2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Scalable Distributed Last-Level TLBs Using Low-Latency Interconnects

Srikant Bharadwaj*
Georgia Institute of Technology
AMD Research
srikant.bharadwaj@amd.com

Guilherme Cox*†
Rutgers University
guilherme.cox@rutgers.edu

Tushar Krishna
Georgia Institute of Technology
tushar@ece.gatech.edu

Abhishek Bhattacharjee
Rutgers University
abhib@cs.rutgers.edu

***Abstract*—Recent studies have shown the potential of last-level TLBs shared by multiple cores in tackling memory translation performance challenges posed by "big data" workloads. A key stumbling block hindering their effectiveness, however, is their high access time. We present a design methodology to reduce these high access times so as to realize high-performance and *scalable* shared L2 TLBs. As a first step, we study the benefits of replacing monolithic shared TLBs with a distributed set of small TLB slices. While this approach does reduce TLB lookup latency, it increases interconnect delays in accessing remote slices. Therefore, as a second step, we devise a lightweight single-cycle interconnect among the TLB slices by tailoring wires and switches to the unique communication characteristics of memory translation requests and responses. Our approach, which we dub NOCSTAR (NOCs for scalable TLB architecture), combines the high hit rates of shared TLBs with low access times of private L2 TLBs, enabling significant system performance benefits.**

***Index Terms*—Virtual memory, TLB, network-on-chip, caches.**

I. Introduction

Memory-intensive workloads pose many performance challenges for modern computer systems. One important challenge is the question of how to achieve efficient virtual-to-physical address translation [1, 2]. Efficient Translation Lookaside Buffers (TLBs) are central to achieving fast address translation. TLB performance depends on three attributes – access time, hit rate, and miss penalty. Recent studies improve TLB hit rates with techniques that use hardware-only or hardware-software co-design approaches like sub-blocking [3], coalescing [4–6], clustering [7], part-of-memory optimizations [8, 9], superpages [10–14], direct segments [15, 16], and range translations [17, 18]. Others have used prefetching and speculative techniques to support the illusion of higher effective TLB capacity [11, 19–26]. Similarly, synergistic TLBs, which evict translations between per-core TLBs, can improve hit rates [27]. Other studies have focused on reducing TLB miss penalties [28–33]. Finally – and most pertinently to this study – shared last-level TLBs have been proposed to improve the overall hit rate by avoiding replication of shared translations that occur in multi-threaded programs or multi-programmed workloads using shared libraries and OS structures [19, 34] .

*Joint first authors, both of whom have contributed equally to this work.
†Now at NVIDIA.

Unfortunately, many of these approaches side-step the attribute of TLB *access time*. Consider, for example, shared TLB organizations. Processor vendors implement two-level TLBs private to each core today. However, recent academic work has shown that replacing them with an equivalently-sized *shared* (among cores) L2 TLB eliminates as much as 70-90% of the page table walks on modern systems [34]. However, sharing also results in larger structures that are physically further from cores, resulting in longer access latency. Recall that address translation latency is on the critical path of *every* L1 cache access. Consequently, a TLB with more hits may not be attractive if each hit actually becomes slower. As memory demands continue to increase, improving TLB reach without significantly increasing access time is a key research challenge.

Our goal is to translate the hit rate benefits of shared TLBs to overall runtime speedup. This requires a conceptual rethink of how architects build scalable shared TLB hierarchies. The challenge with a multi-banked monolithic shared L2 TLB structure is that it suffers from high latency. A natural alternative is a distributed shared L2 TLB, akin to NUCA LLCs. Each distributed shared TLB slice can be made small to keep access latency low. Unfortunately, this makes TLB access non-uniform, depending on the location of the slice where the translation is cached. Our studies on a 32-core Haswell system show that a distributed shared L2 TLB consequently *degrades* performance by 7%, despite having 70% fewer misses on average than private L2 TLBs. This is because TLB accesses are more latency critical than data cache accesses.

We propose **NOCSTAR** (**NOC**s for **s**calable **TLB** **ar**chitectures), a design methodology to architect scalable low-latency shared last-level TLBs. NOCSTAR relies on the latency characteristics of on-chip wires and the bandwidth characteristics of address translation requests to realize a lightweight specialized interconnect that provides near single-cycle access to remote shared TLB slices, however far they may be on-chip. Consequently, NOCSTAR provides the hit rate benefits of shared TLBs at the access latency of private TLBs via the following features:

① High capacity: NOCSTAR offers higher hit rates than private L2 TLBs by eliminating replication and improving utilization.

② Low lookup latency: NOCSTAR achieves low lookup latency by replacing a monolithic shared L2 TLB structure with

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00030

smaller TLB slices distributed across cores.

③ Low network latency: NOCSTAR employs a light-weight interconnect to connect cores to the distributed TLB slices. This interconnect provides near single-cycle latencies from any source to any remote TLB, reducing network traversal latency.

The confluence of these features enables NOCSTAR to offer within 95% of the performance of an ideal, zero-interconnect-latency shared TLB. With an area-equivalent configuration*, NOCSTAR outperforms private L2 TLBs on 16-64 core Haswell systems by an average of 1.13× and up to 1.25× across a suite of real-world workloads.

II. Background and Motivation

While NOCSTAR is applicable to both instruction and data TLBs, we focus on the latter. Our focus is driven in part by the fact that the data-side TLB pressure is growing with the prevalence of big-data workloads [4, 14–16, 18, 24, 35–38].

Fig. 1 summarizes the TLB architectures considered in this study. Fig. 1(a) shows conventional private L2 TLBs, while Fig. 1(b) shows the shared L2 TLB alternative proposed in prior work [19, 34, 39]. As we scale the size of the shared TLB, a practical design would involve banking this monolithic structure, as shown in Fig. 1(c). We evaluate this design and ultimately find that distributing the TLB slices across cores (see Fig. 1(d)) with a fast NOC is a better choice. Throughout this paper, we use the term *TLB access latency* to refer to *TLB's SRAM lookup latency + interconnect latency*.

A. Limitations of Private TLBs and Promise of Shared TLBs

Private two-level TLBs are a staple in modern server-class chips like Intel's Skylake or AMD's Ryzen processors. For example, Intel's Skylake chip uses 64-entry L1 TLBs backed by 1536-entry, 12-way set associative L2 TLBs per core. Unfortunately, private L2 TLBs suffer from the classic pitfalls of private caching structures – i.e., replication and poor utilization [34]. Consider the problem of replication. Multi-threaded applications running on a multi-core naturally replicate virtual-to-physical translations across private L2 TLBs as they are part of the same virtual address space. Perhaps more surprisingly, even multiprogrammed combinations of single-threaded programs exhibit replication as different processes can share libraries and OS structures [34]. Private L2 TLBs also suffer from poor utilization because chip-wide TLB resources are partitioned statically (usually equally) at design time. But this means that there are situations where, at runtime, a private L2 TLB on one core may thrash while its counterpart on another core may experience far less traffic [34].

Recent work has evaluated the potential of shared last-level TLBs (which we call *shared L2 TLBs*) [34]. Shared L2 TLBs eliminate the redundancy of private L2 TLBs and seamlessly divide TLB resources to cores based on their runtime demands, overcoming the problem of poor utilization. Shared TLBs also offer implicit prefetching benefits; i.e., a thread on one core can demand (and hence prefetch) translations eventually required by threads on other cores. The original paper finds that shared TLBs eliminate as much as 70-90% of the misses suffered when using private L2 TLBs [34].

*We conservatively reduce TLB sizes to account for our interconnect area to ensure area-equivalence between a baseline design with per-core L2 TLBs and our approach with a shared last-level TLB.

B. Shared L2 TLB Hit Rate

Fig. 2 quantifies the benefits of shared L2 TLBs on an Intel Haswell system described in Section IV. Fig. 2 shows that shared L2 TLBs eliminate the majority of L2 TLB misses suffered by private TLBs. Note that for every one of our workloads, *the entire* private L2 TLB is used to store entries – that is, no translations are wasted. Furthermore, like prior work [4, 34], we found that private L2 TLB miss rates range from 5-18%. Naturally, the main reason these miss rates are harmful is the fact that each TLB miss is a long-latency event.

Generally, the higher the core count, the more effectively the shared L2 TLB eliminates private L2 TLB misses. Consider, for example, a situation with 4 cores, and one with 16 cores. If private TLBs are N entries, the 4-core case can replace the private L2 TLBs with a shared L2 TLB with 4×N entries. A 16-core case can realize a 16×N-entry L2 TLB instead. We are therefore able to eliminate the replication and utilization problems of private TLBs even more effectively at higher core counts. Workloads with notably poor locality of access (e.g., `canneal`, `gups`, and `xsbench`) are particularly aided by shared TLBs at higher core counts.

