2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)

Enhancing Server Efficiency in the Face of Killer Microseconds

Amirhossein Mirhosseini Akshitha Sriraman Thomas F. Wenisch

University of Michigan
{*miramir,akshitha,twenisch*}*@umich.edu*

***Abstract*—We are entering an era of "killer microseconds" in data center applications. Killer microseconds refer to μs-scale "holes" in CPU schedules caused by stalls to access fast I/O devices or brief idle times between requests in high throughput microservices. Whereas modern computing platforms can efficiently hide ns-scale and ms-scale stalls through micro-architectural techniques and OS context switching, they lack efficient support to hide the latency of μs-scale stalls. Simultaneous Multithreading (SMT) is an efficient way to improve core utilization and increase server performance density. Unfortunately, scaling SMT to provision enough threads to hide frequent μs-scale stalls is prohibitive and SMT co-location can often drastically increase the tail latency of cloud microservices.**

In this paper, we propose *Duplexity*, a heterogeneous server architecture that employs aggressive multithreading to hide the latency of killer microseconds, without sacrificing the Quality-of-Service (QoS) of latency-sensitive microservices. Duplexity provisions *dyads* (pairs) of two kinds of cores: *master-cores*, which each primarily executes a single latency-critical *master-thread*, and *lender-cores*, which multiplex latency-insensitive throughput threads. When the master-thread stalls, the master-core borrows *filler-threads* from the lender-core, filling μs-scale utilization holes of the microservice. We propose critical mechanisms, including separate memory paths for the master-thread and filler-threads, to enable master-cores to borrow filler-threads while protecting master-threads' state from disruption. Duplexity facilitates fast master-thread restart when stalls resolve and minimizes the microservice's QoS violation. Our evaluation demonstrates that Duplexity is able to achieve $1.9\times$ higher core utilization and $2.7\times$ lower iso-throughput 99th-percentile tail latency over an SMT-based server design, on average.

I. Introduction

We are entering the "killer microsecond" era in data center applications [1]. Due to advances in processor, memory, storage, and networking technologies, events that stall execution increasingly fall in a microsecond-scale latency range. Accesses to emerging storage-class memories [2–9], rack-scale memory disaggregation [10–14], 100+ gigabit network communication [15], and accelerator/GPU micro-offloads [16–18] are example program activities that incur microsecond delays.

Lower latencies make it possible for data center architects to decompose monolithic applications into a collection of loosely-coupled microservices that interact over high-speed I/O to improve isolation, scalability, and maintainability [19]. Many cloud-based companies, including Amazon [20], Netflix [21], Gilt [22], LinkedIn [23], and SoundCloud [24] have adopted microservice architectures. Example microservices include content caching [25, 26], protocol routing [27, 28], key-value lookup [29, 30], query rewriting [31], or other steps performed across various application tiers [32]. Mid-tier microservices are particularly interesting objects of study since (1) they deal with both incoming and outgoing requests, (2) they must manage fan-out to leaf nodes and wait for the responses, and (3) their computation typically takes only a few microseconds, which is often shorter than the delay waiting for leaves to respond [33]. However, as a consequence of shorter service times and higher throughputs, idle periods between requests also shrink to microsecond scales, even under moderate load.

Whereas contemporary computing systems are effectively equipped with mechanisms to hide nanosecond- and millisecond-scale stalls, they lack efficient support for microsecond-scale stalls [1]. Nanosecond-scale stalls are effectively hidden by microarchitectural mechanisms, such as Out-of-Order (OoO) execution and deep memory hierarchies, but these mechanisms are insufficient to hide microsecond-scale stalls. Conversely, operating systems use context switching to hide millisecond-scale latencies, such as when accessing disk. However, context switch overheads (5-20μs [34, 35]) are within the same latency orders as microsecond-scale stalls, so they are not a plausible latency-hiding technique for the microsecond regime.

Total cost of ownership (TCO)-conscious data center operators try to maximize performance per dollar by maximizing performance density and energy efficiency (throughput per unit area/power) [36–38]. Cycles wasted on microsecond-scale stalls or idle periods erode execution efficiency and increase TCO. User-facing workloads, such as web search, have strict latency objectives and time-varying load [38], thereby imposing the same characteristics to their underlying microservices. Nonetheless, data centers have myriad latency-insensitive scale-out applications (e.g., offline graph analytics) that can be flexibly scheduled to fill utilization holes during off-peak loads. Thus, a common way to improve server utilization is to co-locate latency-critical and batch workloads, allowing them to share resources [39–43].

Simultaneous multithreading (SMT) has been proposed to co-locate latency-critical and batch threads on the same core so that the batch threads fill the utilization holes caused by brief I/O stalls or inter-request idle periods [44, 45]. Already today, scale-out workloads deployed in data centers exhibit low CPU utilization due to lack of memory level parallelism and front-end inefficiencies, calling for more SMT threads even in the absence of microsecond-scale stalls [46, 47]. As batch workloads also adopt mechanisms like storage-class memory or rack-scale disaggregation, these workloads, too, will incur such stalls. As a consequence, even more SMT threads must be added to ensure that, at any time, there are enough unstalled threads to fill a core's available execution bandwidth—the two threads offered by Intel's Hyper-Threading are not nearly enough.

Unfortunately, scaling SMT microarchitecture to support many more threads is prohibitive, due to high logic complexity, wire delay, limited register file (RF) capacity, and cache pressure/thrashing among threads. Moreover, as previous studies have shown [39, 45, 48], some SMT thread co-locations can have catastrophic impact

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00037

on the tails of latency-critical threads, especially at high loads, due to contention for shared resources. To avoid compromising the tail latency of critical threads due to SMT interference, we instead design *Duplexity*, a server architecture that seeks directly to address the killer-microsecond challenge—to fill in the microsecond-scale "holes" in threads' execution schedules, which arise due to idleness and stalls, with useful execution, without impacting the tail latency of latency-critical threads.

Duplexity is a heterogeneous server architecture that comprises two kinds of cores: *master-cores*—optimized for latency-sensitive microservices, and *lender-cores*—optimized for latency-insensitive throughput applications, which are arranged in pairs called *dyads*. Duplexity addresses microsecond-scale stalls by allowing master-cores to borrow threads from the lender-core in their dyad. Master-cores build on the concept of morphable cores [49] to switch between a single-threaded dynamically scheduled execution mode (when running the latency-critical *master-thread*) and a multi-threaded mode with in-order issue per thread (to fill in idle/stall periods with *filler-threads*). A key novel aspect of Duplexity is protection of the master-thread's micro-architectural state to maintain its QoS—filler-threads do not disrupt the caches, branch predictor, and other state held by the master-thread. When the master-thread becomes ready, Duplexity rapidly evicts filler-threads and grants the master-thread exclusive use of the master-core.

Lender-cores employ a Hierarchical Simultaneous Multithreading (HSMT) architecture to maintain a backlog of *virtual contexts* that time-multiplex the lender-core's physical hardware contexts, and from which the master-core may borrow. We develop new mechanisms to support rapid transfer of virtual contexts into and out of the master-core. Overall, we seek to maximize performance density and energy efficiency (by increasing filler-thread throughput) while giving the master-thread nearly the performance it would enjoy running alone. Such a cooperative composition of cores yields Duplexity, a unique server architecture that is well-suited to the killer-microsecond era.

Our evaluation demonstrates that Duplexity can improve core utilization by $4.8\times$ and $1.9\times$, and iso-throughput 99th-percentile tail latency by $1.8\times$ and $2.7\times$, on average, over a baseline OoO and an SMT-based server architecture, respectively. Duplexity is the first server architecture that aims to improve server utilization in the presence of microsecond-scale stalls and idle periods, without sacrificing QoS and tail latency of microservices. In summary, we make the following contributions:

- We quantitatively explore the killer-microsecond challenge as it relates to the load of latency-sensitive microservices and show that microsecond-scale stalls arise due to both fast communication and brief idle periods.
- We show that conventional SMT is not a satisfactory solution as it may drastically harm tail latencies of microservices and cannot be scaled to hide microsecond-scale stalls.
- We propose Duplexity, a server architecture comprising highly multi-threaded and morphable cores that can borrow threads to recover cycles lost to microsecond-scale stalls and idle periods while providing isolation mechanisms to preserve QoS of latency-critical microservices.
- We compare Duplexity to other server designs. Existing alternatives either compromise tail latency of microservices or fail to fully recover the cycles lost to microsecond-scale stalls.

II. Motivation and Background

We first motivate the problem space Duplexity seeks to address.

A. Killer Microseconds

With the advent of low-latency I/O in modern data centers, applications increasingly access data with single-digit microsecond latencies. For example, with state-of-the-art data center networking, a network round-trip at 40 Gbps can take only 2-4 μs [1]. At such latencies, RDMA-based disaggregated-memory systems [10–14] are expected to provide μs-scale remote memory accesses. Similarly, emerging memory technologies, such as 3D XPoint, have comparable access latencies [50]. Intel Optane SSD is an example low-latency storage device that builds upon 3D XPoint and enables 7-15 μs random block access [51]. At the higher-end of the microsecond spectrum, raw Flash can be accessed within tens of microseconds [52]. Fine-grain GPU/accelerator micro-offloads have similar latencies [16–18]. As a consequence, μs-scale stalls are quickly becoming a primary latency bottleneck, particularly as microservices replace monolithic data center applications.

Modern microarchitectures effectively hide ns-scale stalls caused by events like cache misses using instruction level parallelism and deep memory hierarchies. Modern operating systems effectively hide ms-scale stalls caused by events like disk I/O accesses through context switches. However, existing mechanisms do not effectively hide μs-scale stalls or idle periods. Single-threaded deep speculation mechanisms like branch prediction and runahead execution [53] are not accurate enough to fill more than 10s of nanoseconds with future instructions from a stalled thread, and are inapplicable to fill idle periods. Similarly, prefetching techniques are at best able to hide the latency of cachable memory accesses (rather than general μs-scale I/O) and are not applicable to idle periods [54–59]. Context switches themselves incur μs-scale overheads [34, 35], and are too expensive to amortize μs-scale stalls. Moreover, modern low-latency communication mechanisms rely on OS bypass interfaces (precisely to avoid latency of deep OS software stacks and their attendant caching inefficiencies) and hence are OS-transparent [12, 60–65]. Alas, current warehouse-scale computers resort to spinning to maintain low latency despite μs-scale stalls, wasting these CPU cycles. It is for this reason that Google recently coined the term *killer microseconds* [1].

