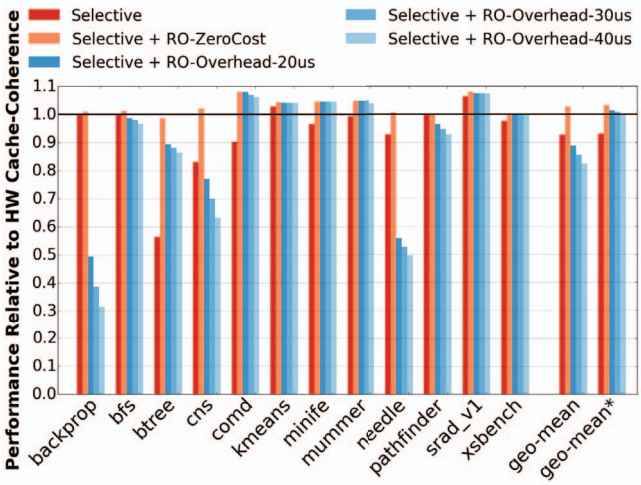

Figure 8: GPU performance when using Selective caching (Request-Coalescing + 512kB-CC + Variable-Transfers) combined with read-only caching. geo-mean*: Geometric mean excluding `backprop`, `cns`, `needle`, where read-only caching should be disabled. (RO: Read-Only)

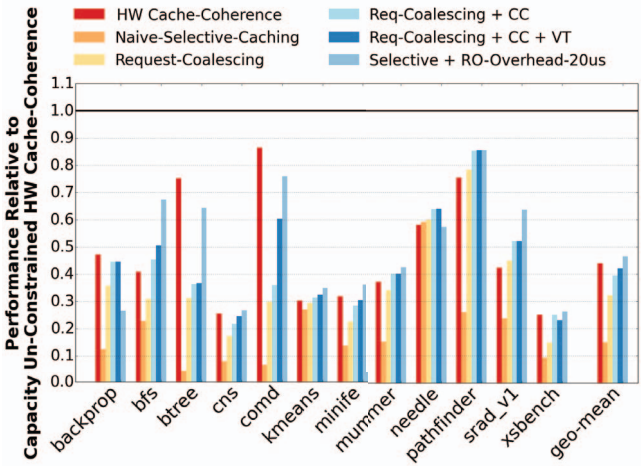

Figure 9: GPU performance under memory capacity constraints. (CC: Client-Cache, VT: Variable-sized Transfer Units)

address are stalled; but, other warps may proceed, mitigating the impact of these faults on SM forward progress. The latency of faults can be hidden if some warps executing on an SM are reading this or other pages. However, if all warps issue writes at roughly the same time, the SM may stall due to a lack of schedulable warps or MSHR capacity to track pending faults. When accounting for fault overheads, our selective caching GPU with promiscuous read-only caching achieves only 89% of the performance of the hardware cache-coherent GPU.

With a 20us fault latency, we see that 7 of 12 workloads exhibit improvement from promiscuous read-only caching and that `btree` sees a large 35% performance gain as it benefits from improvements to average memory access latency. In contrast, three workloads, `backprop`, `cns`, and `needle`, suffer considerable slowdowns due to the incurred protection fault latency. These workloads tend to issue many concurrent writes, exhausting the GPUs ability to overlap execution with the faults. For such workloads, we advocate disabling promiscuous read-only caching in software (e.g., via a mechanism that tracks the rate of protection faults, disabling promiscuous read-only caching when the rate exceeds a threshold).

In summary, the effectiveness of promiscuous read-only caching depends heavily on the latency of protection faults and the GPU microarchitecture's ability to overlap the execution of non-faulting warps with those faults, which can vary substantially across both operating systems and architectures. In systems where the fault latency is higher than the 20us (as measured on current NVIDIA systems), more judicious mechanisms must be used to identify read-only pages (e.g., explicit hints from the programmer via the `mprotect` system call.)

