SIMR: Single Instruction Multiple Request Processing for Energy-Efficient Data Center Microservices

Mahmoud Khairy
School of ECE
Purdue University
abdallm@purdue.edu

Ahmad Alawneh
School of ECE
Purdue University
aalawneh@purdue.edu

Aaron Barnes
School of ECE
Purdue University
barnes88@purdue.edu

Timothy G. Rogers
School of ECE
Purdue University
timrogers@purdue.edu

***Abstract*—Contemporary data center servers process thousands of similar, independent requests per minute. In the interest of programmer productivity and ease of scaling, workloads in data centers have shifted from single monolithic processes toward a micro and nanoservice software architecture. As a result, single servers are now packed with many threads executing the same, relatively small task on different data.**

State-of-the-art data centers run these microservices on multi-core CPUs. However, the flexibility offered by traditional CPUs comes at an energy-efficiency cost. The Multiple Instruction Multiple Data execution model misses opportunities to aggregate the similarity in contemporary microservices. We observe that the Single Instruction Multiple Thread execution model, employed by GPUs, provides better thread scaling and has the potential to reduce frontend and memory system energy consumption. However, contemporary GPUs are ill-suited for the latency-sensitive microservice space.

To exploit the similarity in contemporary microservices, while maintaining acceptable latency, we propose the Request Processing Unit (RPU). The RPU combines elements of out-of-order CPUs with lockstep thread aggregation mechanisms found in GPUs to execute microservices in a Single Instruction Multiple Request (SIMR) fashion. To complement the RPU, we also propose a SIMR-aware software stack that uses novel mechanisms to batch requests based on their predicted control-flow, split batches based on predicted latency divergence and map per-request memory allocations to maximize coalescing opportunities. Our resulting RPU system processes $5.7\times$ more requests/joule than multi-core CPUs, while increasing single thread latency by only $1.44\times$.

***Keywords*-SIMT, Data Center, Microservices, GPU**

I. Introduction

The growth of hyperscale data centers has steadily increased in the last decade, and is expected to continue in the coming era of Artificial Intelligence and the Internet of Things [1]. However, the slowing of Moore's Law [2] has resulted in energy [3], environmental [4], [5] and supply chain [6] issues that has lead data centers to embrace custom hardware/software solutions [7], [8].

While improving Deep Learning (DL) inference has received significant attention [7], [9], general purpose compute units are still the main driver of a data center's total cost of ownership (TCO). CPUs consume 60% of the data center power budget [10], half of which comes from the pipeline's frontend (i.e. fetch, decode, branch prediction (BP), and Out-of-Order (OoO) structures) [11]–[15]. Therefore; 30% of the data-center's total energy is spent on CPU instruction supply.

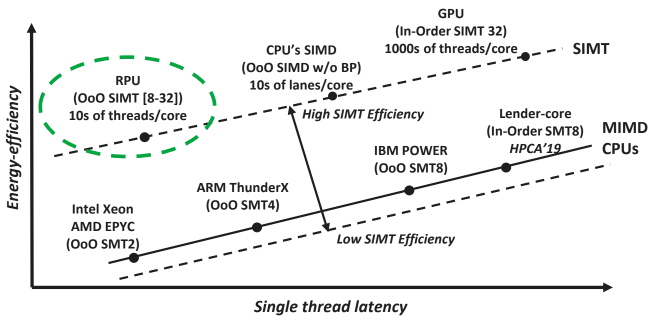


Figure 1: Conceptual energy-efficiency vs. single thread latency for different compute unit design points.

Coupled with the hardware efficiency crisis is an increased desire for programmer productivity, flexible scalability and nimble software updates that has lead to the rise of software microservices. Monolithic server software has been largely replaced with a collection of micro and nanoservices that interact via the network [16]–[18]. Compared to monolithic services, microservices spend much more time in network processing [17], [19], have a smaller instruction and data footprint [17], and can suffer from excessive context switching due to frequent network blocking [16], [20]–[22].