Killer microseconds due to stalls. We reason quantitatively about μs-scale stalls using a simple model. We consider a single-job closed-loop model representing a period of computation leading to a μs-scale stall event, such as a disaggregated memory access. The modeled system alternates between periods of computation and stalls. During stalls, CPU time is wasted, reducing utilization.

Figure 1(a) illustrates the utilization loss as we vary the length of stalls and the computation time between them. When stalls are short (left edge of front axis), utilization converges to 100%. So, for example, a DRAM-scale stall every few microseconds sacrifices an insignificant fraction of utilization. Correspondingly, when the computation interval between stalls is large (far edge of right axis), stalls reduce utilization only gradually. However, when stalls and computation periods are of a similar order, utilization drops precipitously, rapidly dropping towards 0% if stalls exceed the average computation interval (near corner). This model implies that, as the distance between μs-scale stalls shrinks, a worrying fraction of CPU time may go to waste. Such compute-to-communication ratios

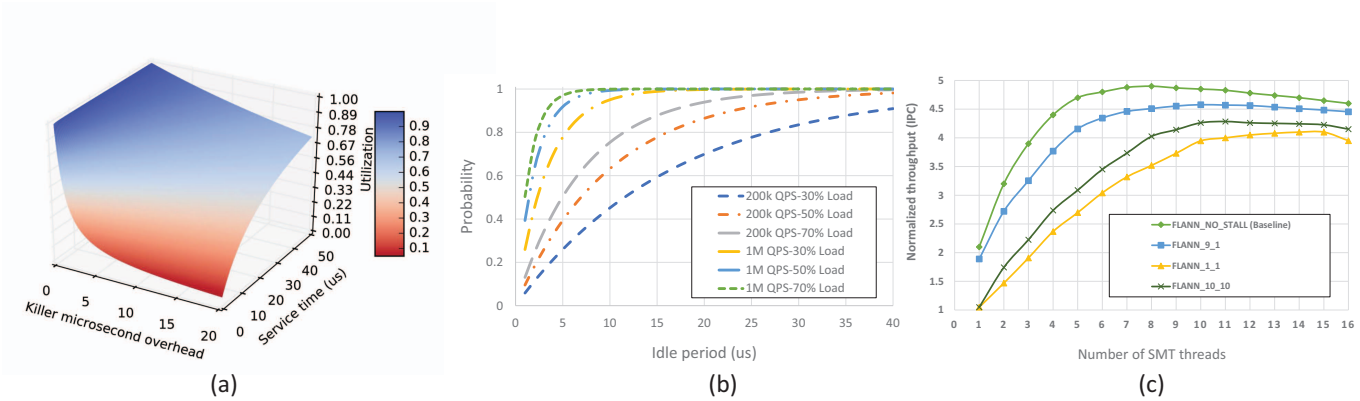

Figure 1. (a) Effect of μs-scale stalls on a closed-loop system, (b) Cumulative distribution of idle periods across various loads and service rates in an M/G/1 server, and (c) Throughput when varying the number of SMT threads for the FLANN workload on a 4-wide OoO core.

are already being seen in mid-tier microservices that accept service-specific queries, fan them out to leaf microservers that perform relevant computations on their respective data shards, and then return the aggregated results [33].

Killer microseconds due to idleness. Utilization losses further mount due to idleness for microservices operating in the microsecond regime, even at moderate offered loads. To avoid long queuing delays, interactive services typically operate at 30-70% capacity [38]. Hence, idle periods are inherent. In ms-scale requests of monolithic services, idle periods correspondingly occur at ms-scale (e.g., around 1ms for a web search leaf service with a median service time around 4ms [66]). However, as faster I/O enables faster microservices, idle period time scales also shift.

Most cloud applications exhibit high service time variability and heavy-tailed service distributions [67, 68]. However, due to the memory-less property of Poisson request arrivals, idle periods of all M/G/1 queuing systems follow an exponential distribution, independent of the service distribution [69]; idle period duration is only a function of service rate and load. Figure 1(b) depicts the cumulative distribution of idle-period durations for M/G/1 microservices serving 200K and 1M Queries-per-Second (QPS) at offered loads of 30%, 50%, and 70% of capacity. As can be seen in the Figure, individual idle periods last only a few microseconds. For example, 200K and 1M QPS services at 50% load average idle periods of only 10μs and 2μs, respectively, despite being idle half the time. Existing mechanisms cannot exploit such short idle periods (indeed, they are too short even for hardware power management [66, 70, 71]).

B. Simultaneous Multithreading

Software/OS-based multi-threading is the typical approach for hiding millisecond-scale I/O stalls and improving resource utilization when a thread is blocked or idle for lack of work. However, software multi-threading is too coarse-grained to react at the microsecond timescales of microservices. Simultaneous Multi-threading (SMT), wherein each core multiplexes instructions from multiple hardware contexts, can increase instruction throughput and improve resource utilization. Several prior works use SMT to improve server utilization [44, 45] and SMT is widely reported to be enabled in modern data centers [47].

Not enough SMT threads. Unfortunately, the number of SMT threads supported by contemporary processors (typically, 2-4), is far too few to effectively hide frequent μs-scale stalls. To demonstrate that such stalls call for far more SMT threads, we consider a microservice benchmark based on FLANN [72], an open-source library for performing fast approximate nearest neighbor searches in high-dimensional spaces. FLANN uses Locality Sensitive Hashing (LSH) to perform k-nearest neighbor identification—a critical microservice employed in content-based similarity search. After an LSH lookup, the benchmark issues accesses to remote memory to retrieve remote objects indicated by the lookup. We modify the gem5 [73] simulator to intercept the remote accesses issued by the FLANN microservice and stall execution. The modified simulator draws stall durations from an exponential distribution; we vary the mean stall duration as a parameter in our experiments.

The computation FLANN performs between remote accesses varies with the number of LSH tables, buckets, and probes used in FLANN's lookup operation. We use these tuning knobs to adjust the interval between remote accesses. By adjusting the mean of the stall duration distribution and the interval between stalls, we model various killer microsecond scenarios. We consider four compute-to-stall ratios (9:1, 10:10, and 1:1; all in microseconds; denoted as FLANN-X-Y) and a configuration that does not stall (baseline). Note that while single–cache-line (64B) RDMA accesses take roughly 1μs [15], since we investigate the impact of stall durations on performance, we assume stalls to take 10μs and zero-latency in two of our workloads. Since we seek to analyze throughput (rather than latency, QoS, or queuing delays), in this experiment, we model a saturated queue of requests (i.e., 100% load; no idle period between requests) to only stall for remote accesses.

We measure normalized throughput as a function of the number of SMT threads, from one to 16. Figure 1(c) illustrates the resulting throughput on a 4-wide OoO superscalar core (we scale only the number of threads; we do not scale microarchitecture resources except provisioning additional architectural registers). The purpose of this experiment is to identify how many SMT threads are needed to saturate the 4-way OoO core. In the baseline with no stalls, 8 threads saturate the pipeline; more threads degrade performance due to interference. However, the workload variants with μs-scale stalls require more threads before performance gains level off. For example, the FLANN-9-1 workload (representing a 1 μs stall every 10 μs, for a 90% effective utilization) peaks at 11 threads, while the FLANN-1-1 (50% effective utilization) peaks at 15.

More threads increase interference and hurt QoS. Co-running latency sensitive threads with others can severely degrade tail latency and violate QoS requirements due to interference and cache pollution effects, even with only two SMT threads [39, 45, 48]. Nevertheless, we show that simply adding more threads is not a satisfying solution to fill μs-scale stalls, even if the only objective is to maximize instruction throughput. Note that in Figure 1(c) all

three workloads with μs-scale stalls underperform the baseline. The frequent stalls in the these workloads result in more cache misses, as threads evict one another's data as their executions interleave. This effect is most apparent in the gap between the FLANN-10-10 and FLANN-1-1 workload: threads in both workloads are stalled 50% of the time, and yet the $10\times$ more frequent stalls of FLANN-1-1 lead to much lower total throughput. Further note that the peak performance of FLANN-1-1 (at 15 threads) lags the peak performance of the baseline (achieved at 8 threads) by 16%; adding more threads cannot recover the throughput lost in FLANN-1-1's μs-scale stalls.

Unfortunately, there are daunting impediments to scaling SMT threads per core. First, more threads add L1 cache pressure. Cache capacity cannot increase without affecting cache hit time, which penalizes single-thread performance. Second, adding threads complicates fetch/dispatch/issue logic, prolonging its critical path and lowering clock frequency. Finally, adding threads requires a larger register file to accommodate at least their architectural state. Again, scaling up this structure inevitably impacts wire delay and clock frequency—an effect we neglect in Figure 1(c) that would further exacerbate throughput loss. As a result, scaling SMT cores beyond 8 threads is ineffective for hiding μs-scale stalls, even when seeking only to maximize throughput.

III. Duplexity

We next present Duplexity—a server architecture that aims to fill in cycles lost to μs-scale stalls or idle periods while preserving tail latency and QoS. Duplexity comprises two kinds of cores: *master-cores*, optimized for latency-sensitive microservices and *lender-cores*, optimized for latency-insensitive scale-out (batch) applications. Duplexity addresses the killer microsecond challenge by borrowing "filler" threads from the lender-cores and executing them on the master-cores during the μs-scale "holes" arising from I/O stalls and idleness. To facilitate borrowing threads, master-cores and lender-cores are arranged in pairs, called 'dyads', with data paths that allow filler-threads running on the master-core to remotely access caches located at the lender-core. Master-cores build upon concepts from morphable cores [49], allowing them to morph between a single-threaded dynamically scheduled execution mode to execute their latency-sensitive *master-thread*, and a multi-threaded in-order execution mode to execute latency-insensitive filler-threads, borrowed from the lender-core. Lender-cores employ a Hierarchical Simultaneous Multithreading (HSMT) architecture, wherein they maintain a backlog of latency-insensitive threads that time-multiplex hardware contexts, from which the master-core may borrow. We integrate these concepts with efficient mechanisms to support rapid thread-context transfer into and out of the master-core and to protect the single latency-critical master-thread from interference by filler-threads. Our key objectives are (1) to fill in idle/stalled periods in the master-core with useful work from filler-threads, and (2) to minimize disruption, especially tail latency increases, of the master-thread.