5.3 Discussion

One use case in the future may be that GPU programmers will size their application's data to extend well beyond the performance-optimal footprint in CPU and GPU memory. With excess data spilling over into the additional capacity provided by the CPU memory, performance bottlenecks will shift away from the GPU towards the CPU memory system. In such cases, the GPU caching policy for CPU memory will come under additional pressure due to the increased traffic skewed towards CPU memory.

To understand how selective caching affects performance under such a scenario, we evaluate a situation wherein the application data has been sized so that 90% of the footprint resides in CPU memory and just 10% can fit within GPU memory, as compared to the nearly inverse performance-optimal 20%-80% ratio. Figure 9 shows the performance of this memory-capacity-constrained case relative to the baseline optimal ratio. We see that naive selective caching and our proposed enhancements follow the same trend of performance improvements shown previously in Section 5. Because this scenario is primarily limited by the CPU memory system, we see that in some cases the client cache and variable sized transfer interconnect optimizations can actually outperform the hardware cache-coherent GPU due to a reduction in data overfetch between the CPU memory and the GPU client. To validate our observation, we added the same client cache and variable transfers to the hardware cache-coherent baseline configuration and saw an average speedup of 4.5%. Whereas the absolute performance achieved, compared to a performance-optimal memory footprint and allocation, may not always be compelling, should software designers chose to partition their problems in this way, we believe selective caching will continue to perform as well as a hardware cache-coherent GPU.

In this work, we have primarily investigated a system where bandwidth-aware page placement provides an initial page placement that has been shown to have optimal performance [2]. Bandwidth-aware page placement is based on the premise that the GPU will place pressure on the CPU and GPU memory system in proportion to the number of pages placed in each memory. Proposals like selective caching that change the on-chip caching policy of the GPU can cause dra-

matic shifts in the relative pressure placed on each memory system, effectively changing the bandwidth-optimal placement ratio. Although we do not evaluate this phenomenon in this work, balancing initial page placement with dynamic page migration to help compensate for the lack of on-chip caching is an area that needs further investigation.

6. RELATED WORK

Cache coherence for CPUs has received great attention in the literature. Recent proposals have started to explore intra-GPU and CPU–GPU cache coherence.

CPU Systems: Scalable cache coherence has been studied extensively for CPU-based multicore systems. Kelm et al. show that scaling up coherence to hundreds or thousands of cores will be difficult without moving away from pure hardware-based coherence [23, 35], due to high directory storage overheads and coherence traffic [11, 39]. Whereas some groups have evaluated software shared memory implementations [17, 23], Martin et al. argue that hardware cache coherence for mainstream processors is here to stay, because shifting away from it simply shifts the burden of correctness into software instead of hardware [41]. Nevertheless, disciplined programming models coupled with efficient hardware implementations are still being pursued [12, 65, 66].

Self-invalidation protocols have been proposed to reduce invalidation traffic and reduce coherence miss latency [38, 39]. Our selective caching request coalescing scheme uses a similar idea, discarding a block immediately after fulfilling requests pending at the MSHR. Other proposals have classified data into private, shared, and instruction pages and have devised techniques to curtail coherence transactions for private data [14, 21, 55, 57]. We instead classify pages into read-only versus read-write and exploit the fact that read-only data can be safely cached in incoherent caches.

Ros and Kaxiras [57] have proposed a directory-less/ broadcast-less coherence protocol where all shared data is self-invalidated at synchronization points. In this scheme, at each synchronization point (e.g., lock acquire/release, memory barrier) all caches need to be searched for shared lines and those lines have to be flushed—an expensive operation to implement across hundreds of GPU caches with data shared across thousands of concurrent threads.