C. Shared TLB Access Time

One might expect the hit rate improvements of Fig. 2 to improve performance overall. However, TLB performance is influenced not just by hit rates, but also the following:

① **SRAM array lookup times:** L2 TLBs are typically implemented as SRAM arrays. Unfortunately, scaling SRAM arrays while ensuring fast access is challenging. We model SRAMs in TSMC 28nm technology node using memory compilers. Fig. 3 quantifies access latency scaling as a function of the number of entries in the array (all numbers are post-synthesis). A 1536-entry L2 TLB (the size of private L2 TLBs in Intel Skylake) takes 9 cycles, while a 32×1536-entry design takes close to 15 cycles to access. Replacing private TLBs with an equivalently-sized shared TLB means that the shared structure grows from a 12K-entry structure for 8 cores (8×1536 entries) to a 96K-entry structure for 64 cores (64×1536 entries), increasing lookup times by factors of 2-4× Ultimately, this high access latency – which worsens as we need larger shared TLBs for higher core counts – counteracts the benefits of higher hit rates.

② **Interconnect traversal times:** The original paper on shared TLBs focused on monolithic designs where the entire structure was placed at one end of the chip [34]. Naturally, this design exacerbated access times further, due to additional interconnect delays to access the shared TLB location. This was observed to counteract the benefits in some cases even for a 4-core system [34]. Higher core counts further worsen this delay. For

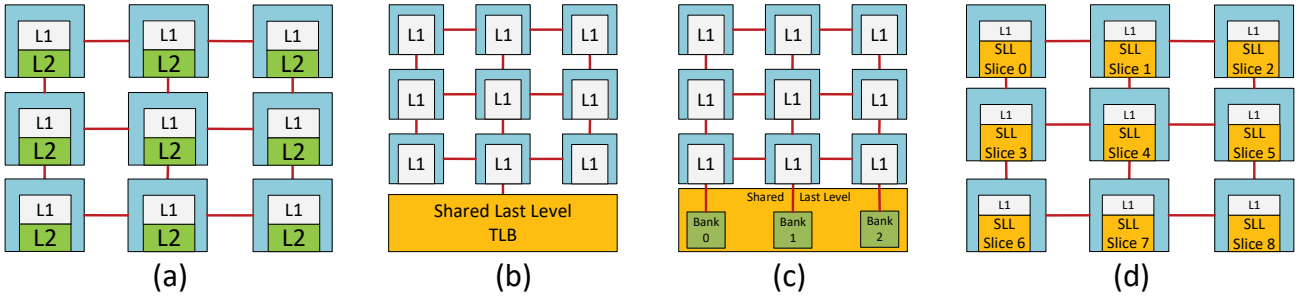


Fig. 1: *Last Level TLB Organization (a) Private, (b) Shared Last Level TLB - Monolithic, (c) Shared Last Level TLB - Banked, and (d) Shared Last Level TLB - Distributed across the cores*

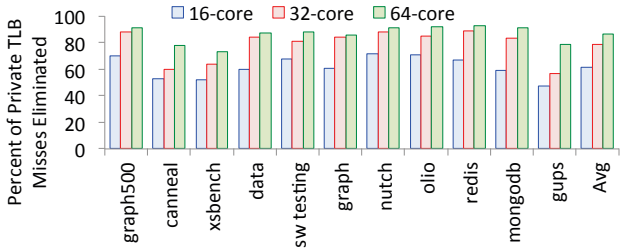


Fig. 2: *Percentage of private L2 TLB misses eliminated by replacing with a shared TLB. Results shown for 16-64-core systems.*

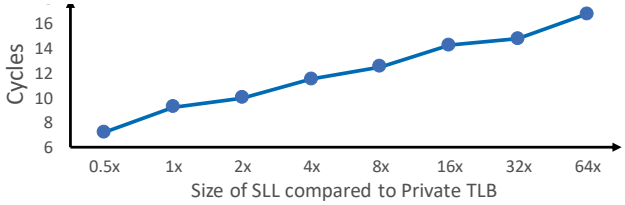


Fig. 3: *Access latency of SRAM TLB compared to number of entries in a TLB. Post-synthesis in 28nm TSMC PDK.*

instance, for a 64-core system, the tiles at the top of the chip would require 8 hops in each direction to access the TLB.

③ **Bandwidth:** A problem with the original shared TLB proposal is that accesses from multiple cores suffer from contention at the shared structure's access ports. While we will show that the likelihood of many concurrent TLB accesses is relatively low, it can still decrease performance versus the private L2 TLB scenario, where each core can access its private TLB without interference from other cores.

D. Shared TLB Performance

Fig. 4 quantifies how attributes ①-③ counteract higher hit rates in determining the overall performance of shared monolithic L2 TLBs. We profile performance on a 32-core Haswell system using monolithic shared L2 TLBs versus private L2 TLBs. Based on our SRAM array memory compiler studies with 28nm TSMC, we determine that the private L2 TLBs have 9-cycle lookup times. These are consistent with other references that measure Haswell TLB lookup times and Intel's product manuals, which state that private L2 TLB lookup latencies are 7-10 cycles [40]. For the shared L2 TLB,

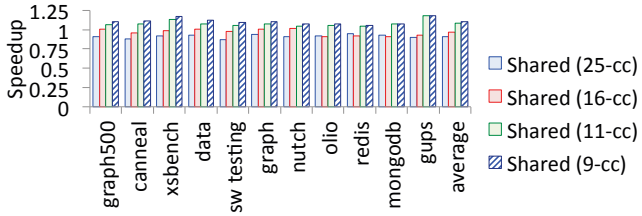


Fig. 4: *Speedups using shared multi-banked TLBs over private L2 TLBs. Shared TLB access latencies varies from 25 to 9 cycles.*

we vary the total access latency from 9 cycles (an unrealizable ideal case where the 32× larger SRAM array has access times that match the private L2 TLBs and the interconnect is zero-latency) to 25 cycles (a more reasonable estimate of the larger SRAM array plus interconnect latency). We bank the shared L2 TLB; we study designs with 16, 32, 64, and 128 banks. We plot results from the highest-performing banking configuration for each workload. Section IV describes the rest of the system configuration. Our experiments assume Linux 4.14 with support for transparent superpages [4, 5]. We find that over half of the memory footprint of the workloads are implemented as superpages (see Section V for more details).

Fig. 4 shows that despite better hit rates, the monolithic shared TLB can perform poorly. For example, at 25-cycle access latency, we see a 10-15% performance dip versus private L2 TLBs. Even worse, consider an unrealizable ideal network with zero interconnect latency (i.e., the only latency arises from port contention and SRAM array latency), which corresponds to the scenario where the shared L2 TLB access takes 16 cycles. Even here, the shared TLB shows little to no speedup over private L2 TLBs.

E. Understanding Shared L2 TLB Access Patterns

We now study key aspects of shared TLB access patterns that can help us overcome access latency problems.

Shared L2 TLB contention across applications. Fig. 5 captures information about contention at the shared L2 TLB. For every shared L2 TLB access, we plot the number of other cores with outstanding shared L2 TLB accesses. Fig. 5 shows that more than 40% of the L2 TLB accesses occur in isolation; i.e., there is no other outstanding TLB access. Roughly another

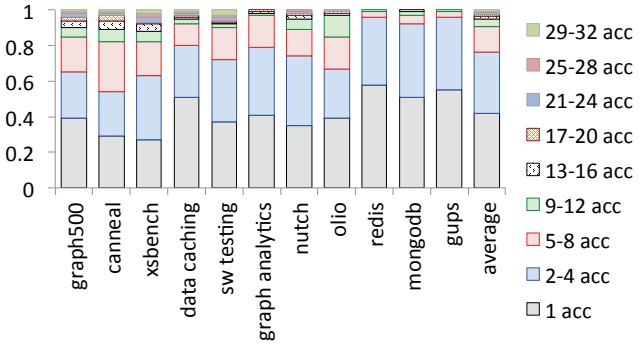


Fig. 5: *Fraction of L2 TLB accesses that occur concurrently with 1 other access, 2-4 other accesses, etc., on a 32-core Haswell system.*

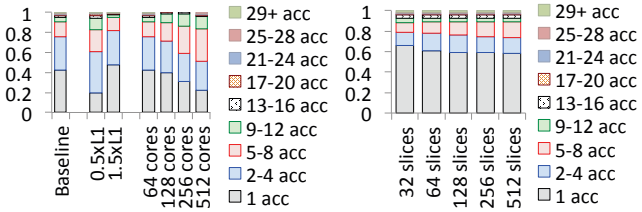


Fig. 6: *(Left) Fraction of L2 TLB accesses that occur concurrently with 1 other access, 2-4 other accesses, etc. Each bar averages results across workloads; (right) fraction of L2 TLB accesses to a TLB slice that occurs concurrently with 1 other access to that slice, 2-4 other accesses to that slice, etc. Each bar shows a distributed shared L2 TLB, where the number of TLB slices is equal to the number of cores.*

20-30% of the L2 TLB accesses occur when there are only 2-4 outstanding shared L2 TLB lookups.