There is a renewed interest in using simple, in-order cores for scale-out workloads [74–76]. However, simple cores incur a higher ratio of tail-to-average latency at large scales, and small configuration or parameter changes can result in large tail latency swings, making it difficult to achieve performance stability [48]. As such, latency-sensitive microservices with strict QoS targets are still typically run on OoO cores with advanced memory systems rather than a sea of scale-out-optimized simple cores [77, 78]. This dichotomy motivates the two operating modes of master-cores and our approach of coupling heterogeneous cores in dyads to facilitate thread borrowing.

When executing the master-thread, a master-core operates as an n-way OoO processor, with all execution resources dedicated to maximizing single-thread performance. However, whenever the master-thread becomes idle or incurs a μs-scale stall, the core's "morphing" feature is activated, which partitions the issue queue and register file and deactivates OoO issue logic to instead support InO issue of multiple filler-threads. The master-core then loads register state for these filler-threads from the lender-core's scheduling backlog and begins their execution. When the master-thread returns (stall resolves or new work arrives), it evicts the filler-threads, using hardware mechanisms that evacuate their register state as fast as possible. Minimizing performance disruption of the master-thread is challenging. In a key departure from prior work, we ensure that filler-threads cannot disrupt the cache state of the master-thread. We provision a path from the master-core's memory stage and front-end to the lender-core's caches; filler-threads access the memory hierarchy of the lender-core. Hence, when the master-thread returns, there is little evidence the filler-threads were ever there.

We first describe the microarchitecture of the lender-cores, as the master-core operates much like a lender-core when it operates in the multithreaded mode. We then explain our main contribution, the master-core, and how its microarchitecture enables adaptation between modes.

A. Lender-cores

The goal of lender-cores is two-fold: (1) support efficient multithreading for latency-insensitive scale-out workloads that nonetheless incur μs-scale stalls, and (2) lend threads to the master-core while it is stalling or idle. As we demonstrated in Section II-B, the key requirement to hide μs-scale stalls is to provision more threads from which the lender-core can schedule. However, if too many threads are co-scheduled on the core, they will interfere with each other and may hurt performance. Hence, we suggest a Hierarchical Simultaneous Multithreading (HSMT) architecture with two levels of virtual/physical contexts, similar to Balanced Multithreading [79] and two-level warp scheduling in GPUs [80, 81]. Lender-core's datapath resembles an SMT core that supports as many threads as physical contexts, and has similar area costs and clock frequency, but, when a thread occupying a physical context faces a μs-scale stall, its architectural state is swapped with a ready virtual context to improve utilization and throughput.

By limiting the number of active threads, the lender-core prevents performance degradation or diminishing returns (as observed in Figure 1(c)) due to interference of many threads, yet it virtually enables sufficient threads to hide μs-scale stalls. We find 8 threads as the sweet spot for the number of physical contexts for three main reasons: First, as we showed in Section II-B, the core's throughput saturates around 6-8 threads and may even drop beyond 8 threads in the absence of μs-scale stalls. Even with stalls, due to interference, the core is not able to match the throughput it achieves without stalls if many threads are co-scheduled. Second, as illustrated in Figure 2(a) and shown by prior work [49, 82, 83],

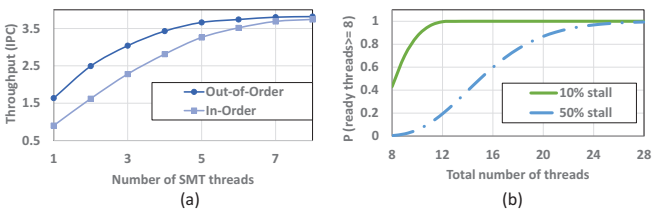

Figure 2. (a) Throughput of multi-threaded SPEC workload mixes for varying InO/OoO SMT threads on a 4-wide OoO core. (b) Probability of having at least 8 ready threads under varying thread counts and stall rates.

the gap between OoO and InO issue vanishes at ∼8 threads. Hence, we can employ an InO datapath to reduce precise state and pipeline flush complexity and avoid area/energy costs of OoO structures, especially since the lender-core targets only latency-insensitive threads. Finally, while up to 8-thread SMT designs are commercially available [84], building a core with more than 8 physical SMT contexts may be impractical due to logic/wire complexity and register file constraints.

We develop a simple analytic model to determine how many virtual contexts are needed to fill eight physical contexts as a function of the fraction of the virtual thread stall time. The distribution of ready threads is then given by a Binomial $k \sim \text{Binomial}(n, 1-p)$, where k represents the number of ready threads, n the number of virtual contexts, and p the probability a thread is stalled. We plot $P(k \geq 8)$ as a function of n for two stall probabilities in Figure 2(b). When threads are stalled only 10% of the time, 11 virtual contexts are sufficient to keep the 8 physical contexts 90% utilized. However, when threads are 50% stalled, 21 virtual contexts are needed. As a result, the number of required virtual contexts may be different depending on the workload.

A lender-core's microarchitecture is shown in Figure 3. The datapath is identical to an 8-threaded InO SMT. We note that this core is quite simple and area-efficient, since it does not require any OoO execution logic. The lender-core's front-end maintains a pointer to a FIFO run queue in dedicated memory, which holds the state of all virtual contexts. When a physical context stalls, its context is dumped to the tail of the run queue. Then, another context's architectural state is loaded from the run queue. The length of the run queue is not limited by hardware, as the number of required virtual contexts may vary. OS/cluster-level scheduling frameworks must provision enough threads to each lender-core to ensure the core is fully utilized and threads do not starve.

Master-cores borrow threads from a lender-core by stealing a virtual context from the head of its run queue. The master-core and lender-core in each dyad share the dedicated memory region where virtual contexts are stored. Additional challenges arise when filler-threads access memory; we defer discussion of these to Section III-B3.

B. Master-cores

As discussed in Section II-B, co-running additional threads alongside a latency-sensitive thread can drastically harm tail latency [39, 45, 48]. As such, many prior works reject SMT for latency-critical applications (e.g., [39, 48]) and most server-optimized scale-out processors, such as Cavium ThunderX [85] and Qualcomm Centriq [86], do not employ multithreaded cores. However, in the microsecond regime, where threads frequently face

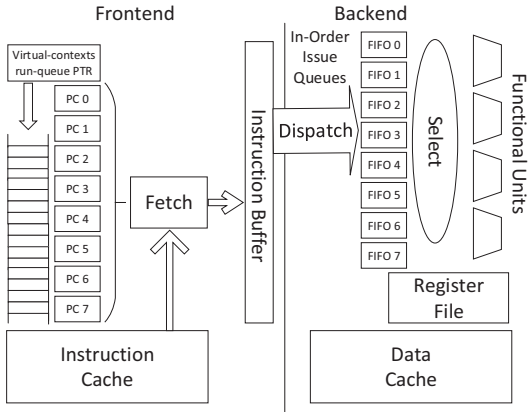

Figure 3. Lender-core: 8-way InO Hierarchical SMT (HSMT).

long stalls, SMT provides the most promising approach to recover the lost cycles.

This dichotomy motivates our design for master-cores, where we execute the *master-thread* by itself, to preserve its tail latency, but multiplex several *filler-threads* during master-thread stalls, to recover throughput. In master-thread mode, the master-core operates as a single-threaded 4-wide OoO superscalar core, optimizing for single-thread performance and minimal tail latency. In filler-thread mode, while the master-thread is stalled/idle, the master-core switches from its single-threaded OoO issue mechanism to the InO HSMT mechanism of a lender-core. It then multiplexes multiple filler-threads to use the available issue bandwidth. Together, these modes maximize performance density and energy efficiency by maximizing executed instructions.

1) From MorphCore to Master-core

Our master-core microarchitecture builds upon *MorphCore* [49]. MorphCore rests on two insights: (1) a 6-8 way in-order SMT core can achieve better total throughput than a single-threaded OoO core and (2) such an in-order core requires a subset of the hardware mechanisms already present in an OoO core. MorphCore reuses most hardware structures (instruction buffer, ALUs, RFs, load/store unit, etc.) in both execution modes. In multi-threaded mode, it partitions instruction buffers, reservation station (into multiple in-order issue queues), and the reorder buffer among threads, which all share functional unit pipelines. It repurposes the core's physical register file as architectural registers for each thread. Finally, it disables register renaming, dynamic scheduling, and the load queue to save energy. On a mode switch, MorphCore swaps the extra threads' architectural registers from a dedicated memory region using microcode.

We describe our master-core design by starting with MorphCore as an initial strawman and successively addressing challenges that arise in the killer microseconds context. First, we replace the conventional in-order SMT operating mode of MorphCore with the HSMT architecture of the lender-cores, described in Section III-A. Thus, when running filler-threads, the master-core will have sufficient available virtual contexts to hide killer microsecond stalls.

Second, we alter several aspects of how MorphCore transitions modes. The master-core triggers a transition whenever the master-thread becomes idle or incurs a μs-scale stall. We drain instructions elder than the stalling instruction and flush younger instructions.

In contrast to MorphCore, a master-core does not evict the architectural register state of the master-thread; it retains its registers to facilitate fast restart when the stall resolves, after which all in-flight instructions from filler-threads are immediately squashed.

There are two challenges with this strawman master-core design, if used alone. First, filler-threads thrash the cache, TLB, and branch predictor state of the master-thread. When the master-thread resumes, it will incur many cache misses, which may adversely affect its tail latency. Second, filler-threads have no guarantee when they will be scheduled on the master-core; they are only scheduled when the master stalls, and hence they may starve. We next solve these problems.

2) Segregating State

We must ensure that filler-threads do not thrash the master-thread's state. The naive approach is to replicate all stateful micro-architectural structures (register files, caches, branch predictor, TLBs, etc.), segregating the filler-threads' from the master-thread's state. We compare against this alternative, shown in Figure 4(a), in our evaluation. The problem with replicating all structures is that caches and register file are large and power-hungry. In particular, depending on microarchitecture, register files usually consume 5%-20% and L1 caches consume 10%-40% of a core's area [87]. So, this approach undermines Duplexity's performance density and energy efficiency objectives.

Instead, Duplexity replicates only the area-inexpensive structures. We provision a full-size TLB and reduced-size branch predictor for exclusive use by filler-threads. For the register file, we provision empty physical registers to store the architectural state of filler-threads, using the renaming logic to track the assignment of logical filler-thread registers to physical registers. Once its in-flight instructions are squashed or drained, the master-thread occupies only enough physical registers to maintain its architectural state.

