Hiding the Microsecond-Scale Latency of Storage-Class Memories with Duplexity

Amirhossein Mirhosseini Akshitha Sriraman Thomas F. Wenisch
University of Michigan
{*miramir,akshitha,twenisch*}@*umich.edu*

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 [2, 3]. Storage-class memories, such as 3D Xpoint, are examples of such events that stall execution for single-digit microseconds. 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 [4]) are within the same latency orders as microsecond-scale stalls, so they are not a plausible latency-hiding technique for the microsecond regime.

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 time [5, 6]. 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 μs-scale stalls [7]. 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 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 [8], some SMT thread co-locations can have catastrophic impact on the tails of latency-critical threads, especially at high loads, due to contention for shared resources [9]. To avoid compromising the tail latency of critical threads due to SMT interference, we instead design *Duplexity* [10], a server architecture that seeks directly to address the killer-microsecond challenge—to fill in the μs-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 the first server architecture that aims to improve server utilization in the presence of μs-scale stalls, without sacrificing QoS and tail latency of micro-services. Our evaluation, using Gem5 [11] and BigHouse [12] simulation frameworks, 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.

II. Duplexity

We next present Duplexity—a server architecture that aims to fill in cycles lost to μs-scale stalls (e.g., caused by storage-class memories) while preserving tail latency and QoS. Duplexity comprises two kinds of cores: *master-cores*, optimized for latency-sensitive micro-services and *lender-cores*, optimized for latency-insensitive scale-out 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 [13], 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.

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

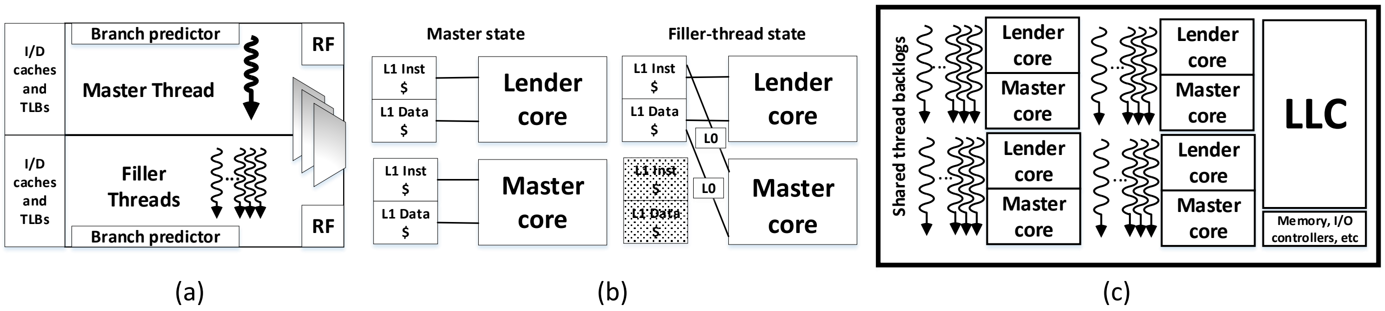

Figure 1. (a) A naive master-core design where stateful micro-architectural components are replicated across modes, (b) A Duplexity dyad composed of a master-core and a lender-core, and (c) Layout of a Duplexity server processor chip.

hierarchy of the lender-core. Hence, when the master thread returns, there is little evidence the filler threads were ever there.

We must ensure that filler threads do not thrash the master thread's state. The naive approach, shown in Figure 1(a), 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. 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. So, this approach undermines Duplexity's performance density and energy efficiency objectives.

Instead, Duplexity replicates only the area-inexpensive structures. We provision a TLB and a 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. 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 1(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. 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. 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.

Figure 1(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 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.

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] Intel, "3D Xpoint." https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[3] 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.

[4] A. Sriraman and T. F. Wenisch, "utune: Auto-tuned threading for oldi microservices," in *Operating Systems Design and Implementation (OSDI)*, 2018.

[5] 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.

[6] 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*, pp. 309–322, 2016.

[7] 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.

[8] 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.

[9] A. Mirhosseini and T. F. Wenisch, "The queuing-first approach for tail management of interactive services," *IEEE Micro*, 2019.

[10] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing server efficiency in the face of killer microseconds," in *High Performance Computer Architecture (HPCA), 2019 IEEE 25th International Symposium on*, IEEE, 2019.

[11] 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.

[12] 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.

[13] 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.