Shared L2 TLB contention with varying L1 TLB size. The larger the L1 TLB, the fewer the shared L2 TLB accesses. Fig. 6 (left) shows the impact of the L1 TLB size on shared L2 TLB contention. The `baseline` bar matches the average access distribution from Fig. 5, while the `0.5×L1` and `1.5×` bars represent distributions as the private L1 TLBs per core are halved or increased by 50%. As one might expect, smaller L1 TLBs lead to more shared L2 TLB lookups. Consequently, the `2-4 access` and `5-8 access` portions of the bars increase significantly, implying greater contention. More interesting however are the trends towards bigger L1 TLBs as this reflects the direction processor vendors are going in. When we increase the L1 TLB sizes by 50%, we see contention dropping, with the `1 access` case dominating and taking up roughly 50% of the shared L2 TLB accesses.

Shared L2 TLB contention with varying core counts. Finally, Fig. 6 also shows the impact of core count on shared TLB contention. The `baseline` represents 32-core Haswell; 0.5×L1 and 1.5× are for 32-core Haswell with half and 1.5 times the baseline L1 TLB size. The `64-512 core` results assume 64- to 512-core Haswell systems and we expect shared L2 TLB contention to increase with a higher number of core counts. However, not only does contention not increase at 64 cores, it only marginally increases at 128 cores (i.e., the `5-8 accesses` and `9-12 accesses` contributions increase by roughly 10% and 5% respectively). Only when we begin to approach 256 cores and beyond does contention visibly increase. However, we have also performed experiments where we have replaced the monolithic (banked) shared L2 TLB with a distributed shared L2 TLB, where the number of L2 TLB slices equals the core count. The graph on the right in Fig. 6 showcases our results, this time quantifying the contention on average per TLB slice. As shown, even with high core counts (256-512 cores), roughly 60% of accesses to a single L2 TLB slice suffer no contention with concurrent accesses.

Takeaways. L2 TLBs must be accessed quickly for performance but *concurrent accesses are rare*. This is true not just for system configurations today, but would continue to remain true and in fact drop further in future systems with larger L1s or more cores. Later in Section V, we also validate this observation for a TLB miss "storm" microbenchmark (where we deliberately create high L1 TLB miss situations). This conceptual underpinning motivates our work - we design a specialized interconnect optimized for low latency rather than high bandwidth to accelerate shared L2 TLB access.

F. Low-Latency Interconnects

On-chip wire delay. As technology scales, transistors become faster but wires do not [41], making wires slower every generation relative to logic. This fact prompted research into NUCA caches [42, 43]. However, since clock scaling has also plateaued, wire delay in *cycles* remains fairly constant across generations. Long on-chip wires have repeaters at regular intervals, and take 75-100 ps/mm [41, 44, 45]. Thus **it is possible to perform a 1-cycle traversal across the chip in modern technology nodes**, as recent chips have demonstrated [44, 45].

NOC traversal delay. The network latency (T) of a message in modern NOCs is denoted as [46]:

$$T = H \times (t_r + t_w) + \sum_{h=1}^{H} t_c(h) + T_s$$

H is the number of hops required to reach the destination, t_r is the router delay, t_w is the wire delay, $t_c(h)$ captures the contention at each router, and T_s is the serialization delay incurred when sending a wide packet over narrow links. The latency is directly proportional to H.

Challenges with designing low-latency NOCs. It is usually difficult to build NOCs optimized for latency, bandwidth, area and power (see Table I). Buses do not scale and each traversal is a broadcast. Meshes are the most popular due to their simplicity and scalability, as they rely on a grid of short links with simple routers (with low t_r) at cross-points. However, the average hop count H (and therefore latency) is increased. High-radix NOC topologies (such as FBFly [47]) add long-distance links between distant routers, reducing H. However, these naturally add more links (i.e., bandwidth), leading to high area and power penalties from multi-ported routers and crossbars. If we use a narrower datapath (i.e., reduced bandwidth), we can reduce area and

TABLE I: *TLB Interconnect Design Choices.*

NOC	Latency	Bandwidth	Area	Power
Bus	✓	✗	✓	✗
Mesh	✗	✓	✗	✗
FBFly-wide [47]	✓	✓✓	✗✗	✗✗
FBFly-narrow	✗	✓	✗	✗
SMART [48]	✓	✓	✗	✗
NOCSTAR	✓	✓	✓	✓

power to that of a mesh, but serialization delay T_s leads to higher latencies. Optimizations such as SMART [48] fall in between these extremes by enabling packets to dynamically construct bypass paths over a mesh, reducing the effective H. However, the paths are not guaranteed, and require expensive control circuitry to setup and arbitrate for, leading to false positives and negatives [48]. Moreover, buffers at routers in a Mesh, FBfly and SMART add high area and power overheads. NOCSTAR proposes an interconnect with $t_r = 0$, $H = 1$, and $t_w = 1$, as we describe in the next section.

III. NOCSTAR DESIGN

Our approach, NOCSTAR, organizes the SLL TLB as a distributed array of TLB slices to reduce lookup latency, connected by a configurable single-cycle network fabric to reduce interconnect latency.

A. TLB Organization: Distributed TLB slices

NOCSTAR is a logically shared last level TLB distributed across the tiles of a many-core system, mirroring the design of NUCA LLCs [42]. Each slice is the equal or smaller than the size of current private L2 TLBs, thereby meeting the same area and power budgets.

- *TLB Entries:* Each entry in a slice includes a valid bit, the translation and a context ID associated with the translation.
- *Indexing:* Although optimized indexing mechanisms can be adopted for better performance, we use a simple indexing mechanism using bits from virtual address.

B. TLB Interconnect

We develop a dedicated side band NOC for communicating between the L1 TLBs and L2 TLB slices. Section II-F and Table I showed that directly adopting NOCs used between data caches today is not desirable for a TLB interconnect. Instead, we develop a latchless, circuit-switched interconnect that can provide single-cycle connectivity between arbitrary source-destination pairs.

1) Datapath: Latchless Switches: The datapath in NOCSTAR leverages the fact that wires are able to transmit signals over 10+ mm within a GHz (Section II-F). To enable single cycle traversal of packets in NOCSTAR, we add a *latchless switch* next to each L2 TLB slice as shown in Fig. 7(a). The switch is simply a collection of muxes (see Fig. 7(c)). The muxes are pre-set before a message arrives, as we will describe in Section III-B2. Fig. 7(b) shows a request arriving from the *West* direction traversing the switch and directly getting routed out of the *South* direction, as selected by the multiplexers, without getting latched. A message gets latched only at the destination switch where it needs to be ejected out to the target L1 TLB or L2 TLB slice. For example, an L1 TLB at the top left corner can send a request within one cycle to the L2 TLB slice at the bottom right, as Fig. 7(b) highlights. Each mux acts a like a *repeater*, and the entire traversal is similar to that of a conventional repeated wire [44, 45].

Bandwidth: This datapath is naturally lower bandwidth than a Mesh or FBFly as it does not have any buffers internally within the NOC. Moreover, unlike a FBfly which has more links, it cannot support multiple simultaneous transmissions unless they are using completely separate set of links. However, as we demonstrated earlier in Section II-E, L1 TLB misses are infrequent – there is only one access 60% of the time, and 1-4 accesses 80% of the time, making this low-bandwidth NOC sufficient for our purpose.

Scalability: Each traversal over this network takes a single-cycle. For large chips running at very high frequencies, this might be multiple cycles by adding pipeline latches as we discuss in Section III-B3.

2) Control Path: Fine-Grained Circuit-Switching: We now describe the steps involved in sending the messages.

Path setup. For each traversal through the interconnect, all data links in the path have to be *acquired* before sending any kind of message. To ensure that the packet reaches the destination in a single cycle, *all* links in the path must be acquired in the same cycle. This is done using separate control wires. Each data link has an associated arbiter which can allocate the link to one of the requesting cores. Fig. 8 shows an example of a core sending *requests* to all link arbiters in its path and receiving *grants* from each link arbiter before traversing the path. If any requester fails to acquire *all the links* in the desired path, because of any contention, it will wait and try again in the next cycle. This ensures that there are no packets traversing partial paths and thus avoids complexity. Once a path is *acquired*, the message can traverse through the datapath as shown in Fig. 7(b).

Fanout from switch. Each core has must have a way to setup a path to any of the slice present in the system. The width of the control wires for each arbiter depends on the routing policy adopted by the TLB system at design time. Consider an XY based policy in a system as shown in Fig. 7(d). Each core is connected to the arbiter associated with a link through which the core can send a request. Thus, the number of wires going out of each core is $(num\_cores\_each\_row - 1) + ((num\_rows - 1) \times (num\_columns))$.

Link arbiters. Each network link has an associated arbiter residing near the switch. The arbiter gets requests from any core which can send a TLB request/response packet through the link. This arbiter then selects one of the requesting cores and grants the link to it for the next cycle by setting the output mux to receive from the appropriate input port, and sending a 1-bit grant back to the requester, as shown in Fig. 8.