To avoid replicating caches, we introduce the concept of dyads, which we discuss next.

3) Master-Lender Dyads

Instead of replicating caches, we pair a master-core with a lender-core to form a *dyad*. When a master-core morphs into filler-thread mode, the filler-threads remotely access the L1 instruction and data caches of the lender-core. The dyad provides data paths from the master-core's fetch and memory units to the lender-core's caches, as shown in Figure 4(b). This approach has two benefits: (1) it protects the master-thread's state, and (2) it allows filler-threads to hit on their own cache state as they migrate between the cores. However, this approach also entails two challenges: (1) The L1 access latency of filler-threads on the master-core is $\sim$3 cycles higher than local cache access in either core. (2) The capacity pressure and bandwidth requirements on the lender-core's caches increase, since both cores may access them.

We address these challenges by provisioning a small 2KB L0 I-cache and a 4KB L0 write-through D-cache in the master-core for accesses to the lender-core's L1 caches. Although these L0 caches have low hit rates, they act as effective bandwidth filters and service many sequential accesses, especially for instructions. Whereas capacity pressure on the lender-core's L1 cache is high, HSMT is inherently latency-tolerant; our evaluation demonstrates a net throughput win. The lender-core L1 D-cache maintains inclusion with L0 D-cache and forwards invalidations to maintain coherence.

4) Fast Filler-thread Eviction

A key Duplexity objective is to ensure fast master-thread resumption when it becomes ready. We use several approaches to accelerate restart.

First, we reuse the L0 data cache to accelerate spilling filler-thread architectural state. The L0 cache is write-through, hence, its contents can be discarded or overwritten at any time. When the master-thread becomes ready, all pending filler-thread instructions are immediately flushed. Then, all physical register file read ports are used to read filler-thread architectural state and write it to the L0 data cache. With 8 read ports and an L0 write bandwidth of one cache-line per cycle, it takes less than 50 cycles to spill the filler-threads. We assume each thread requires 16 64-bit GP integer registers and 16 128-bit XMM floating-point/SIMD registers, per the x86-64 ISA. The master-core's physical register files include 144 registers—sufficient for architectural registers of 9 threads (the master and 8 filler-threads). The 4KB L0 capacity is sufficient to absorb the spill of all filler-thread registers.

During the spill, the master-core can begin dispatching master-thread instructions but instructions do not issue until read ports become available. As the master-thread's cache state is intact, fetches are likely to hit. Furthermore, the master-thread's architectural state is already present in the physical register file, as we do not evict it. Filler-thread register state is drained from the L0 to the dedicated backing store in memory in the background. In short, master-thread resumption incurs roughly a 50-cycle delay.

C. Summary

Figure 4(c) depicts the final Duplexity design, comprising several dyads each with a master- and a lender-core that share virtual contexts. The lender-core uses HSMT with 8 physical contexts sharing an 8-way InO datapath. HSMT enables the lender-core to hide μs-scale stalls in its latency-insensitive virtual context pool. The master-core can fill the master-thread's μs-scale holes with filler-threads borrowed from the lender-core by morphing into an InO HSMT architecture, while still protecting the master-thread from tail latency disruption. Sharing virtual contexts across the dyad prevents contexts from starving.

IV. Discussion

Scheduling. Duplexity affects several aspects of how the OS must manage SMT threads. The OS must schedule latency-critical threads on master-cores and provision the virtual contexts for each dyad. Since the number of virtual contexts is variable, and should be tuned based on the frequency and duration of stalls, a dyad appears to software as if it supports a variable number of hardware threads. The scheduling of virtual contexts on the physical contexts of master- and lender-cores is transparent to software. Conceptually, the master-core is exposed to software as a single-threaded core, while virtual contexts belonging to a dyad belong to the lender-core. Existing CPU hot-plug mechanisms [88] may be applicable to vary the number of virtual contexts at runtime. Alternatively, OS designs like Barrelfish [89], which separates core, thread, and OS abstractions, might be adopted.

The OS must select how many virtual (filler) contexts to activate in a dyad. One option is to simply over-provision, but this may lead to long scheduling delays for ready virtual contexts. Alternatively, a data-center-scale scheduling layer might optimize thread

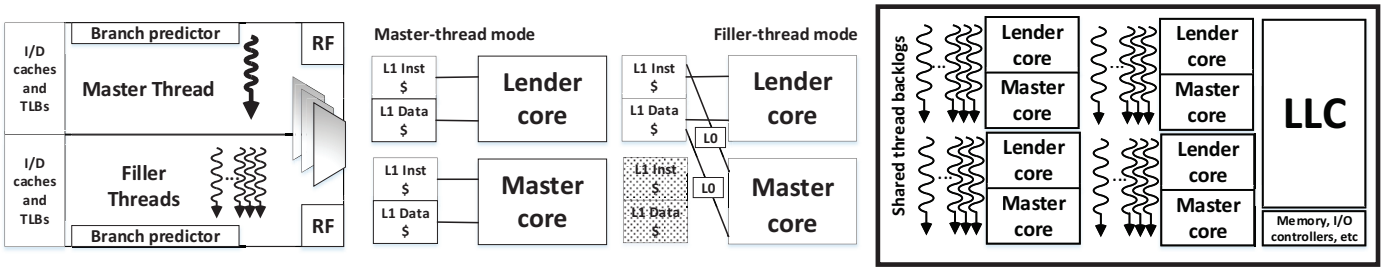

Figure 4. (a) A naive master-core design where stateful micro-architectural components are replicated across modes, (b) A Duplexity dyad composed of a master-core and a lender-core, and (c) Layout of a Duplexity server processor chip.

assignments via data-center-wide optimization [40, 90]. We find empirically that 32 virtual contexts per dyad are sufficient to hide stalls in our most pessimistic scenarios, wherein both the latency sensitive and batch threads incur frequent stalls (1 μs stall per μs of compute). If batch threads do not incur μs-scale stalls, 16 batch threads are sufficient; eight each to fill contexts on the lender and master-cores. If only batch threads incur μs-scale stalls (and thus never run on the master-core), 21 threads are sufficient to occupy the lender-core (see Figure 2(b)).

We use a simple round-robin scheduling policy for virtual contexts, which is easy to implement in hardware and provides some fairness/starvation avoidance. Virtual contexts are scheduled on a physical context for a 100 μs quantum to prevent starvation. Because this quantum is far lower than the OS scheduling quantum, the two scheduling mechanisms do not interfere—from the OS perspective, all virtual contexts are active, much like hardware threads in an SMT system. Unused virtual contexts are parked via HLT, much like unused hyperthreads. Note that the two-level scheduling applies only to latency-insensitive batch threads.

Throughput threads. Duplexity's approach increases the number of active threads per chip. The need for more threads to maintain utilization is an inherent consequence of more frequent and longer stalls and is a key aspect of scale-out server architectures. For multi-programmed workloads, a possible consequence is an increase in server memory capacity requirements. Duplexity's improved latency tolerance dovetails with emerging memory technologies like 3D XPoint [2, 50] which trade improved capacity for longer access latency. For many classes of scale-out batch workloads (e.g., graph analytics [91], task-parallel applications [92, 93], Spark [94], and Hadoop [95]), it is often possible to partition data shards or tasks among threads at finer granularity to exploit more parallelism within the same memory footprint and provide flexibility in the number of threads. Moreover, individual tasks are often latency insensitive, making them well-suited to Duplexity. These workloads typically benefit substantially from hardware multithreading as they can overlap multiple remote accesses and provide (remote) memory-level parallelism (MLP) through thread-level parallelism (similar to the execution model of GPUs). In the absence of sufficient hardware threads, such distributed big-data algorithms must rely on complex asynchronous programming models and continuation/call-back mechanisms to provide MLP [33].

Duplexity protects the master-thread from interference by filler-threads. Nevertheless, batch/filler-threads may interfere with one another. Existing work on intelligent co-location may be applicable to mitigate such interference [40, 42, 43].

Demarcating stalls. A second aspect of Duplexity is that we assume that hardware can recognize the start and end of μs-scale stalls. For example, remote disaggregated memory accesses can be recognized from their memory translations or use queue pair-based memory models [12] that bypass the kernel, as in other forms of polling-based high-performance I/O protocols that are transparent to the OS [60–65]. Stalls end when remote loads return. Alternatively, special monitoring instructions (e.g., `mwait`, variants of `hlt` [96]), can wake upon cache coherence activity or data/work arrival [97].

Alternative approaches. GPUs and user-level multithreading present two alternative strategies to hide μs-scale stalls by multiplexing many threads. GPUs employ large register files to accommodate all active threads and accelerate context switching [98–100]. However, GPUs are applicable only to workloads amenable to their distinct programming model (CUDA/OpenCL), which is typically ill-suited for most software frameworks that run in the cloud—especially I/O intensive workloads or those where concurrency arises from request rather than data parallelism. User-level multithreading (e.g., [54, 101, 102]) enables fast context-switching through cooperative threading. This approach also entail substantial software re-engineering and does not apply to existing binaries. Both of these approaches are better suited for throughput rather than latency-sensitive applications. Moreover, neither approach protects a latency-critical thread from throughput-thread interference.

V. Evaluation Methodology

We use gem5 x86-64 [73] to evaluate Duplexity. We extend gem5 to model the master-core (and our other OoO baselines) and evaluate its performance in detailed simulation. We model a single dyad. For the scale-out workloads running on filler-threads, we determine the throughput of multi-threaded workloads on the in-order master-/lender-cores through trace-based simulation. We analyze energy and area with McPAT [87], and apply the changes described in [103] to more accurately model OoO cores. We estimate tail latencies using the BigHouse [67] methodology. We simulate the queuing system until we achieve 95% confidence intervals of 5% error in reported results. We measure IPC in gem5 and use it to determine the service rate of an FCFS M/G/1 queuing system. We then simulate the high-level behavior of the queue at request (rather than instruction) granularity. The M/G/1 assumption is in line with prior studies [44, 104, 105]. We generate service times in BigHouse by measuring their distribution on real hardware, and scaling them using IPC slowdowns measured in gem5.