Heterogeneous Systems and GPUs: With the widespread adoption of GPUs as a primary computing platform, the integration of CPU and GPU systems has resulted in multiple works assuming that CPUs and GPUs will eventually become hardware cache-coherent with shared page tables [1, 2, 52, 53]. CPU–GPU coherence mechanisms have been investigated, revisiting many ideas from distributed shared memory and coherence verification [20, 34, 51]. Power et al. [54] target a hardware cache-coherent CPU–GPU system by exploiting the idea of region coherence [3,8,43,68]. They treat the CPU and the GPU as separate regions and mitigate the effects of coherence traffic by replacing a standard directory with a region directory. In contrast, we identify that CPUs and GPUs need not be cache-coherent; the benefits of unified shared memory can also be achieved via selective caching, which has lower implementation complexity.

Mixing incoherent and coherent shared address spaces has been explored before in the context of CPU-only systems [28] and the appropriate memory model for mixed CPU–GPU systems is still up for debate [19, 22, 25, 40]. Hechtman et al. propose a consistency model for GPUs based on release consistency, which allows coherence to be enforced only at release operations. They propose a write-through no-write-allocate write-combining cache that tracks dirty data at byte granularity. Writes must be flushed (invalidating other cached copies) only at release operations. Under such a consistency model, our selective caching scheme can be used to avoid the need to implement hardware support for these invalidations between the CPU and GPU.

Cache coherence for GPU-only systems has been studied by Singh et al. [60], where they propose a timestamp-based hardware cache-coherence protocol to self-invalidate cache lines. Their scheme targets single-chip systems and would require synchronized timers across multiple chips when implemented in multi-chip CPU–GPU environments. Kumar et al. [37] examine CPUs and fixed-function accelerator coherence, balancing coherence and DMA transfers to prevent data ping-pong. Suh et al. [64] propose integrating different coherence protocols in separate domains (such as MESI in one domain and MEI in another). However, this approach requires invasive changes to the coherence protocols implemented in both domains and requires significant implementation effort by both CPU and GPU vendors.

Bloom Filters: Bloom Filters [6] and Cuckoo Filters [18, 49] have been used by several architects [63, 69, 70] in the past. Fusion coherence [51] uses a cuckoo directory to optimize for power and area in a CMP system. JETTY filters [44] have been proposed for reducing the energy spent on snoops in an SMP system. We use a cuckoo filter to implement the GPU remote directory.

7. CONCLUSIONS

Introducing globally visible shared memory in future CPU–GPU systems improves programmer productivity and significantly reduces the barrier to entry of using such systems for many applications. Hardware cache coherence can provide such shared memory and extend the benefits of on-chip caching to all system memory. However, extending hardware cache coherence throughout the GPU places enormous scalability demands on the coherence implementation. Moreover, integrating discrete processors, possibly designed by distinct vendors, into a single coherence protocol is a prohibitive engineering and verification challenge.

We demonstrate that CPU–GPU hardware cache coherence is not needed to achieve the simultaneous goals of unified shared memory and high GPU performance. We show that *selective caching* with request coalescing, a CPU-side GPU client cache and variable-sized transfer units can perform within 93% of a cache-coherent GPU for applications that do not perform fine grained CPU–GPU data sharing and synchronization. We also show that promiscuous read-only caching benefits memory latency-sensitive applications by using OS page-protection mechanisms rather than relying on hardware cache coherence. Selective caching does not needlessly force hardware cache coherence into the GPU memory system, allowing decoupled designs that can maximize CPU and GPU performance, while still maintaining the CPU's traditional view of the memory system.

8. REFERENCES

[1] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2015, pp. 354–365.

[2] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs within Heterogeneous Memory Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2015, pp. 607–618.

[3] M. Alisafaee, "Spatiotemporal Coherence Tracking," in *International Symposium on Microarchitecture (MICRO)*, Decemeber 2012, pp. 341–350.

[4] AMD Corporation, "Compute Cores," https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf, 2014, [Online; accessed 15-April-2015].

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009, pp. 163–174.

[6] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM (CACM)*, vol. 13, no. 7, pp. 422–426, July 1970.

[7] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," in *European Symposium on Algorithms (ESA)*, September 2006, pp. 684–695.