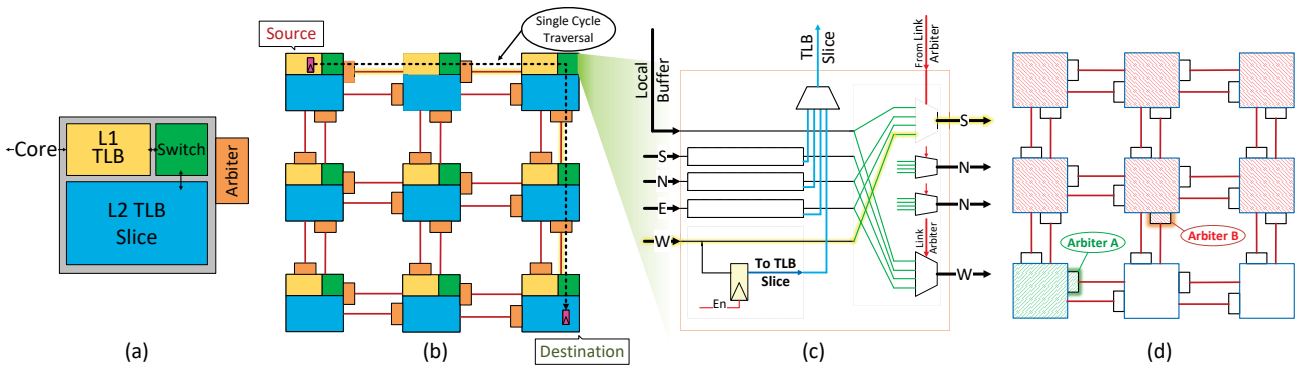


Fig. 7: *(a) TLB hierarchy near each core in* NOCSTAR. *(b) Source and destination of a request and the path of taken by the request. (c) Micro-architecture of the switch which enables single cycle traversal through the network. (d) Cores that can send requests to a given arbiter.*

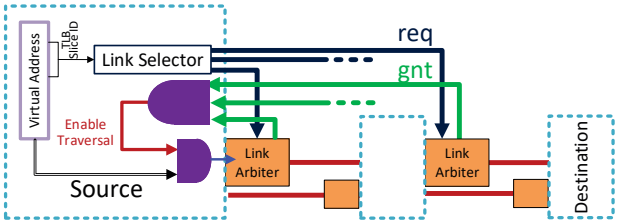
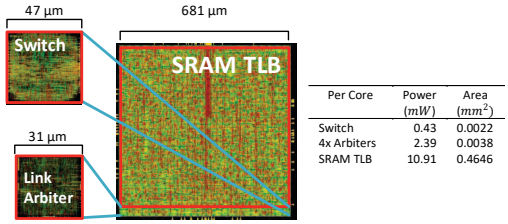


Fig. 8: *For setting up the path a core sends requests to all link arbiters in the path and waits for grants from them.*



Per Core	Power (mW)	Area (mm^2)
Switch	0.43	0.0022
4x Arbiters	2.39	0.0038
SRAM TLB	10.91	0.4646

Fig. 9: *Place-and-routed* NOCSTAR *tile in 28nm TSMC with the L2 TLB SRAM, switch and link arbiters highlighted and power/area of a switch and link arbiters for each slice in comparison to a SRAM based TLB slice. Target Clock Period = 0.5ns.*

Fanin at link arbiters. Depending on its physical location on-chip and the routing policy, different arbiters will have different number of requests coming in. For example, suppose we only allow XY routing. Fig. 7(d) shows that the green Arbiter A for an X link can only have one requester, while the red Arbiter B for a Y link has six possible requesters.

Arbitration priority. As the arbitration for each link is decentralized, there could be a possibility of livelock if two or more requests only acquire a partial set of links during each arbitration. To avoid this, the arbiters follow a static priority order among the requesters, to allot the links. In other words, a requester with higher priority will be guaranteed to get all its requested links. Further to avoid starvation, the static priority changes in a round-robin fashion every 1000 cycles.

3) Implementation: We implemented the NOCSTAR interconnect in TSMC 28nm with a 2GHz clock. Fig. 9 shows the place-and-routed design. We observe the following.

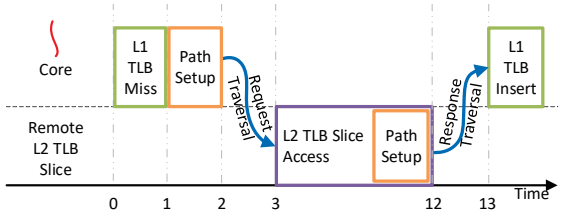


Fig. 10: *Timeline of a virtual address translation in case of an L1 TLB miss and remote L2 TLB access in* NOCSTAR.

Critical path. There are two sets of critical paths in the interconnect. On the *datapath*, a multi-hop traversal through all the intermediate switches needs to be performed within one clock cycle. Recall that the TLB interconnect is created at design-time. If timing is not met at the desired clock frequency, pipelined latches can be added at the maximum hops per cycle (HPC_{max}) [48] boundaries. This will increase the network traversal delay, but does not affect the operation of the design. Moreover, as core counts increase and tiles become smaller, the maximum hops per cycle will actually go up. On the *control path*, the critical path consists of the path setup request to the furthest link arbiter, link arbitration, and the grant traversal back to the core (Fig. 8). We observed that the place-and-route tool placed all the arbiters close to the center of the design to reduce the average wire lengths to meet timing.

Area and power. Fig. 9 shows the post-synthesis power and area consumed by the NOCSTAR switch and arbiter. We contrast it with the cost of the L2 TLB SRAM present in the same tile. The area consumed by switch and arbiter is less than 1% of the tile's L2 TLB SRAM. The link arbiters, due to high fanin and tight timing, are the most power hungry component and key overhead. We can reduce this overhead by restricting the routing algorithm (and correspondingly the fanin), as discussed earlier in Section III-B2.

C. Timeline of L2 TLB Access in NOCSTAR

Fig. 10 presents a timeline of address translation when there is an L1 TLB miss.

L1 TLB miss. The L1 TLB miss triggers a circuit-switched path setup. The path setup can be performed speculatively during the L1 TLB access as well.

Request path setup. The remote TLB slice to which the translation is mapped is identified by the indexing. A path setup request is then sent to the arbiters of the links in the path. The grants from all the requests are ANDed to determine if the full path was granted or not. If not, the path setup is retried. If the full path is granted, the request is sent out.

Request traversal. The TLB request is forwarded to the switch connected to the TLB slice (Fig. 7(a)). No header or routing information needs to be appended, since the path is already setup. The request takes a single-cycle through all the intermediate switches, and is latched at the remote TLB slice and enqueued into its request queue.

L2 TLB slice access. The remote TLB slice receives the request and services the request. The translation may either exist or not. If it is a TLB hit, a response should be sent. The response contains the physical page associated with the virtual address in the request. A TLB miss would lead to a page walk which is discussed in Section III-F.

Response path setup. A circuit-switched path for the response is requested. The response path can be setup speculatively, during the L2 TLB lookup, as a response will be sent to the requester regardless of access result.

Response traversal. The response traverses the TLB interconnect within a single-cycle.

L1 TLB insertion The requested translation is inserted into the requesting L1 TLB if it was a hit.

D. L2 TLB Access Latency and Energy

We quantify NOCSTAR's latency and energy benefits versus monolithic and distributed shared TLBs. Fig. 11(a) shows the latency of a message when traversing different number of hops through the TLB interconnect in the different shared last-level TLB designs. We consider two cases:

Case 1: The requested translation is indexed in the slice of the requesting core: The virtual address is used to index into the SLL slice in the local node and the translation is returned to L1 TLB. The total latency incurred is equal to $lookup\_latency$ of the TLB slice for both Distributed and NOCSTAR designs. This is identical to private last-level TLB latency.

Case 2: The requested translation indexes to a remote slice: The required translation request is sent to the remote node containing the slice through a dedicated network. Once it reaches the destination node, the virtual address is used to index into the SLL slice and the translation is then sent back to the requesting slice. Upon receiving the translation response, the requesting core can then forward the translation to the L1 TLB. The total latency in this case is $lookup\_latency + network\_latency$. Here, NOCSTAR provides a latency advantage over both Monolithic and Distributed. Even when the maximum hops per cycle HPC_{max} in NOCSTAR goes down, it is still much faster than the distributed case.

Fig. 11(b) shows the energy consumed by a message when traversing different number of hops through the TLB interconnect to understand trade-off spaces among the shared TLB designs. Most of the energy savings for the distributed design and NOCSTAR come from accessing a smaller SRAM structure than a monolithic(M) SLL TLB. Further, on the datapath, because of circuit switching, the energy consumed by an intermediate switch in NOCSTAR (N) is less compared to a switch in a traditional distributed network (D) with multi-cycle hops. However, NOCSTAR has a more expensive control path because of multiple request and grant wires spanning to all the link arbiters for simultaneous arbitration (Fig. 8). For instance, to traverse 14 hops within a cycle, NOCSTAR will require 14 links to be arbitrated for simultaneously. This shows up as a slightly higher control cost than Distributed. However, the latency gains from this approach leads to an overall energy savings, as we discuss in Section V.