Overheads. The master-core builds upon a 4-wide OoO microarchitecture. We add the ability to transition to filler-thread mode, much like MorphCore. As such, the master-core entails all the hardware overheads of MorphCore (extra muxing paths in the front-end, select, and wakeup logic, and additional bypass paths in the back-end). Khubaib reports an area overhead of $\sim$2% for these structures [49]. In addition, the master-core provisions a TLB, reduced-size branch predictor, L0 I/D caches for use by filler-threads, and fetch/memory-access data-paths to the lender-core's caches. We model these additional structures with McPAT and find that the additional TLBs, branch predictors, and L0 caches impose area overheads of 0.7%, 1.2%, and 1%, respectively. The total area overhead of the master-core is approximately 5% compared to a baseline 4-wide OoO core. The static power overhead is within 5% of the baseline. In contrast, a master-core variant that replicates all stateful structures, including L1 caches, incurs a 38% area overhead. Our master-core requires additional multiplexers at various pipeline stages to mux between InO/OoO data paths used in different modes. Assuming 20 gates per pipeline stage [106], we estimate a cycle time penalty of 4% for these muxes. We include area, frequency, and power overheads in our results.

Workloads. We consider the following microservices, two of which are simplified/simulator-friendly versions of microservices from [32]; the other two are constructed using the same framework.

- **FLANN:** We evaluate two configurations of the FLANN [72] microservice introduced in Section II-B; *FLANN-HA* (High-Accuracy) has an LSH lookup latency of 10μs and identifies a large number of nearest-neighbor candidates. *FLANN-LL* (Low-Latency) reduces lookup latency to only 1μs by using longer hash keys. Both of these configurations issue a one-sided single–cache-line remote access to retrieve one of the identified nearest neighbors. We assume single–cache-line RDMA read latency to be exponentially distributed with a 1μs average [15].
- **Remote Storage Caching (RSC):** We implement a remote storage caching microservice; a simplified variant of Flash caches [107–110]. Our RSC microservice maps linear block addresses of a remote storage system to a local low-latency SSD using Cuckoo hashing [111]. We only consider read transactions; allocation and coherence mechanisms fall outside the scope of our experiments. Look-up latency is 3μs, which, upon a hit, is followed by 8μs average access latency to Intel's Optane SSD [51] through user-level polling [52] and 4μs average latency for a 4KB *memcpy*. Though optimistic for current-generation Optane, we believe these characteristics are representative of future devices.
- **McRouter:** We employ a consistent hashing microservice based on Facebook's McRouter [27, 28]. This microservice routes Key-Value (KV) operations to 100 leaf servers via a consistent hash function and synchronously waits for leaf responses. We consider a state-of-the-art RDMA-based low-latency KV store that uses single-sided operations to minimize communication latency [29, 112]. The root microservice requires 3μs to route each request and the leaf KV store requires 3-5μs depending on the KV operation [29].
- **Word Stemming:** Stemming is a normalization process used to reduce words to their root and is a core query rewriting microservice employed in various cloud applications, such as web search. We develop a word stemming microservice based on Oleander's implementation of the Porter stemming algorithm [113, 114]. This microservice incurs no μs-scale stalls, since it is a leaf service. Hence, core under-utilization arises only due to the idle time between requests. Furthermore, it is state-less; it hard-codes all stemming paths (prefixes, suffixes, etc.) into the program control-flow. It requires an average processing time of 4μs.

Filler-threads execute distributed PageRank and Single-Source Shortest Path algorithms based on bulk synchronous processing [115] and synchronous queue pair-based disaggregated memory model [12] on a single dataset representing a subset of the Twitter graph [116]. Reading a remote vertex requires a single–cache-line RDMA read that takes 1μs [15]. Since almost half of vertices are accessed remotely through RDMA, our filler-threads also require 1μs stall time per each 1-2μs of compute. We execute 32 filler-threads per dyad.

Design Configurations. We compare a Duplexity dyad to a variety of alternative core microarchitectures. Our performance density and energy efficiency studies pair each core alternative with a throughput-oriented HSMT core (configured to match Duplexity's lender-core) for a fair throughput comparison. Our main objective is to contrast the impact of these architectures on the microservice's tail latency/QoS.

We consider the following alternatives:

(1) Baseline: A 4-wide OoO core that only executes the latency-sensitive microservice.

(2) SMT: Baseline augmented with a second SMT batch thread, using ICOUNT [117]. The core does not prioritize the latency-critical thread.

(3) SMT+: Similar to SMT but prioritizes the latency-sensitive microservice over its co-runner unless the microservice thread is stalled. For bandwidth (per-cycle) resources (Fetch, Issue, Commit), SMT+ always prioritizes the latency-sensitive thread and only allocates slots to the co-runner if the microservice thread does not need them [118]. For storage resources (IQ, ROB, LSQ), SMT+ limits the co-runner to occupy at most 30% of the slots [119].

(4) MorphCore: MorphCore as proposed in [49], running 8 filler-threads when it morphs.

(5) MorphCore+: MorphCore extended with our HSMT mechanism and paired with a lender-core (i.e., it borrows threads from a shared virtual context pool, like a master-core).

(6) Duplexity + replication: A Duplexity (master-core+lender-core) variant wherein all master-core's stateful structures, including caches, are replicated.

(7) Duplexity: Our final Duplexity (master-core+lender-core) design. (Master-core shares L1 I/D caches with its neighbor lender-core when running filler-threads; L0 caches as bandwidth filters/register-buffers).

We report microarchiture configuration details in Table I and area/frequency results, obtained using McPAT [87] and CACTI [120] for 32nm technology, in Table II.

VI. Efficiency Results

A. Core Utilization

Figure 5(a) reports average core utilization. We calculate core utilization by dividing the number of retired instructions per

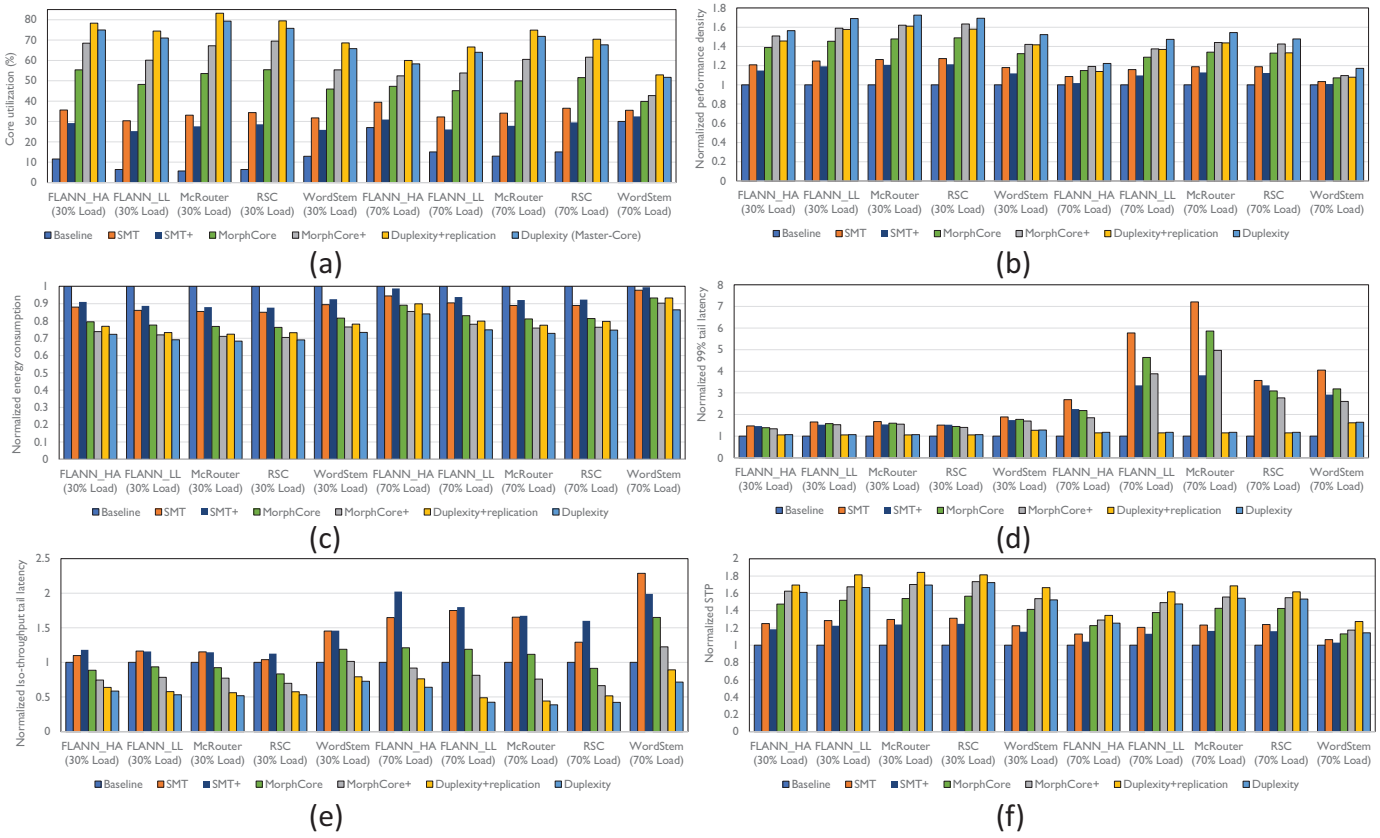

Figure 5. (a) Core utilization, (b) Normalized performance density, (c) Normalized energy consumption, (d) Normalized 99% tail latency, (e) Normalized iso-throughput 99% tail latency (f) Normalized system throughput (STP) for batch threads.