To meet both latency and throughput demands, contemporary data centers typically run microservices on multicore, OoO CPUs with and without Simultaneous Multithreading (SMT). Previous academic and industrial work [20], [23]–[28] has shown that current CPUs are inefficient in the data center as many on-chip resources are underutilized or ineffective. To make better use of these resources, on-chip throughput is increased [20], [25], [29] by adding more cores and raising the SMT degree [30]–[35]. Figure 1 visualizes the energy-efficiency and single thread latency of different processor design points, logically separated by their execution model. On the low-latency end are OoO Multiple Instruction Multiple Data (MIMD) CPUs with a low SMT-degree. Different CPU designs trade-off single thread latency for energy-efficiency by increasing the

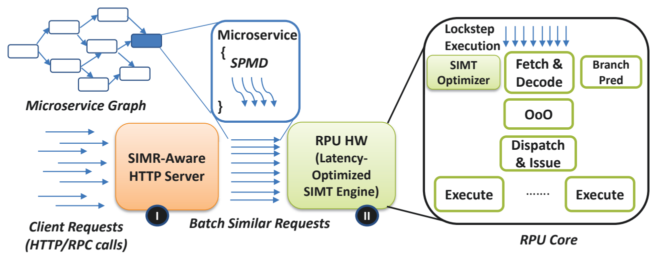


Figure 2: High level view of our SIMR system.

SMT-degree and moving from OoO to in-order execution. On the high-efficiency end are in-order Single Instruction Multiple Thread (SIMT) GPUs that support thousands of scalar threads per core. Fundamentally, GPU cores are designed to support workloads where single-threaded performance can be sacrificed for multi-threaded throughput. However, we argue that the energy-efficient nature of the GPU's execution model and scalable memory system can be leveraged by low-latency OoO cores, provided the workload performs efficiently under SIMT execution. SIMT machines aggregate scalar threads into vector-like instructions for execution (i.e. a warp). To achieve high energy-efficiency, the threads aggregated into each warp must traverse similar control-flow paths, otherwise lanes in the vector units must be masked off (decreasing SIMT-efficiency) and the benefits of aggregation disappear.

We make the observation that contemporary microservices exhibit a SIMT-friendly execution pattern. Data center nodes running the same microservice across multiple requests create a natural batching opportunity for SIMT hardware, if service latencies can be met. Contemporary GPUs are ill-suited for this task, as they forego single threaded optimizations (OoO, speculative execution, etc.) in favor of excessive multithreading. Prior work on directly using GPU hardware to execute data center applications [36], [37] reports up to 6000$\times$ [37] higher latency than the CPU. Furthermore, accessing I/O resources on GPUs requires CPU co-ordination [37]–[41] and GPUs do not support the rich set of programming languages represented in contemporary microservices [17], hindering programmer productivity.

SIMT-on-SIMD compilers, like Intel ISPC [42], provide a potential path to run SIMT-friendly microservices on CPU SIMD units. This method has the potential to achieve high energy efficiency while leveraging some of the CPU pipeline's latency optimizations by assigning each thread to a SIMD lane. However, this approach has several drawbacks. First, each microservice thread requires more register file and cache capacity than work typically assigned to a single fine-grained SIMD lane, negatively impacting service latency. Second, this approach transforms conditional scalar branches into predicates, limiting the benefit of the CPU's branch predictor. Finally, this method requires a complete recompilation of the microservice code and new ISA extensions for the scalar instructions with no 1:1 mapping in the vector ISA (see Section VI-A for further details).

Table I: CPU vs RPU vs GPU Key Metrics

Metric	CPU	GPU	RPU
Thread/Execution Model	SMT	SIMT	SIMT
General Purpose Programming	✓	✗	✓
System Calls Support	✓	✗	✓
Service Latency	✓	✗	✓
Energy Efficiency (Requests/Joule)	✗	✓	✓