E. Insertion/Replacement Policy

Like recent studies on TLB architecture, we assume that L1 and L2 TLBs use the lower-order bits of the virtual page number to choose the desired set using modulo-indexing, and use LRU replacement [4, 5, 7, 10, 11, 21, 24, 28]. Furthermore, like all recent work on two-level TLBs [4, 5, 7, 11, 34], we assume that the L1 and L2 TLBs are mostly-inclusive. Like multi-level caches, mostly-inclusive multi-level TLBs do not require back-invalidation messages [49].

F. Handling Page Table Walks

Suppose that a core suffers an L1 TLB miss and must look up the shared last-level L2 TLB. Suppose further that it determines that the TLB slice housing the desired translation lies on a remote node. If lookup of the remote node's TLB slice ultimately results in a miss, there are two options for performing the resulting page table walk. In the first option, the remote slice can send a *miss* message back to the requester node, which must now perform the page table walk. In the second option, the remote node can itself perform the page table walk. Both approaches have pros and cons. Handling page table walks at the remote node is attractive in that it eliminates the need for a *miss* message to be relayed between the remote and requester nodes. However, handling page table walks on the remote node also increases the potential for page table walker congestion; i.e., if multiple core's send requests to a particular remote slice and all of them miss, page table walks can be queued up.

G. TLB Shootdowns

A key design issue involves how NOCSTAR responds to virtual memory operations performed by the OS. In particular, consider a situation where a page table entry is modified by the OS on a particular core. When this happens, the OS kernel usually launches inter-processor interrupts (IPIs) that pause other cores and run an interrupt handler that "shoots down"

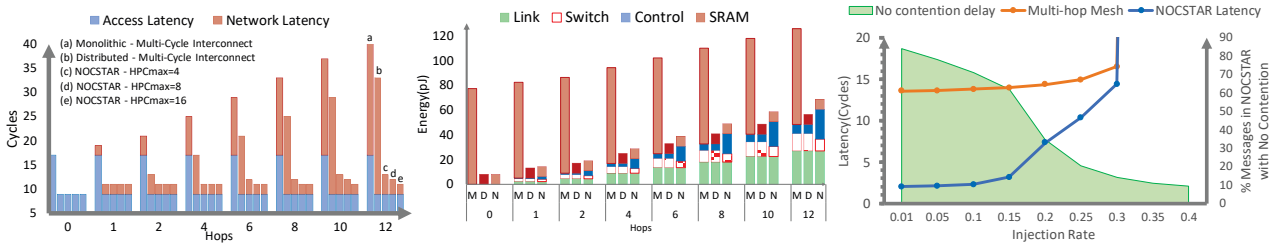


Fig. 11: *(a) Latency of each message in the TLB Interconnect in various configurations. (b) Energy consumed by each message in the TLB Interconnect in various configurations. (M)onolithic, (D)istributed, and (N)OCSTAR vs number of hops (c) Average latency of messages with respect to increasing injection rate in* NOCSTAR *interconnect compared to a multi-hop interconnect.*

or invalidates the stale translation in the TLB [35–37, 50]. This operation requires care in NOCSTAR – specifically, it is now possible that multiple cores simultaneously relay a translation invalidation signals to a single TLB slice that houses the stale translation. This can quickly congest the system by cascading TLB invalidation lookups of a single TLB slice.

We sidestep this by designating some node(s) as the *invalidation leader(s)*. In other words, even though *any* core can receive IPIs, and each core invalidates its private L1 TLB, only specific cores are permitted to then relay invalidation signals to the shared TLB. For example, if core 0 is considered the invalidation leader, *any core* that receives an IPI has to relay a message to core 0. Core 0 in turn relays a message to the relevant shared TLB slice to invalidate the stale translation. The actual TLB invalidation process for NOCSTAR from here on out mirrors that of a private L2 TLB. That is, during a private L2 TLB invalidation event, accesses to other translations in the private L2 TLB can be made; similarly, during the invalidation of a shared L2 translation, accesses to other translations (within the same slice or to other slices) are permitted. In Section V, we study our approach. The ideal scenario is a middle ground where the number of leaders is far fewer than the core count, but where it is not so small that the messages become congested at any particular leader core.

IV. METHODOLOGY

Simulation framework. We evaluate the benefits of NOCSTAR using an in-house cycle-accurate simulator based on Simics [51]. We model Intel Haswell systems [52] running Ubuntu Linux 4.14 with transparent superpages (which is the standard configuration). We model Intel Haswell cores with 32KB 8-way L1 instruction/data caches with 4 cycle access times, 256KB 8-way L2 caches with 12 cycle access times, and an LLC with 8MB per core and 50 cycle access times. These parameters are chosen based on Haswell specification parameters from the Intel manual [40, 52]. System memory is 2TB, with the workload inputs scaled so that each workload actually makes use of the full memory capacity.

Our cores maintain private L1 TLBs for different page sizes; i.e., 64-entry 4-way associative L1 TLBs for 4KB pages, 32-entry 4-way L1 TLBs for 2MB pages, and 4-entry TLBs for 1GB pages. As per Haswell specifications [40], our L1 TLBs are single-cycle and are accessed in parallel with the L1 caches using the standard virtually-indexed physically-tagged configuration [10]. All L1 TLBs have two read ports and a write port. Misses in the L1 TLB are followed by an L2 TLB lookup. Our baseline assumes the Intel Haswell configuration of private 1024-entry, 8-way associative L2 TLBs that can concurrently support 4KB and 2MB pages. In our studies, this baseline is 9 cycles based on post-synthesis SRAM numbers we generate, which also matches data from Intel manuals [40]. We vary this L2 TLB organization and latency; furthermore, we assume 2/1 read/write ports for each private L2 TLB and per shared L2 TLB slice. Our simulator models the L2 TLBs accesses as being pipelined, so one request can be serviced every cycle [10, 53, 54]. Finally, we combine our simulation framework with McPAT for energy studies [55].

Target configurations. Table II details the shared L2 TLB configurations that we evaluate. The first approach we evaluate is the standard `monolithic` approach posed in the original shared L2 TLB study [34]. We have evaluated several banking configurations for `monolithic` and settle on 4 banks for 16- and 32-core configurations, and 8 banks for 64 cores. We evaluate this with a regular mesh, and a single-cycle SMART NOC [48]. The second approach we study is a `distributed` approach where the shared L2 TLB is made up of an array of TLB slices placed near each core and connected by a NOC. We consider two different types of NOCs for shared distributed L2 TLBs: (a) *Mesh (Multi-hop):* This involves a traditional 1-cycle router coupled with 1-cycle link latency. To compete against a single-cycle-traversal-based NOCSTAR, we place enough buffers and links in the system to prevent link contention. Including any network contention may further degrade performance of workloads for traditional mesh networks. (b) NOCSTAR*:* A single cycle traversal if there is no contention; otherwise waits for another cycle as explained in Section III-B2; routing is XY-based. Our NOCSTAR evaluations assume that each core maintains a 920-entry (rather than a 1024-entry) shared TLB slice. This is a conservative area-normalized analysis, even though our interconnect consumes less than 1% area of each TLB slice.

Benchmarks. We use benchmarks from Parsec [56] and CloudSuite [57] for our studies. Furthermore, we study the performance of multi-programmed workloads by creating combinations of 4 applications. Each application in a multi-

TABLE II: *Major configurations of TLB that were simulated.*

	L2 TLB Entries (8-way associative)	Physical Org	Interconnect
Private	1024	1 TLB Per Core	-
Monolithic (Shared)	1024×NumCores	Monolithic	Mesh (Multi-Hop), SMART
Distributed (Shared)	1024×NumCores	1 slice Per Core	Mesh (Multi-Hop)
NOCSTAR	920×NumCores	1 slice Per Core	NOCSTAR

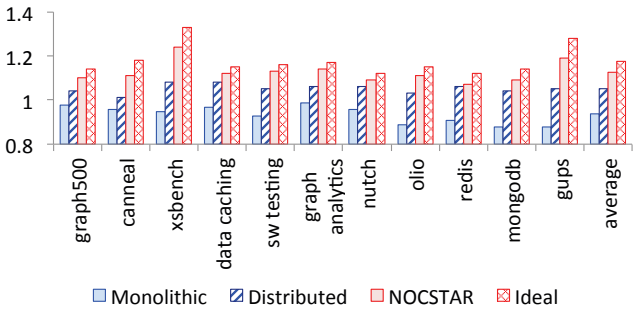


Fig. 12: *Speedups for monolithic, distributed, and* NOCSTAR *compared to ideal case with zero interconnect latency to the shared L2 TLB. Results assume 16-core Haswell systems using only 4KB pages.*

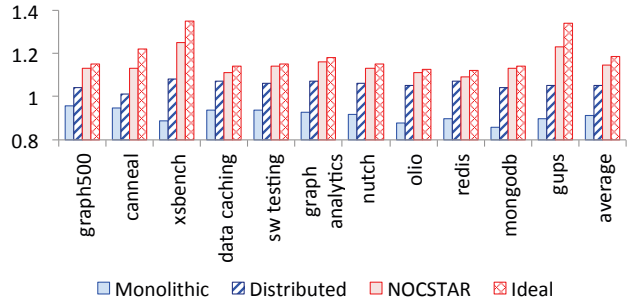


Fig. 13: *Complementary results to Fig. 12 but when Linux uses transparent superpages for a mix of 4KB and 2MB pages.*

programmed workload has 8 threads executing and scaled up to use 2TB of memory.