Table I
MICROARCHITECTURE DETAILS

Baseline/SMT	4-wide OoO, 144-entry ROB/PRF, 48-entry LQ, 32-entry SQ, ICOUNT fetch for SMT Tournament predictor: bimodal (16K), gshare (16K) and selector (16K); 32-entry RAS; 2K-entry BTB, 64-entry I/D TLBs
Lender-core	8-way InO HSMT, 32 virtual contexts, 4-wide issue, 128-entry ARF, Round-Robin fetch, gshare (8K) predictor, 2K-entry BTB, 64-entry I/D TLBs
Master-core	Transitions between singe-threaded OoO and InO HSMT, uarch same as baseline; tournament(16k)/gshare(8k), separate TLBs for the two modes, 2KB/4KB I/D write-through L0 caches
L1 caches	Private 64KB I/D, 64B lines, 2-way SA
LLC	1 MB per core, 64B lines, 8-way SA
Memory	50 ns access latency
NIC	FDR 4x Infiniband (56Gbit/s, 90M ops/s)

Table II
AREA AND CLOCK FREQUENCIES

Component	Area	Frequency
Baseline OoO	12.1 mm^2	3.4 GHz
SMT	12.2 mm^2	3.35 GHz
MorphCore	12.4 mm^2	3.3 GHz
Master-core	12.7 mm^2	3.25 GHz
Master-core + replication	16.7 mm^2	3.25 GHz
Lender-core	5.5 mm^2	3.4 GHz
LLC	3.9 mm^2/MB	N/A

cycle by the core's peak retire bandwidth (i.e, 4). These results include only the utilization of the master-core or its alternatives. Whereas instructions executed from borrowed threads are included, instructions executed on the lender-core are not. Baseline OoO core utilization is at most 29% and drops to 5.7% when load is low (30%) and the stall ratio is high (e.g., ∼60% in McRouter). The baseline OoO scheme is single-threaded and has no mechanism to mitigate μs-scale stalls, so this result is not surprising. All other architectures improve utilization by executing instructions from filler-threads. SMT and MorphCore yield considerably lower utilization than HSMT-based designs (MorphCore+ and Duplexity variants) as 8 filler-threads are insufficient to hide stalls.

The Duplexity variants achieve the highest utilization. However, as load or work/stall ratio increases, utilization falls since filler-threads execute instructions only when the master-thread is idle/stalled. When active, the master-thread runs by itself and utilization depends only on the master-thread's IPC; prior work [46, 47] reports that scale-out applications exhibit low ILP/MLP and fail to fully utilize a core. Conversely, at low load or work/stall ratio (e.g., McRouter and RSC at 30% load), Duplexity fills wasted cycles with useful work from filler-threads and increases issue bandwidth utilization to 79%. Finally, whereas Duplexity improves average utilization by $4.8\times$ and $1.9\times$ over the baseline and SMT, respectively, it always achieves lower utilization (3.6%, on average) than Duplexity + replication. Replication reduces lender-core cache pressure when the master-core runs filler-threads.

Nevertheless, replicating caches is area-inefficient and incurs a drastic performance density penalty (see Table II).

The low utilization of MorphCore+ compared to Duplexity arises because of (1) cold misses when the latency-critical thread resumes execution (due to cache pollution by filler-threads) and (2) mode switching latency. SMT+ achieves the lowest utilization (except for the baseline OoO) as it limits the co-runner thread to use only 30% of hardware resources to minimize master-thread interference; it achieves $2.4\times$ lower utilization, on average, compared to Duplexity. The WordStem microservice provides the least opportunity for utilization improvement as it does not incur μs-scale stalls; opportunity arises only during idle periods. However, even under 70% load of WordStem, Duplexity improves issue bandwidth utilization by 69% and 41% compared to baseline and SMT, respectively, because the IPC of 8 co-running filler-threads is substantially higher than OoO and SMT.

B. Performance Density & Energy Efficiency

Performance density (Figure 5(b)) and energy efficiency (Figure 5(c)) are widely used in the literature to normalize performance over cost and estimate TCO [36, 121]. In these results, we pair alternative core variants with a throughput-oriented InO HSMT core (configured to match Duplexity's lender-core) to compare throughput fairly.

Performance density—instructions retired per unit-time per unit-area—enables comparison of area efficiency, which is critical when comparing TCO across heterogeneous designs [36]. Figure 5(b) reports normalized performance density across design points. Duplexity achieves the highest result: 49% and 28%, on average (up to 72% and 37%), higher than baseline and SMT, respectively. These results generally track utilization results (Figure 5(a)) with two exceptions: First, the gaps between designs are smaller, since we normalize by the area of an entire chip, including the shared LLC. The throughput core (i.e., lender-core) and LLC area mask differences in core efficiency. Second, while Duplexity + replication yields the highest core utilization, its performance density is, on average, 9.2% (up to 13.4%) lower than Duplexity due to the large area overhead of replication.

Although it achieves slightly higher utilization, Duplexity + replication remains an undesirable design point. When the master-core does not borrow threads, all designs except Duplexity + replication achieve roughly the same performance density as the baseline. However, Duplexity + replication loses $\sim$17% density relative to the baseline—due to its considerably higher area—which translates to higher TCO. The replication cost is even higher if we apply Duplexity to scale-out processors [36] (e.g., Cavium ThunderX [85]), which trade SRAM for cores and share a modestly sized LLC (e.g., 8-16MB) across many cores (e.g., 32-64 cores).

To measure energy, we divide the power consumption of each design by the average number of instructions retired each cycle. Figure 5(c) reports normalized energy results, which largely mirror the trend of performance density. Duplexity nearly always consumes the least energy, as Duplexity is able to retire the highest number of instructions per cycle among all designs except Duplexity + replication, which falls short on energy-efficiency because it replicates power-hungry structures. In particular, Duplexity is able to reduce energy consumption by 34% and 21%, on average, compared to baseline and SMT architectures, respectively.

Duplexity replicates some units (e.g., TLBs, predictors), which are not used when the master-thread executes. At first blush, this may appear antithetical to our goal of maximizing utilization. We decide whether to replicate or borrow a structure based on area and power—we only replicate inexpensive structures, for which replication provides a performance density and energy efficiency win. The metrics measured in Figures 5(b) and 5(c) capture this trade-off; if replication is area- or power-inefficient, these metrics worsen. In other words, these metrics represent the overall system utilization normalized against the area/power cost of all units.

VII. Performance & QoS Results

We report QoS via aggregate throughput for batch threads and tail-latency for the latency-critical microservices.

We determine Duplexity's impact on tail latency as described in Section V. Even small service time increases are amplified in the tail by queuing effects, especially at high loads [122]. Figure 5(d) reports the normalized 99th percentile tail latency under various load levels. Whereas SMT, Morphcore, and Morphcore+ increase tail latency by up to $7.2\times$, $5.8\times$, and $4.9\times$, respectively, Duplexity only increases tail latency by 19%, while recovering $4.8\times$ higher core instruction throughput.

We make two further observations from Figure 5(d): First, SMT+ usually achieves considerably (up to 89%) lower tail latency than SMT as it minimizes the master-core interference via prioritization and partitioning. In fact, when the mode switch frequency is high and stalls are short, SMT+ achieves lower tail latency than MorphCore/MorphCore+, as it is less disruptive to master-thread issue bandwidth. However, its tail latencies are still higher than the baseline (up to $3.8\times$), due to interference on caches and core resources. Second, while WordStem does not incur μs-scale stalls and does not maintain any state across requests, it nevertheless is sensitive to instruction cache interference and suffers high tail latencies under SMT and MorphCore variants.

Figure 5(e) reports normalized iso-throughput 99% tail latency of the master-thread across workloads and load levels, to make a system-level assessment. These results normalize across designs such that they achieve the same cost by varying input load in proportion to the performance density reported in Figure 5(b). The intent of this metric is to compare the impact of two microarchitectures on tail latency at a particular throughput while accounting for the fact that the designs differ in area, and therefore cost. Improving this metric implies a more tail-tolerant microarchitecture at a given cost. Duplexity achieves the lowest iso-throughput tail-latency, which is up to $2.6\times$ and $4.3\times$ ($1.8\times$ and $2.7\times$, on average) lower than iso-throughput tail latencies achieved by baseline and SMT, respectively. Whereas MorphCore variants achieve lower iso-throughput tail latencies over the baseline due to their high efficiencies, SMT variants lengthen iso-throughput tail latency compared to the baseline, as they do not sufficiently improve utilization. Perhaps surprisingly, while SMT+ provides isolation and prioritization mechanisms to protect the master-thread's performance, in some cases it yields worse iso-throughput tail latencies than SMT (up to 23%), due to lower utilization.

For the batch threads, we report system throughput (STP) [123], a metric that considers both performance and fairness for multi-threaded workloads. Figure 5(f) reports normalized batch-thread STP. Again, we pair all cores with an HSMT core for a fair

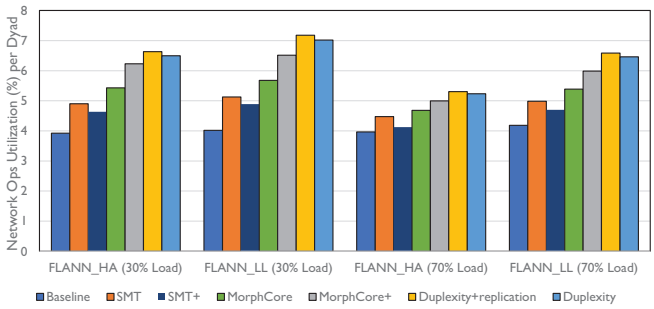

Figure 6. Network BW (IOPS) utilization (%) per dyad.

comparison. MorphCore+ and Duplexity + replication yield better STP than Duplexity, because Duplexity shares the lender-core's caches between the master-core and the lender-core when the master-thread is idle/stalled, degrading performance. Nevertheless, Duplexity still improves batch STP over the baseline and SMT by an average of 52% and 24%, respectively, within 8% of the best STP achieved by Duplexity + replication.

VIII. Case Study: Interconnect Utilization Analysis

Duplexity improves CPU utilization via thread-level parallelism; thus, it requires enough busy threads to be effective. As such, it is critical that bottleneck resources provide sufficient bandwidth to support all threads, otherwise, we simply shift from one bottleneck to another. For disaggregated memory, interconnect bandwidth is a key resource. So, we confirm that Duplexity's interconnect requirements are feasible.

We consider a single FDR 4x Infiniband link to calculate network bandwidth utilization, following [15]. Most NICs impose two bandwidth constraints: a maximum data rate, and a maximum I/O operations per second (IOPS), respectively 56Gbit/s and 90M ops/s for FDR [124, 125]. As our workloads issue single–cache-line remote accesses, they are IOPS-limited. Figure 6 reports network IOPS utilization per dyad, which largely tracks core utilization. Duplexity improves average network utilization over the baseline and SMT by 58% and 29%, respectively. The key takeaway is that, although the main purpose of Duplexity is to improve compute utilization, it also improves utilization for other resources.

Further, Figure 6 confirms that Duplexity incurs requirements that fall within current networking capabilities: the maximum IOPS of each dyad is less than 7.1% of the FDR capability. Hence, 14 dyads can share one NIC port. Further scalability is possible with multiple NICs or through continued scaling of Infiniband technology to higher rates [15, 126].