To this end, we propose replacing the CPUs in contemporary data centers with a general-purpose architecture customized for microservices: the Request Processing Unit (RPU). The RPU improves the energy-efficiency of contemporary CPUs by leveraging the frontend and memory system design of SIMT processors, while meeting the single thread latency and programmability requirements of microservices by maintaining OoO execution and support for the CPU's ISA and software stack. Under ideal SIMT-efficiency conditions, the RPU improves energy-efficiency in three ways. First, the 30% of total data center energy spent on CPU instruction supply can be reduced by the width of the SIMT unit (up to 32 in our proposal). Second, SIMT pipelines make use of vector register files and SIMD execution units, saving area and energy versus a MIMD pipeline of equivalent throughput. Finally, SIMT memory coalescing aggregates access among threads in the same warp, producing up to 32$\times$ fewer memory system accesses. Although the cache hit rate for SMT CPUs may be high when concurrent threads access similar code/data, bandwidth and energy demands on both cache and OoO structures will be higher than an OoO SIMT core where threads are aggregated.

Moving from a scalar MIMD pipeline to a vector-like SIMT pipeline has a latency cost. To meet timing constraints, the clock and/or pipeline depth of the SIMT execution units must be longer than that of a MIMD core with fewer threads. However, the SIMT core's memory coalescing capabilities help offset this increase in latency by reducing the bandwidth demand on the memory system, decreasing the queueing delay experienced by individual threads. In our evaluation, we faithfully model the RPU's increased pipeline latency (Section IV) and demonstrate that despite a pessimistic assumption that the ALU pipeline is 4$\times$ deeper in the RPU [43] and that L1 hit latency is $>2\times$ higher, the average service latency is only 44% higher than a MIMD CPU chip.

A well designed software system that is aware of the hardware's aggregating nature and can balance SIMT efficiency with end-to-end request latency is critical to the RPU's success. To meet these demands, we co-design the RPU with a SIMR-aware software system pictured in Figure 2. The RPU executes a general-purpose CPU ISA, supporting all the same functionality as a typical CPU core, but aggregates the use of all its frontend structures over multiple threads. Table I contrasts CPUs, GPUs and the RPU at a high level. At runtime, a SIMR-aware HTTP server groups similar requests together as they enter the microser-

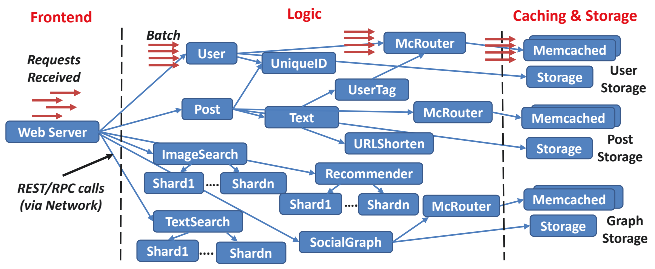


Figure 3: Social network microservice graph studied in this work, similar to [17].

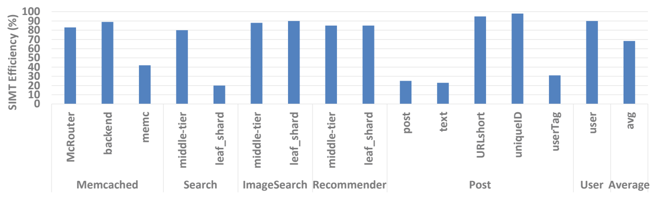


Figure 4: SIMT control efficiency of naive batching for some microservices.

vice graph. To maintain end-to-end latency requirements and keep throughput high, we introduce a similarity-aware batching technique to increase SIMT efficiency, hardware resource tuning to reduce cache and memory contention, SIMR-aware memory allocation to maximize coalescing opportunities, and a system-wide batch split mechanism to minimize latency when requests traverse divergent paths with drastically different latencies.

This work makes the following contributions:

1) We perform the first SIMT-efficiency characterization of microservices using their native CPU binaries. We demonstrate that, given the right batching mechanisms, microservices execute efficiently on SIMT hardware.
2) We propose a new hardware architecture, the Request Processing Unit (RPU). The RPU improves the energy-efficiency and thread-density of contemporary OoO CPU cores by exploiting the similarity between concurrent microservice requests. With a high SIMT efficiency, the RPU captures the single-threaded advantages of OoO CPUs, while increasing requests/joule.
3) We propose a novel software stack, co-designed with the RPU hardware that introduces SIMR-aware mechanisms to compose/split batches, tune SIMT width, and allocate memory to maximize coalescing.
4) On a diverse set of 15 CPU microservices, we demonstrate that the RPU improves requests/joule by an average of 5.7x versus OoO single threaded and SMT CPU cores, while maintaining acceptable end-to-end latency.