V. EXPERIMENTAL EVALUATIONS

Performance. Fig. 12 shows performance results for a 16-core Haswell configuration, assuming only 4KB pages. We plot speedups versus a baseline with private L2 TLBs; i.e., higher numbers are better (note that the y-axis begins at 0.8). Our `monolithic` data corresponds to a monolithic banked shared L2 TLB with access latencies determined from our circuit-level studies (see Section IV). We also show a `distributed` configuration as well an `ideal` case, where all shared TLB accesses have zero interconnect latency. Note that the `ideal` case does not imply an infinite TLB.

Fig. 12 shows that NOCSTAR achieves an average of 1.13× and a max of 1.25× the performance of private L2 TLBs. Importantly, this is better than any other configuration. In fact, `monolithic` degrades performance versus private L2 TLBs because of the perniciously high access latency. While

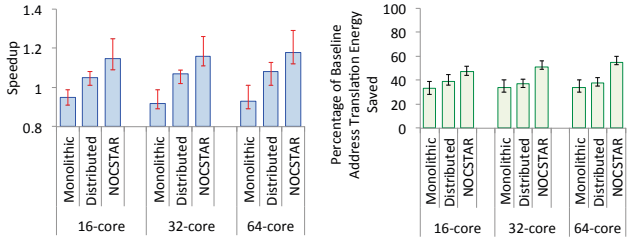


Fig. 14: *(Left) Speedups for varying core counts for Linux with transparent 2MB superpage support; and (right) percent of address translation energy saved versus private L2 TLBs.*

`distributed` partly helps, NOCSTAR achieves over 8% additional performance and comes within 2% of `ideal`.

Fig. 13 shows performance with Linux's native support for transparent 2MB superpages. We found that Linux was able to allocate 50-80% of each workload's memory footprint with superpages. One might expect superpages to reduce L1 TLB misses, reducing the gains from NOCSTAR. We find, however, *even better* performance with NOCSTAR in the presence of superpages. This is because the workloads are so memory-intensive (i.e., 2TB) that even with superpages, L1 TLB misses/shared L2 accesses are frequent. However, superpages do a good job of reducing shared L2 TLB misses, meaning that L2 TLB access times become a bigger contributor to overall performance. This explains why workloads such as `xsbench` and `gups` achieve large speedups of 1.2×+. NOCSTAR also outperforms `monolithic` and `distributed` with even larger margins than when simply using 4KB pages.

Scalability. The graph on the left in Fig. 14 quantifies speedups for varying core counts, when Linux supports transparent 2MB superpages along with 4KB pages. We show average, minimum, and maximum speedup numbers. In the `monolithic` case, high hit rates are overshadowed by high access times, particularly worsening performance at higher core counts. Employing a `distributed` approach helps, but NOCSTAR consistently outperforms other approaches.

Energy. Recent work shows that address translation can constitute as much as 10-15% of overall processor power and that the energy spent accessing hardware caches for page table walks is orders of magnitude more expensive than the energy spent on TLB accesses [58]. Using a shared TLB saves address translation energy by eliminating a large fraction of page table walks. Fig. 14 shows this, by plotting the percent of energy saved versus a baseline with private L2 TLBs. Even the `monolithic` approach eliminates roughly a third of address translation energy. However, NOCSTAR eliminates even more energy (as much as 60% on 64 cores). We have identified several reasons for these energy savings. One source is that NOCSTAR dramatically reduces runtime, thereby reducing static energy contributions of our system. Another important source of energy savings is that NOCSTAR reduces TLB misses and the ensuing page table walks. This means that cache lookups and memory references for the page table lookup are eliminated. In practice, like prior work [25, 29], we have found

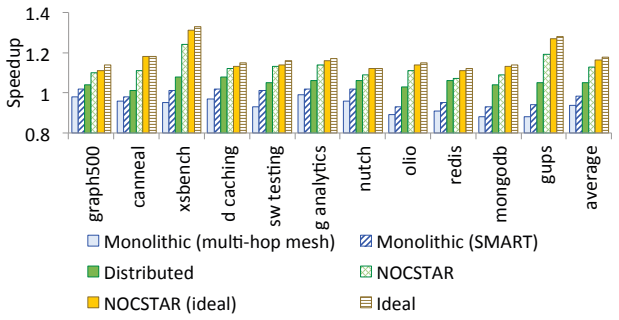


Fig. 15: *Speedup over baseline configuration with private L2 TLBs. We show two monolithic approaches (with traditional* `multi-hop mesh` *and* `SMART`*, as well as an* `ideal` NOCSTAR*, where we have no contention on the interconnect. We compare this to an* `ideal` *case where the TLB slices have zero interconnect latency.*

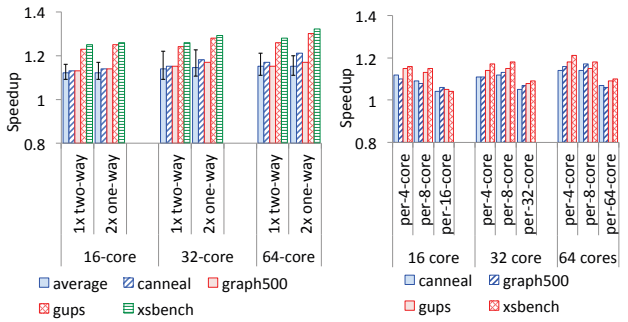


Fig. 16: *(Left) Speedups with varying core counts versus private L2 TLBs for round-trip acquire (*`1×two-way`*) and one-way acquire (*`2×one-way`*); and (right) speedups of TLB invalidation policies.*

that most page table walk memory references are serviced from the LLC. In our experiments on a baseline without NOCSTAR 70-87% of the page table walks in the workloads we evaluate prompt LLC and main memory lookups for the desired page table entry. Using NOCSTAR eliminates the bulk – over 85% on average – of the LLC/memory references, thereby saving lookup energy. These energy savings far outweigh the the energy overheads of the dedicated NOCSTAR network.

Interconnect. We now tease apart the performance contributions of distributing TLB slices versus a faster interconnect with Fig. 15. All bars represent speedups versus private L2 TLBs in a 32-core Haswell configuration. We show two versions of the banked monolithic approach, one with traditional multi-hop mesh, and one where we implement SMART with the monolithic approach. On average, both approaches suffer performance degradation; that is, even with a better interconnect (i.e., SMART), the monolithic approach experiences SRAM array latencies that are harmfully high. Instead, when we distribute the L2 TLB into slices per core (i.e., `distributed`), we achieve an average of 5% performance improvements. However, NOCSTAR performs even better.

Ideally, messages in NOCSTAR should take only 1 cycle to traverse the NOC. However, this number may increase because of contention for the path taken by the message. We find that on average, latencies are 1-3 cycles, with only two workloads – `xsbench` and `gups` – suffering latencies that can go beyond 3 cycles. Overall, this means that NOCSTAR achieves performance close to an idealized case, where the interconnect faces zero contention (represented by `NOCSTAR (ideal)` in Fig. 15. Finally, Fig. 15 also shows the achievable performance with an `ideal` scenario where the interconnect has zero latency. We see that NOCSTAR achieves within 95% of the performance of this idealized case.

To test the interconnect mechanism adopted in NOCSTAR, we injected random synthetic traffic to a 64 core system. Fig. 11(c) shows the average network latency faced by messages. Ideally messages in NOCSTAR would experience 1 cycle in path setup and another cycle to traverse the network. We see that even with an injection rate of 0.1 (1 message every 10 cycles per core, which is high for TLB traffic), the average latency of messages in the NOCSTAR interconnect remains within 3 cycles. Further, Fig. 11(c) also shows the percentage of messages which experience no delay in acquiring a path.

Path setup options. We study two modes of link reservation: (a) Round trip acquire: links are acquired for the total period of accessing a remote slice. In this mode, link selection has to be performed only once for sending a request and response. (b) One-way acquire: Links are acquired only for sending a one-way message. Each message in the system selects links before traversal. The graph on the left in Fig. 16 shows that acquiring links separately for each message delivers better performance than acquiring links for round trips.

TLB invalidation. We investigated the effect of sending an invalidate request to a TLB slice because of a shootdown or flush from any core. We considered various ways in which an invalidate message can be sent across a the TLB interconnect. The straightforward way is to send an invalidate from each core to the TLB slice. This policy is simple but may lead to congestion in the interconnect if all the cores are trying to invalidate from the same slice. The other way is to send the invalidate message to a central location which can then manage invalidations to all the slices. This can be further split up by having a manager for a set of n slices. The graph on the right in Fig. 16 shows the speedup of workloads with different ways of sending an invalidate message compared to each core sending its own invalidate message.