IX. Related Work

To the best of our knowledge, Duplexity is the first work to provide architectural support to fill the utilization holes caused by killer-microsecond, without sacrificying QoS and tail latency. Concurrent to this work, [54] proposes a software multithreading and prefetching-based solution to hide μs-scale memory accesses and μDPM [127] seeks to improve server energy-efficiency in the killer microseconds era. While Duplexity is a server design that targets general μs-scale I/O accesses and idle periods, the approach used in [54] is only amenable to cachable memory accesses and provides no QoS guarantees.

We review related studies on the key aspects of Duplexity.

Co-location and isolation. There is a large body of work that aims to improve processor utilization in data center workloads by co-locating batch and latency-sensitive applications using mechanisms to minimize interference among co-running applications [39–43]. Bubble-Flux [42] and Bubble-up [41] are online schemes that detect interference and identify "safe" co-locations to bound performance degradation while maximizing core utilization. Heracles [39] focuses on latency-critical workloads and employs isolation techniques to minimize interference between latency-critical and batch workloads. Dirigent [43] seeks to improve utilization by minimizing variation in latency-critical workloads. Finally, Paragon [90] and Quasar [40] use online classification techniques to co-locate workloads that are unlikely to interfere. These studies do not co-locate applications on different hardware threads of the same core, likely because SMT co-location (without hardware support to mitigate interference) results in drastic tail latency increases [39]. Moreover, these studies consider application behavior at millisecond (and higher) time-scales; they do not seek to address μs-scale stalls.

Further work addresses thread interference in SMT cores using specialized performance accounting hardware [128], shared resource usage tracking [129, 130], performance sampling [131], and competition heuristics [132]. However, most of the classic works do not provide QoS guarantees with respect to tail latency. Lo et al. [39] show that SMT co-location of batch and latency-critical threads can have catastrophic impacts on the tail latency and, in particular, Google's latency-critical workloads are not able to meet their QoS targets if co-located with batch threads without isolation mechanisms. SMiTe [44] and Elfen scheduling [45] aim to minimize interference while providing QoS guarantees. SMiTe [44] follows Bubble-Up to identify co-locations that minimize QoS violations by determining workload contentiousness and interference-sensitivity. However, their results show that even the best SMT co-locations may violate QoS for $\sim$20% of requests. Elfen scheduling [45] prioritizes the latency-sensitive thread over the batch thread; the batch thread polls regularly to see if the latency-sensitive thread is running and then voluntarily deschedules itself. However, polling at μs time scales implies untenable overhead. Furthermore, both SMiTe and Elfen scheduling only consider a single batch thread, which we show is insufficient if the batch threads also incur μs-scale stalls. With many batch threads, all storage-based components of the core (especially L1 caches) become very important and it would be essential for the core to incorporate some isolation mechanism with respect to such resources, which neither SMiTe nor Elfen scheduling provides.

There have been many proposals for isolation and partitioning schemes with respect to shared last level caches [133–142] and memory bandwidth [140, 143–147]. These proposals are orthogonal and can compose with Duplexity.

Reconfigurable/heterogeneous architectures. Other work aims to exploit thread-level parallelism (TLP) through microarchitectural reconfiguration. Such designs provision numerous simple compute units/cores that can either serve multiple threads individually or be ganged together to build more powerful cores when TLP is low or peak single-threaded performance is needed [148–150]. MorphCore [49] takes the opposite approach and morphs a wide OoO core into a multithreaded SMT core when threads are abundant. In

conjoined/composite cores [151, 152], two cores share components to increase efficiency, as in Duplexity. Duplexity differs from these architectures as it seeks to fill killer microsecond stalls and prevent QoS harm for the latency-critical microservices.

Heterogeneous multicores, which incorporate heterogeneous cores representing different points in the power/performance design space, have been proposed to improve energy-efficiency [153, 154]. Recent proposals allocate threads/tasks at either service-level [155] or request-level [68] to particular core types to improve throughput, density, and energy-efficiency, while meeting QoS targets. Duplexity fits within this class of multicore architecture, introducing the novel notions of dyads and thread borrowing.

X. Conclusion

Duplexity is a server architecture that aims to maximize performance density and energy efficiency by filling the *killer microsecond* utilization "holes" of microservices. These holes result from stalls due to accessing fast I/O devices or brief idle periods between requests. Neither existing microarchitectual techniques nor OS context switches can hide μs-scale stalls. Duplexity couples a latency-oriented master-core and throughput-oriented lender-core into a dyad. The master-core primarily executes a latency-critical master-thread. However, when idle or stalled, the master-core morphs into a multithreaded throughput mode and borrows filler-threads from the lender-core to fill utilization holes. By provisioning separate memory paths for the master and filler-threads, Duplexity protects master-thread cache state facilitating fast restart when a stall resolves. Our evaluation shows that Duplexity improves core utilization and iso-throughput tail-latency by $1.9\times$ and $2.7\times$ over an SMT-based server design.

XI. Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We especially thank Geoffrey Blake (ARM) and David Nellans (NVIDIA) for their insightful suggestions that helped improve this work. We thank Hossein Golestani for proof-reading the manuscript. This work was generously supported by ARM.

References

[1] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[2] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 631–644, ACM, 2017.

[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 133–146, ACM, 2009.

[4] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *European Conference on Computer Systems*, ACM, 2014.

[5] A. Mirhosseini, A. Agrawal, and J. Torrellas, "Survive: Pointer-based in-dram incremental checkpointing for low-cost data persistence and rollback-recovery," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 153–157, 2017.

[6] A. Tavakkol, A. Kolli, S. Novakovic, K. Razavi, J. Gomez-Luna, H. Hassan, C. Barthels, Y. Wang, M. Sadrosadati, S. Ghose, *et al.*, "Enabling efficient rdma-based synchronous mirroring of persistent memory transactions," *arXiv preprint arXiv:1810.09360*, 2018.

[7] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM SIGARCH Computer Architecture News*, 2014.

[8] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *ACM/IEEE International Symposium on Computer Architecture*, 2017.

[9] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[10] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *ACM SIGARCH Computer Architecture News*, ACM, 2009.

[11] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *USENIX Conference on Networked Systems Design and Implementation*, 2014.

[12] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out numa," in *ACM SIGPLAN Notices*, vol. 49, pp. 3–18, ACM, 2014.

[13] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap.," in *NSDI*, 2017.

[14] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote memory in the age of fast networks," in *Symposium on Cloud Computing*, ACM, 2017.

[15] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," *Proceedings of the VLDB Endowment*, vol. 9, no. 7, pp. 528–539, 2016.

[16] D. Lustig and M. Martonosi, "Reducing gpu offload latency via fine-grained cpu-gpu synchronization," in *IEEE International Symposium on High Performance Computer Architecture*, 2013.

[17] A. Caulfield, E. Chung, A. Putnam, *et al.*, "A cloud-scale acceleration architecture," in *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[18] A. Mirhosseini, M. Sadrosadati, B. Soltani, H. Sarbazi-Azad, and T. F. Wenisch, "Binochs: Bimodal network-on-chip for cpu-gpu heterogeneous systems," in *IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2017.

[19] Y. Gan and C. Delimitrou, "The Architectural Implications of Cloud Microservices," in *Computer Architecture Letters (CAL), vol.17, iss. 2*, Jul-Dec 2018.

[20] Staci D. Kramer, "The biggest thing amazon got right: The platform." [Online; accessed 27-Apr-2018].

[21] Tony Mauro, "Adopting microservices at netflix: Lessons for architectural design." [Online; accessed 27-Apr-2018].

[22] Yoni Goldberg, "Scaling gilt: from monolithic ruby application to distributed scala micro-services architecture." [Online; accessed 27-Apr-2018].

[23] Steven Ihde and Karan Parikh, "From a monolith to microservices + rest: the evolution of linkedin's service architecture." [Online; accessed 27-Apr-2018].

[24] Phil Calcado, "Building products at soundcloud part i: Dealing with the monolith." [Online; accessed 27-Apr-2018].

[25] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, 2004.

[26] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing.," in *NSDI*, 2013.

[27] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani, "Introducing mcrouter: A memcached protocol router for scaling memcached deployments," 2014.

[28] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, "Scaling memcache at facebook.," in *nsdi*, vol. 13, pp. 385–398, 2013.

[29] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," *ACM SIGCOMM Computer Communication Review*, 2015.

[30] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store.," in *USENIX Annual Technical Conference*, pp. 103–114, 2013.

[31] M. Barhamgi, D. Benslimane, and B. Medjahed, "A query rewriting approach for web service composition," *IEEE Transactions on Services Computing*, 2010.

[32] A. Sriraman and T. F. Wenisch, "μSuite: A Benchmark Suite for Microservices," in *International Symposium on Workload Characterization*, IEEE, 2018.

[33] A. Sriraman and T. F. Wenisch, "μTune: Auto-Tuned Threading for OLDI Microservices," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[34] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the workshop on Experimental computer science*, ACM, 2007.

[35] D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *Proceedings of the 2007 workshop on Experimental computer science*, p. 4, ACM, 2007.

[36] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, *et al.*, "Scale-out processors," in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society, 2012.

[37] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE micro*, 2010.

[38] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[39] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 450–462, ACM, 2015.

[40] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *ACM SIGPLAN Notices*, ACM, 2014.