[8] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking," in *International Symposium on Computer Architecture (ISCA)*, June 2005, pp. 246–257.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[10] H. Chen and C. Wong, "Wiring And Crosstalk Avoidance In Multichip Module Design," in *Custom Integrated Circuits Conference*, May 1992, pp. 28.6.1–28.6.4.

[11] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter, "Interconnect-Aware Coherence Protocols for Chip Multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, June 2006, pp. 339–351.

[12] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. Adve, V. Adve, N. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2011, pp. 155–166.

[13] C. Chou, A. Jaleel, and M. K. Qureshi, "BATMAN: Maximizing Bandwidth Utilization of Hybrid Memory Systems," Georgia Institute of Technology, Tech. Rep. TR-CARET-2015-01, March 2015.

[14] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks," in *International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 93–104.

[15] M. Daga, A. M. Aji, and W.-C. Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing," in *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, July 2011, pp. 141–149.

[16] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013, pp. 381–394.

[17] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood, "Application-Specific Protocols for User-Level Shared Memory," in *International Conference on High Performance Networking and Computing (Supercomputing)*, November 1994.

[18] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *International on Conference on Emerging Networking Experiments and Technologies*, December 2014, pp. 75–88.

[19] B. R. Gaster, D. Hower, and L. Howes, "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 1, pp. 7:1–7:26, April 2015.

[20] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010, pp. 347–358.

[21] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," in *International Symposium on Computer Architecture (ISCA)*, June 2009, pp. 184–195.

[22] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, and D. A. Wood, "QuickRelease: A Throughput-oriented Approach to Release Consistency on GPUs," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 189–200.

[23] M. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors," *ACM Transactions on Computer Systems*, vol. 11, no. 4, pp. 300–318, November 1993.

[24] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "A Case of System-level Hardware/Software Codesign and Co-verification of a Commodity Multi-processor System with Custom Hardware," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2012, pp. 513–520.

[25] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free Memory Models," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014, pp. 427–440.

[26] HSA Foundation, "HSA Platform System Architecture Specification - Provisional 1.0," http://www.slideshare.net/hsafoundation/hsa-platform-system-architecture-specification-provisional-verl-10-ratifed, 2014, [Online; accessed 28-May-2014].

[27] M. Huang, M. Mehalel, R. Arvapalli, and S. He, "An Energy Efficient 32-nm 20-MB Shared On-Die L3 Cache for Intel Xeon Processor E5 Family," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 48, no. 8, pp. 1954–1962, August 2013.

[28] J. Huh, J. Chang, D. Burger, and G. S. Sohi, "Coherence Decoupling: Making Use of Incoherence," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004, pp. 97–106.

[29] HyperTransport Consortium, "HyperTransport 3.1 Specification," http://www.hypertransport.org/docs/twgdocs/HTC20051222-0046-0035.pdf, 2010, [Online; accessed 7-July-2014].

[30] IBM Corporation, "POWER8 Coherent Accelerator Processor Interface (CAPI)," http://www-304.ibm.com/webapp/set2/sas/f/capi/home.html, 2015, [Online; accessed 16-April-2015].

[31] INTEL Corporation, "An Introduction to the Intel QuickPath Interconnect," http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009, [Online; accessed 7-July-2014].

[32] ——, "Intel Xeon Processor E5 v3 Family," http://ark.intel.com/products/family/78583/Intel-Xeon-Processor-E5-v3-Family#@All, 2015, [Online; accessed 16-April-2015].

[33] D. Johnson, M. Johnson, J. Kelm, W. Tuohy, S. S. Lumetta, and S. Patel, "Rigel: A 1,024-Core Single-Chip Accelerator Architecture," *IEEE Micro*, vol. 31, no. 4, pp. 30–41, July 2011.

[34] S. Kaxiras and A. Ros, "A New Perspective for Efficient Virtual-cache Coherence," in *International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 535–546.

[35] 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," in *International Symposium on Computer Architecture (ISCA)*, June 2009, pp. 140–151.

[36] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "WAY-

POINT: Scaling Coherence to Thousand-core Architectures," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2010, pp. 99–110.