II. Background and Motivation

In this section, we detail five key observations from contemporary cloud and microservices that motivate the RPU.

Key Observation #1*: Data center workloads have an abundant number of similar requests*: Public and private data centers receive a significant amount of independent requests from millions of users running the same service code [44]. These requests follow a Single Program Multiple Data (SPMD) pattern that can be efficiently leveraged on SIMT hardware [36], [37], [45].

Key Observation #2*: Microservices reduce the cache required per-thread and minimize control-flow variations between concurrent threads*: In the microservice design paradigm, a monolithic logic tier is broken down into smaller, software-friendly microservices where each is responsible for a small piece of the system. Figure 3 depicts a simple microservice graph for a social network service similar to [17]. Each node in the data center is tasked with many threads all running the same microservice. When monolithic services are disaggregated, divergent control-flow paths are often split into different microservices. That is, *if/else* conditionals in the monolithic service are split into one service for *if* and one service for *else*. Such an organization makes it much more common that concurrent microservices on the same machine traverse exactly the same control-flow path before sending their request to the next microservice. In addition, the per-thread data cache requirement is significantly reduced, as each thread fundamentally does less work. Figure 4 shows the SIMT control flow efficiency of modern microservices, assuming they are batched on arrival into groups of 32 threads. On average, we are able to achieve 68% SIMT efficiency when applying naive batching. In Section III-B1, we propose optimized batching techniques, which bring efficiency to 92%.

Key Observation #3*: Modern data centers already rely on request batching*: In order to enable SIMT execution, requests have to be batched and executed together. Batching can introduce additional latency, however, batching is already heavily used in data centers and employed in at least one microservice on each network path. For example: (1) deep learning inference batches requests to increase accelerator compute throughput [9], [46], (2) key-value store applications, like memcached [47], batch to amortize the network overhead, (3) streaming graph analytics [48] batch to alleviate lock contention, and (4) dynamic power management [49]–[51] applies batching to save power. Therefore, if we apply batching to exploit request similarity, the batching overhead is amortized, as there are already microservices on the same path that employ batching.

Key Observation #4*: In the data center, all throughput gains must be made under a tight latency constraint*: The trade-off between brawny and wimpy cores in the data center is a well-studied problem [52], [53]. However, the use of wimpy cores has not been widely adopted by data center providers [52]. Under the same power budget, wimpy cores can increase throughput [54], but have a higher task execution latency than brawny cores. This increase in

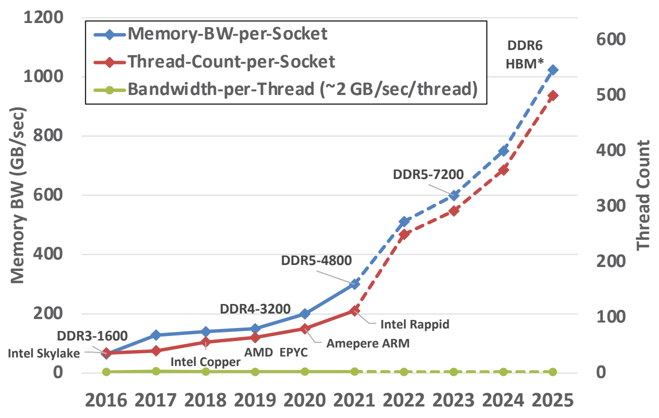


Figure 5: Off-chip DRAM BW and thread scaling.

total request latency makes them ill-suited for the data center's QoS-sensitive workloads. Prior work has argued that energy-optimized systems in the data center must ensure that their single-thread latency is no worse than 2× that of brawny cores [10], [52]. The same argument applies for GPUs, that have high energy-efficiency, but have unacceptably high service latency, 6000× worse than CPUs for SPEC-Web [37], and 10× worse for memcached [36].