Page table walk policies. We considered two policies for performing page table walks:
Page table walk at the remote core: The core which has the L2 slice for the virtual address performs the page walk and then sends the new translation as a response to the requesting core after inserting it in the L2 slice.
Page table walk at the request core: On an L2 TLB slice miss, a *miss* message is sent to the requesting core. The requesting core then performs the page table walk and sends an *insert* message to the remote slice.

Fig. 17 shows speedups using policies. While performing the page table at the remote node involves sending fewer messages on the interconnect, it pollutes the local cache of the

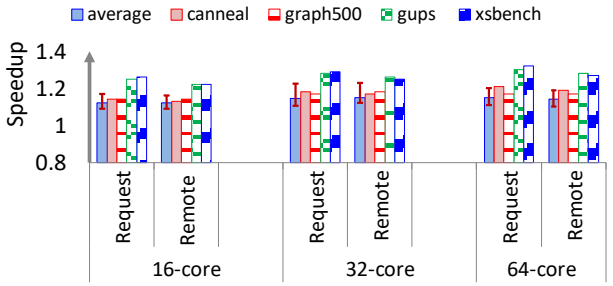


Fig. 17: *Page walks at requesting and remote core.*

TABLE III: *Speedups for a 32-core Haswell system. We study the impact of prefetching, hyperthreading, and page table walk latencies on the speedups achieved by* NOCSTAR *and other shared L2 TLB configurations versus private L2 TLBs. Speedup averages across workloads, as well as minima/maxima are shown.*

Pref.	SMT	PTW Lat.		Min	Avg	Max
No	1	Variable	Monolithic	0.89	0.92	0.99
			Distributed	1.02	1.07	1.09
			NOCSTAR	1.11	1.16	1.26
1	1	Variable	Monolithic	0.85	0.94	1.01
			Distributed	0.99	1.1	1.12
			NOCSTAR	1.08	1.2	1.29
1, 2	1	Variable	Monolithic	0.89	0.96	1.01
			Distributed	1.01	1.13	1.15
			NOCSTAR	1.1	1.25	1.32
1-3	1	Variable	Monolithic	0.87	0.89	0.99
			Distributed	0.99	1.08	1.11
			NOCSTAR	1.12	1.18	1.28
No	2	Variable	Monolithic	0.92	0.94	1.01
			Distributed	1.04	1.1	1.12
			NOCSTAR	1.14	1.21	1.31
No	4	Variable	Monolithic	0.93	0.95	1.03
			Distributed	1.01	1.13	1.15
			NOCSTAR	1.16	1.27	1.33
No	1	Fixed-10	Monolithic	0.84	0.88	0.93
			Distributed	0.94	0.95	0.99
			NOCSTAR	1.01	1.04	1.08
No	1	Fixed-20	Monolithic	0.89	0.92	0.99
			Distributed	1.02	1.07	1.09
			NOCSTAR	1.08	1.14	1.24
No	1	Fixed-40	Monolithic	0.93	0.97	1.03
			Distributed	1.05	1.09	1.13
			NOCSTAR	1.11	1.18	1.27
No	1	Fixed-80	Monolithic	1.05	1.08	1.12
			Distributed	1.08	1.13	1.17
			NOCSTAR	1.18	1.26	1.33

remote core (degrading performance). We see that performing the page table walk at requesting core delivers slightly better results compared to page table walk at remote core.

Sensitivity studies. We have quantified the NOCSTAR with other configurations (see Table III). The first row quantifies the average and min/max speedups for our workloads for a 32-core Haswell. We compare this to scenarios with prefetching (Pref. column label), with hyperthreading (SMT column), and with varying page table walk latency (PTW Lat. column).

We first compare these numbers to a scenario where TLB prefetching is enabled. The original shared TLB paper studied the impact of prefetching translations ± 1, 2, and 3 virtual pages adjacent to virtual pages prompting a TLB miss [34]. We run these experiments with our monolithic, distributed, and NOCSTAR configurations in rows 2-4. We find that NOCSTAR's benefits are consistently enjoyed even in the presence of prefetching. Like the original shared TLB paper, we find that prefetching translations for up to ± 2 virtual pages away is most effective, with more aggressive prefetching polluting the TLB. However, in every one of these scenarios, the shared L2 TLB's bigger size implies that there is less pollution versus private L2 TLBs. Additionally, NOCSTAR's reduced access latency versus the monolithic and distributed approaches means that accurate prefetching can yield better performance.

Table III quantifies the impact of running multiple hyperthreads. The more the number of hyperthreads run per core, the higher the TLB pressure. As expected, this means that shared L2 TLBs offer hit rate benefits over private L2 TLBs; when combined with NOCSTAR's superior access latency, the performance exceeds distributed and monolithic results.

Finally, Table III quantifies NOCSTAR's performance as a function of the page table walk latency. We classify page table walk latency as variable (corresponding to a realistic simulation environment where the page table walk latency depends upon where in the cache the desired translations reside) or fixed-N (where we fix the page table walk latency to N cycles). As expected, when the page table walk latency is unrealistically low (i.e., 10 cycles), the monolithic and distributed TLBs *severely harm* performance. This is because these configurations suffer higher access latencies, while their higher hit rates are not useful because the impact of a TLB miss is minor. Nevertheless, even in this situation, NOCSTAR outperforms private L2 TLBs. In more realistic scenarios where the page table walk latency is 20-40 cycles (which is what we typically find them to be on real systems [4, 24, 28, 34]), NOCSTAR's performance notably exceeds other options. And in scenarios where page table walks are very high (i.e., 80 cycles), these benefits become pronounced, with NOCSTAR outperforming distributed L2 TLBs by 13% on average.

Multiprogrammed combinations of sequential workloads. Our target platform is the 32-core Haswell system. Our workloads consist of combinations of four workloads, leading to 330 combinations overall. Each workload executes 8 threads to utilize all 32 cores. Fig. 18 sorts our results by overall IPC improvement. NOCSTAR is particularly effective for multiprogramming because it offers the utilization benefits of shared TLBs without penalizing applications with high access latency. So, it *always* improves aggregate IPC compared to the other approaches; in contrast, `monolithic` degrades performance for about half the workloads because of access latency issues while `distributed` degrades 10% of the workloads.

The bottom graph in Fig. 18 shows the speedup of the worst-performing application. As shown, `monolithic` and `distributed` see many cases (almost half the combina-

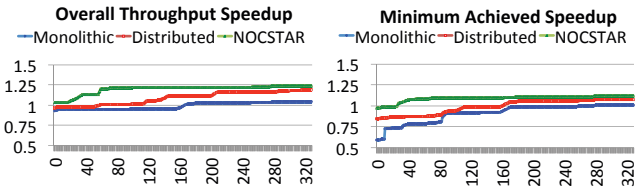


Fig. 18: *(Left) Overall throughput on 32 cores with 330 combinations of 4 workloads; and (Right) Speedup of the worst-performing sequential application over private L2 TLBs.*

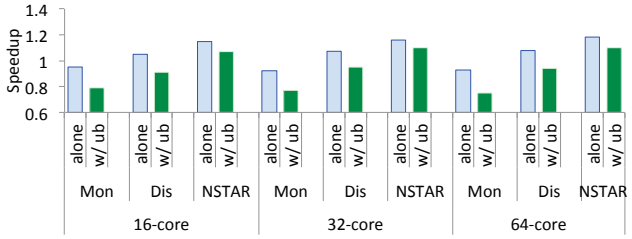


Fig. 19: *Average speedups for workloads versus private L2 TLB configuration, for varying core counts. Bars for* `alone` *represent results from when the workloads run alone (i.e., matching already-presented data). Bars for* `w/ub` *represent data for when the workloads were concurrently run with the TLB storm microbenchmark.*

tions) where at least one application suffers performance loss due to high access latency. Sometimes, degradation is severe; e.g., 40%. In contrast, only in 7% of the workloads does NOCSTAR degrade performance. Not only is this relatively rare, the extent of the performance loss is relatively benign, with worst cases of 2-3% versus private L2 TLBs. This problem is reminiscent of interference issues in LLCs and can likely be alleviated with LLC QoS/fairness mechanisms [59, 60]. We leave these for future work.

Pathological workloads. Our studies thus far suggest that most real-world workloads do not tend to generate significant congestion. For this reason, to stress-test NOCSTAR, we have devised two classes of microbenchmarks.

① TLB storm microbenchmark: The first microbenchmark triggers frequent context switches and page remappings. This forces "storms" of L2 TLB invalidations/accesses that congest the network. We take the workloads that we have profiled so far and we concurrently execute a custom-microbenchmark. We modify the Linux scheduler to context switch between our workloads and the microbenchmark; normally, Linux permits context switching at 10ms granularity, but we study unrealistically aggressive context switches from 0.5ms onwards for the purposes of stressing NOCSTAR. The custom microbenchmark is then designed to allocate 4KB pages, promote them to 2MB superpages, and then break then into 4KB pages again. The confluence of our modified Linux scheduler and our microbenchmark is a massive number of TLB misses and invalidations. Every context switch on our x86 Haswell systems forces all shared TLB contents to be flushed, followed by a storm of L2 TLB lookups for data. Furthermore, every time our microbenchmark promotes 4KB pages to a 2MB superpage, it invalidates 512 distinct L2 TLB entries.