[37] S. Kumar, A. Shriraman, and N. Vedula, "Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators," in *International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 733–745.

[38] A.-C. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," in *International Symposium on Computer Architecture (ISCA)*, June 2000, pp. 139–148.

[39] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, June 1995, pp. 48–59.

[40] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level Implications of Disaggregated Memory," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2012, pp. 1–12.

[41] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why On-chip Cache Coherence is Here to Stay," *Communications of the ACM (CACM)*, vol. 55, no. 7, pp. 78–89, July 2012.

[42] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach For Mixing Die-stacked And Off-package Memories," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2015, pp. 126–136.

[43] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," in *International Symposium on Computer Architecture (ISCA)*, June 2005, pp. 234–245.

[44] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi, "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, January 2001, pp. 85–96.

[45] NVIDIA Corporation, "Unified Memory in CUDA 6," http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, 2013, [Online; accessed 28-May-2014].

[46] ——, "NVIDIA Launches World's First High-Speed GPU Interconnect, Helping Pave the Way to Exascale Computing," http://nvidianews.nvidia.com/News/NVIDIA-Launches-World-s-First-High-Speed-GPU-Interconnect-Helping-Pave-the-Way-to-Exascale-Computin-ad6.aspx, 2014, [Online; accessed 28-May-2014].

[47] ——, "CUDA C Programming Guild v7.0," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2015, [Online; accessed 09-May-2015].

[48] ——, "New NVIDIA TITAN X GPU Powers Virtual Experience "Thief in the Shadows" at GDC," http://blogs.nvidia.com/blog/2015/03/04/smaug/, 2015, [Online; accessed 16-April-2015].

[49] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[50] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures," in *International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 142–153.

[51] S. Pei, M.-S. Kim, J.-L. Gaudiot, and N. Xiong, "Fusion Coherence: Scalable Cache Coherence for Heterogeneous Kilo-Core System," in *Advanced Computer Architecture*, ser. Communications in Computer and Information Science. Springer, 2014, vol. 451, pp. 1–15.

[52] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014, pp. 743–758.

[53] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 568–578.

[54] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *International Symposium on Microarchitecture (MICRO)*, December 2013, pp. 457–467.

[55] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "SWEL: Hardware Cache Coherence Protocols to Map Shared Data Onto Shared Caches," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2010, pp. 465–476.

[56] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 13–24.

[57] A. Ros and S. Kaxiras, "Complexity-effective Multicore Coherence," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2012, pp. 241–252.

[58] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2012, pp. 1–12.

[59] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models," in *International Symposium on Microarchitecture (MICRO)*, December 2015, pp. 647–659.

[60] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2013, pp. 578–590.

[61] P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 12–24, June 1990.

[62] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, v.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign, Tech. Rep., March 2012.

[63] K. Strauss, X. Shen, and J. Torrellas, "Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, June 2006, pp. 327–338.

[64] T. Suh, D. Blough, and H.-H. Lee, "Supporting Cache Coherence In Heterogeneous Multiprocessor Systems," in *Design Automation Conference (DAC)*, February 2004, pp. 1150–1155.

[65] H. Sung and S. V. Adve, "DeNovoSync: Efficient Support for Arbitrary Synchronization Without Writer-Initiated Invalidations," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2015, pp. 545–559.

[66] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient Hardware Support for Disciplined Non-determinism," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013, pp. 13–26.

[67] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, "Scaling the Power Wall: A Path to Exascale," in *International Conference on High Performance Networking and Computing (Supercomputing)*, November 2014.

[68] J. Zebchuk, E. Safi, and A. Moshovos, "A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy," in *International Symposium on Microarchitecture (MICRO)*, Decemeber 2007, pp. 314–327.

[69] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A Tagless Coherence Directory," in *International Symposium on Microarchitecture (MICRO)*, Decemeber 2009, pp. 423–434.

[70] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan, "SPATL: Honey, I Shrunk the Coherence Directory," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2011, pp. 33–44.