[41] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-

locations," in *International Symposium on Microarchitecture*, ACM, 2011.
[42] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 607–618, ACM, 2013.
[43] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 33–47, 2016.
[44] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 406–418, IEEE Computer Society, 2014.
[45] X. Yang, S. M. Blackburn, and K. S. McKinley, "Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading.," in *USENIX Annual Technical Conference*, 2016.
[46] 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 *ACM SIGPLAN Notices*, ACM, 2012.
[47] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *ACM/IEEE International Symposium on Computer Architecture*, 2015.
[48] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martınez, "Workload characterization of interactive cloud services on big and small server platforms," in *IEEE International Symposium on Workload Characterization*, 2017.
[49] K. Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, Y. N. Patt, *et al.*, "Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp," in *IEEE/ACM International Symposium on Microarchitecture*, 2012.
[50] Intel, "3D Xpoint." https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.
[51] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3d xpoint technology," *Proceedings of the IEEE*, 2017.
[52] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "Mqsim: a framework for enabling realistic studies of modern multi-queue ssd devices," in *USENIX Conference on File and Storage Technologies*, 2018.
[53] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *International Symposium on High-Performance Computer Architecture*, 2003.
[54] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the killer microsecond," in *International Symposium on Microarchitecture 2018*.
[55] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *International Symposium on High-Performance Computer Architecture*, 2019.
[56] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *IEEE International Symposium on High Performance Computer Architecture*, 2018.
[57] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," *ACM SIGARCH Computer Architecture News*, 2005.
[58] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ACM SIGARCH Computer Architecture News*, 2006.
[59] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," *ACM SIGARCH Computer Architecture News*, 2009.
[60] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level tcp stack for multicore systems.," in *NSDI*, vol. 14, pp. 489–502, 2014.
[61] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, no. EPFL-CONF-201671, USENIX, 2014.
[62] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash local flash," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2017.
[63] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, no. EPFL-CONF-231395, 2017.
[64] P. Rogers and A. Fellow, "Heterogeneous system architecture overview," in *Hot Chips*, vol. 25, 2013.
[65] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds.," in *HotStorage*, 2016.
[66] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *International Symposium on Computer Architecture*, IEEE, 2011.
[67] D. Meisner, J. Wu, and T. F. Wenisch, "Bighouse: A simulation infrastructure for data center systems," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, IEEE, 2012.
[68] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting heterogeneity for tail latency and energy efficiency," in *IEEE/ACM International Symposium on Microarchitecture*, 2017.
[69] M. Harchol-Balter, *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.
[70] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," in *ACM Sigplan Notices*, ACM, 2009.
[71] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *ACM/IEEE International Symposium on Computer Architecture*, 2014.
[72] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, 2014.
[73] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.
[74] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 315–326, IEEE Computer Society, 2008.
[75] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Web search using mobile cores: quantifying and mitigating the price of efficiency," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 314–325, ACM, 2010.
[76] D. Loghin, B. M. Tudor, H. Zhang, B. C. Ooi, and Y. M. Teo, "A performance study of big data on small nodes," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 762–773, 2015.
[77] U. Hölzle, "Brawny cores still beat wimpy cores, most of the time," *IEEE Micro*, vol. 30, no. 4, pp. 23–24, 2010.
[78] C. Delimitrou and C. Kozyrakis, "Amdahl's Law for Tail Latency," in *Communications of the ACM (CACM)*, August 2018.
[79] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder, "Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy," in *Microarchitecture, 2004. 37th International Symposium on*, IEEE, 2004.
[80] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *International Symposium on Microarchitecture*, ACM, 2011.
[81] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, "Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2018.
[82] S. Hily and A. Seznec, "Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading," in *International Symposium On High-Performance Computer Architecture*, IEEE, 1999.
[83] F. M. Sleiman and T. F. Wenisch, "Efficiently scaling out-of-order cores for simultaneous multithreading," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 431–443, IEEE Press, 2016.
[84] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "Ibm power9 processor architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, 2017.
[85] Cavium, "ThunderX ARM Processors." https://cavium.com/product-thunderx-arm-processors.html.
[86] Qualcomm, "Qualcomm Centriq 2400." https://www.qualcomm.com/products/qualcomm-centriq-2400-processor.
[87] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *IEEE/ACM International Symposium on Microarchitecture*, 2009.
[88] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, and S. Vaddagiri, "Linux kernel hotplug cpu support," in *Linux Symposium*, vol. 2, 2004.
[89] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe, "Decoupling cores, kernels, and operating systems.," in *OSDI*, 2014.
[90] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, ACM, 2013.
[91] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD International Conference on Management of data*, ACM, 2010.
[92] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55 – 69, 1996.
[93] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.
[94] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, 2016.
[95] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.
[96] P. Guide, "Intel® 64 and ia-32 architectures software developers manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.
[97] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti, "LASER: Light, Accurate Sharing dEtection and Repair," in *IEEE International Symposium on High Performance Computer Architecture*, 2016.
[98] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *International Symposium on Microarchitecture*, ACM, 2015.

[99] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "Regmutex: Inter-warp gpu register time-sharing," in *ACM/IEEE 45th International Symposium on Computer Architecture (ISCA)*, 2018.

[100] H. A. Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "Corf: Coalescing operand register file for gpus," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2019.

[101] A. A. Donovan and B. W. Kernighan, *The Go programming language*. Addison-Wesley Professional, 2015.

[102] G. Boudol, "Fair cooperative multithreading," in *CONCUR 2007 – Concurrency Theory* (L. Caires and V. T. Vasconcelos, eds.), (Berlin, Heidelberg), pp. 272–286, Springer Berlin Heidelberg, 2007.

[103] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in mcpat and potential impacts on architectural studies," in *IEEE International Symposium on High Performance Computer Architecture*, 2015.

[104] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *International Symposium on Microarchitecture*, ACM, 2015.

[105] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *IEEE International Symposium on High Performance Computer Architecture*, 2015.

[106] Z. Chishti and T. Vijaykumar, "Optimal power/performance pipeline depth for smt in scaled technologies," *IEEE Transactions on Computers*, 2008.

[107] S. Byan, J. Lentini, A. Madan, and L. Pabon, "Mercury: Host-side flash caching for the data center," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pp. 1–12, IEEE, 2012.

[108] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *2015 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 51–60, IEEE, 2015.

[109] D. Arteaga and M. Zhao, "Client-side flash caching for cloud systems," in *Proceedings of International Conference on Systems and Storage*, ACM, 2014.

[110] D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer, "Flash caching on the storage client," in *USENIX Annual Technical Conference*, 2013.

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

[112] M. Kaminsky, A. K. Michael, and D. G. Andersen, "Design guidelines for high performance rdma systems," in *USENIX Annual Technical Conference*, 2016.

[113] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[114] M. F. Porter, "Snowball: A language for stemming algorithms," 2001.

[115] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[116] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?," in *International conference on World wide web*, ACM, 2010.

[117] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *ACM SIGARCH Computer Architecture News*, ACM, 1996.

[118] G. K. Dorai and D. Yeung, "Transparent threads: Resource sharing in smt processors for high single-thread performance," in *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pp. 30–41, IEEE, 2002.

[119] S. E. Raasch and S. K. Reinhardt, "The impact of resource partitioning on smt processors," in *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pp. 15–25, IEEE, 2003.

[120] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[121] Y. Zhou and D. Wentzlaff, "The sharing architecture: sub-core configurability for iaas clouds," *ACM SIGARCH Computer Architecture News*, 2014.

[122] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *IEEE International Symposium on Workload Characterization*, 2016.

[123] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE micro*, vol. 28, no. 3, 2008.

[124] Mellanox, "ConnectX-3 VPI ." http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_VPI_Card.pdf.

[125] O. Olusanya and M. Hussain, "Need for Speed: Comparing FDR and EDR InfiniBand." http://en.community.dell.com/techcenter/high-performance-computing/b/general_hpc/archive/2016/02/02/need-for-speed-comparing-fdr-and-edr-infiniband-part-1.

[126] A. Sriraman, S. Liu, S. Gunbay, S. Su, and T. F. Wenisch, "Deconstructing the Tail at Scale Effect Across Network Protocols," *The Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2016.

[127] C.-H. Chou, L. N. Bhuyan, and D. Wong, "μdpm: Dynamic power management for the microsecond era," in *IEEE International Symposium on High Performance Computer Architecture*, 2019.

[128] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for smt processor scheduling," *ACM Sigplan Notices*, vol. 45, no. 3, pp. 91–102, 2010.

[129] F. J. Cazorla, A. Ramirez, M. Valero, P. M. Knijnenburg, R. Sakellariou, and E. Fernández, "Qos for high-performance smt processors in embedded systems," *Ieee Micro*, 2004.

[130] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable performance in smt processors: Synergy between the os and smts," *IEEE Transactions on Computers*, 2006.

[131] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous mutlithreading processor," *ACM SIGPLAN Notices*, 2000.

[132] A. Vega, A. Buyuktosunoglu, and P. Bose, "Smt-centric power-aware thread placement in chip multiprocessors," in *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, IEEE, 2013.

[133] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, *et al.*, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *IEEE International Symposium on High Performance Computer Architecture*, 2018.

[134] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *IEEE/ACM International Symposium on Microarchitecture*, 2006.

[135] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *ACM SIGARCH Computer Architecture News*, ACM, 2011.

[136] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (prism)," in *ACM SIGARCH computer architecture news*, vol. 40, pp. 428–439, IEEE Computer Society, 2012.

[137] S. Srikantaiah, M. Kandemir, and Q. Wang, "Sharp control: controlled shared cache management in chip multiprocessors," in *International Symposium on Microarchitecture*, ACM, 2009.

[138] Y. Xie and G. H. Loh, "Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 174–183, ACM, 2009.

[139] H. Kasture and D. Sanchez, "Ubik: efficient cache sharing with strict qos for latency-critical workloads," in *ACM SIGPLAN Notices*, ACM, 2014.

[140] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "Qos policies and architecture for cache/memory in cmp platforms," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, pp. 25–36, ACM, 2007.

[141] F. Guo, H. Kannan, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis, "From chaos to qos: case studies in cmp resource management," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 21–30, 2007.

[142] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family," in *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[143] Y. Zhou and D. Wentzlaff, "Mitts: memory inter-arrival time traffic shaping," in *ACM SIGARCH Computer Architecture News*, IEEE, 2016.

[144] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *ACM Sigplan Notices*, vol. 45, pp. 335–346, ACM, 2010.

[145] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *IEEE/ACM International Symposium on Microarchitecture*, 2011.

[146] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *International Symposium on Microarchitecture*, 2006.

[147] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das, "Mete: meeting end-to-end qos in multicores through system-wide resource management," in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ACM, 2011.

[148] Y. Watanabe, J. D. Davis, and D. A. Wood, "Widget: Wisconsin decoupled grid execution tiles," in *ACM SIGARCH Computer Architecture News*, ACM, 2010.

[149] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *IEEE/ACM International Symposium on Microarchitecture*, 2007.

[150] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: accommodating software diversity in chip multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 186–197, ACM, 2007.

[151] R. Kumar, N. P. Jouppi, and D. M. Tullsen, "Conjoined-core chip multiprocessing," in *IEEE/ACM International Symposium on Microarchitecture*, 2004.

[152] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *IEEE/ACM International Symposium on Microarchitecture*, 2012.

[153] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 81–92, IEEE, 2003.

[154] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[155] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang, "Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers," in *IEEE International Symposium on High Performance Computer Architecture*, 2015.