Fig. 19 quantifies the slowdown of our workload with this TLB activity. Results are averaged across all workloads and we vary core counts. We focus on the case which generates the maximum network congestion by context switching at 0.5ms; our microbenchmark generates as many as 200-300 L2 TLB accesses per kilo-instruction, which is more than the TLB stressmarks in prior work [5, 28].

Fig. 19 shows that even the TLB pressure imposed by our microbenchmark naturally degrades performance versus the scenario where the benchmark is standalone (i.e., `alone`). As we can see, the `w/ub` results representing the microbenchmark suffer from as much as 10-20% performance degradation. However, *in ever single case*, NOCSTAR vastly outperforms the other approaches. For example, the `monolithic` banked L2 TLBs degrade performance by as much as 20-30% versus private L2 TLBs in the presence of this level of contention. On the other hand, NOCSTAR continues to achieve 7-11% performance improvements on average. While this is certainly lower than the 18%+ performance improvements achievable without congestion, these results are promising. Furthermore, the improvements achieved by NOCSTAR improve when we change our context switching granularity from an unreasonably aggressive 0.5ms to 1-10ms.

② TLB slice microbenchmark: We have also crafted a second microbenchmark to test what happens when there is immense congestion on one TLB slice. In this microbenchmark, we run *N-1* threads on our *N*-core machine. All these threads are designed to continuously access the L2 TLB slice assigned to the *Nth* core. Naturally, this approach degrades performance most severely. However, we find that in *every single case*, NOCSTAR continues to do better (by 3-5%) over private L2 TLBs. Furthermore, NOCSTAR is, in the most conservative scenario, 7% better than any other shared L2 TLB approach (i.e., either the monolithic banked, or distributed approaches). Consequently, NOCSTAR continues to be a better alternative than any other shared L2 TLB configuration.

VI. Conclusions

This study proposes NOCSTAR, a TLB-NOC co-design that achieves the high hit rates of shared TLBs without compromising access time. We show that the higher hit rate delivered by shared TLBs is overshadowed by the high latency posed by the TLB structure and the network involved in traversing to it. Moreover, a traditional distributed architecture does not deliver the potential performance gains because of network latency. By co-designing distributed TLBs with a SMART interconnect, NOCSTAR improves multi-threaded and multi-programmed workload performance.

VII. Acknowledgments

We thank Gabriel Loh and Jan Vesely for their feedback on improving early drafts of this work. We thank Google and VMware for their support in making this work possible. We thank Hyoukjun Kwon for providing scripts for synthesis and place-and-route of the SRAMs. Srikant Bharadwaj's work was partly supported by the DARPA CHIPS project.

References

[1] A. Bhattacharjee, "Preserving virtual memory by mitigating the address translation wall," in *IEEE Micro*, 2017.

[2] A. Bhattacharjee and D. Lustig, "Architectural and operating system support for virtual memory," in *Synthesis Lectures on Computer Architecture, Morgan Claypool Publishers*, 2017.

[3] M. Talluri and M. D. Hill, "Surpassing the TLB performance of superpages with less operating system support," in *ASPLOS*, 1994.

[4] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *ASPLOS*, 2017.

[5] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced large-reach TLBs," in *MICRO*, 2012.

[6] C. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations," in *ISCA*, 2017.

[7] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *HPCA*, 2014.

[8] Y. Marathe, N. Gulur, J. Ryoo, S. Song, and L. John, "CSALT: Context switch aware large TLB," in *MICRO*, 2017.

[9] J. Ryoo, N. Gulur, S. Song, and L. John, "Rethinking tlb designs in virtualized environments: A very large part-of-memory TLB," in *ISCA*, 2017.

[10] M. Parasar, A. Bhattacharjee, and T. Krishna, "SEESAW: Using superpages to improve VIPT caches," in *ISCA*, 2018.

[11] B. Pham, J. Vesely, G. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *MICRO*, 2015.

[12] M. Talluri, S. Kong, M. Hill, and D. Wood, "Tradeoffs in Supporting Two Page Sizes," in *ISCA*, 1992.

[13] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," in *OSDI*, 2002.

[14] Y. Kown, H. Yu, S. Peter, C. Rossbach, and E. Witchel, "Coordiated and Efficient Huge Page Management with Ingens," in *OSDI*, 2016.

[15] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ISCA*, 2013.

[16] J. Gandhi, A. Basu, M. Hill, and M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *MICRO*, 2014.

[17] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *ISCA*, 2015.

[18] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. Hill, K. McKinley, M. Swift, and O. Unsal, "Range translations for fast virtual memory," in *MICRO Top Picks*, 2016.

[19] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB improvements for chip multiprocessors: Inter-core cooperative tlb prefetchers and shared last-level TLBs," in *ACM TACO*, 2013.

[20] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *PACT*, 2009.

[21] A. Bhattacharjee and M. Martonosi, "Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors," in *ASPLOS*, 2010.

[22] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB preloading," in *ISCA*, 2000.

[23] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for TLB prefetching: an application-driven study," in *ISCA*, 2002.

[24] A. Bhattacharjee, "Translation-triggered prefetching," in *ASPLOS*, 2017.

[25] T. Barr, A. Cox, and S. Rixner, "SpecTLB: A mechanism for speculative address translation," in *ISCA*, 2011.

[26] A. Bhattacharjee, "Breaking the address translation wall by accelerating memory replays," in *IEEE Micro Top Picks*, 2018.

[27] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for high performance address translation in chip multiprocessors," in *MICRO*, 2010.

[28] A. Bhattacharjee, "Large-reach memory management unit caches," in *MICRO*, 2013.

[29] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *ISCA*, 2010.

[30] B. Pichai, L. Hsu, and A. Bhattacharjee, "Address translation for throughput oriented accelerators," in *IEEE Micro Top Picks*, 2015.

[31] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPU," in *ASPLOS*, 2014.

[32] J. Power, M. Hill, and D. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *HPCA*, 2014.

[33] S. Shin, G. Cox, M. Oskin, G. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular GPU applications," in *ISCA*, 2018.

[34] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *HPCA*, 2011.

[35] M. Kumar, S. Maass, S. Kashyap, J. Vesely, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy translation coherence," in *ASPLOS*, 2018.

[36] B. Pham, D. Hower, A. Bhattacharjee, and T. Cain, "TLB shootdown mitigation for low-power many-core servers with L1 virtual caches," in *Computer Architecture Letters*, 2018.

[37] Z. Yan, J. Vesely, G. Cox, and A. Bhattacharjee, "Hard-

ware translation coherence for virtualized systems," in *ISCA*, 2017.
[38] J. Vesely, A. Basu, M. Oskin, G. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *ISPASS*, 2016.
[39] A. Bhattacharjee, "Large-reach memory management unit caches," in *MICRO*, 2013.
[40] Intel, "Intel 64 and IA-32 architectures optimization reference manual," 2016.
[41] R. Ho, K. Mai, and M. Horowitz, "Managing wire scaling: a circuit perspective," in *Proceedings of the IEEE International Interconnect Technology Conference (IITC)*, 2003.
[42] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009.
[43] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS*, 2002.
[44] C. H. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan, and L. S. Peh, "Smart: A single-cycle reconfigurable noc for soc applications," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013.
[45] Y. H. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2016.
[46] N. D. E. Jerger, T. Krishna, and L. Peh, *On-Chip Networks, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.
[47] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: A cost-efficient topology for high-radix networks," *ISCA*, 2007.
[48] T. Krishna, C. H. O. Chen, W. C. Kwon, and L. S. Peh, "Breaking the on-chip latency barrier using smart," in *HPCA*, 2013.
[49] A. Jaleel, E. Borch, M. Bhandaru, S. Steely, and J. Emer, "Achieving non-inclusive cache performance with inclusive caches temporal locality aware (TLA) cache management policies," in *MICRO*, 2010.
[50] J. Vesely, A. Basu, A. Bhattacharjee, G. Loh, M. Oskin, and S. Reinhardt, "Generic system calls for GPUs," in *ISCA*, 2018.
[51] WindRiver, "Wind River Simics product note," 2015.
[52] P. Hammarlund, "4th generation Intel x2122; core processor, codenamed Haswell," in *IEEE Hot Chips Symposium (HCS)*, 2013.
[53] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying memory ordering at the hardware-OS interface," in *ASPLOS*, 2016.
[54] D. Lustig, G. Sethi, A. Bhattacharjee, and M. Martonosi, "Transistency models: Memory ordering at the hardware-OS interface," in *IEEE Micro Top Picks*, 2017.
[55] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.
[56] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *PACT*, 2008.
[57] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *ASPLOS*, 2012.
[58] V. Karakostas, J. Gandhi, A. Cristal, M. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Energy-efficient address translation," in *HPCA*, 2016.
[59] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grained partitioning," in *ISCA*, 2011.
[60] H. Cook, M. Moreto, S. Bird, K. Dao, D. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *ISCA*, 2013.