Key Observation #5*: Future data center nodes need to increase their on-chip thread count*: Previous academic and industrial work [20], [23]–[28] has shown that current CPUs are inefficient when executing data center workloads as there are many underutilized resources. They suggest that an increase in the number of threads on-chip is necessary to better use these resources [20], [24], [25]. Figure 5 depicts the off-chip bandwidth and thread count per socket scaling in the future. CPU vendors typically ensure 2 GB/sec of DRAM BW per thread. If this is the case, we need to provide up to 256 threads per socket with DDR5 [55]–[58] and 512 threads with DDR6 [59] and HBM [60] to utilize the available off-chip BW. The industry standard to increase on-chip throughput is by adding more chiplets [30], [31], cores [32], [33] and increasing the SMT degree [34], [35]; however, we argue that introducing SIMT to OoO CPU cores will provide a more energy-efficient mechanism to scale on-chip throughput.

Given these five observations, we design our RPU hardware and software system to exploit the similarity among requests in microservices through intelligent batching. The RPU's OoO SIMT frontend is able to meet the latency constraints of contemporary services, while improving upon the energy-efficiency and thread-density of modern CPUs. In the next section, we discuss our system's design.

III. SIMR System

Figure 2 presents a high level overview of our SIMR system. Groups of independent Remote Procedure Call (RPC) or HTTP requests are received by our SIMR-Aware server. The server (❶ in Figure 2) groups requests into a batch based on each request's Application Program Interface (API) similarity and argument size. The batches in the RPU are analogous to warps in a GPU. Our batch size is tunable based on resource contention, desired QoS, arrival rate and system configuration (Section III-B explores these parameters). Then, the server launches a service request to the RPU driver and hardware. The RPU hardware (Ⅱ) executes the batch in lock-step fashion over the OoO SIMT pipeline (Section III-A).

A. RPU Hardware

Figure 6 presents a detailed overview of our RPU hardware. Our RPU chip contains multiple RPU cores, and a few CPU cores. The role of the CPU cores is to run the OS process, HTTP server, and RPU driver while the RPU cores run the microservices requests' workload. Each RPU core is similar to a brawny OoO CPU core, except hardware is added (highlighted in green) to perform multithreading in a SIMT fashion. The design philosophy of the RPU is that the area/power savings gained by SIMT execution and amortizing frontend (e.g., OoO control logic, branch predictor, fetch&decode), are used to increase the thread context and throughput at the backend (❶ in Figure 6, e.g., scalar/SIMD physical register file (PRF), execution units, and cache resources); thus we still maintain the same area/power budget and improve overall throughput/watt. It is worth noting that the RPU thread has the same coarse granularity as the CPU thread, such that the RPU thread has a similar thread context of integer and SIMD register file space. In addition, all execution units, including the SIMD engines, are increased by the number of SIMT lanes.

OoO SIMT Pipeline: When merging the RPU's SIMT pipeline with speculative, OoO execution, we assume the following design principles. First, the active mask is propagated with the instruction throughout the entire pipeline (❷). Therefore, register alias table (RAT), instruction buffer and reorder buffer entries are extended to include the active mask (AM). Second, to handle register renaming of the same variable used in different branches, a micro-op is inserted to merge registers from the different paths [61], [62]. Third, the branch predictor operates at the batch (or warp) granularity, i.e., only one prediction is generated for all the threads in a batch. When updating the branch history, we apply a majority voting policy of branch results (❸) to optimize for the most common control flow. The instructions from mispredicted threads are flushed at the commit stage and the corresponding PCs and active mask are updated accordingly. Adding majority voting circuitry before branch prediction increases branch execution latency and energy. We account for these overheads in our evaluation, detailed in Section IV.

Control Flow Divergence Handling: To address control flow divergence, a hardware SIMT convergence optimizer (❹) is employed to serialize divergent paths [63], [64]. The optimizer relies on stack-less reconvergence with a *MinPC* heuristic policy [64]–[66]. In this scheme, each thread has its

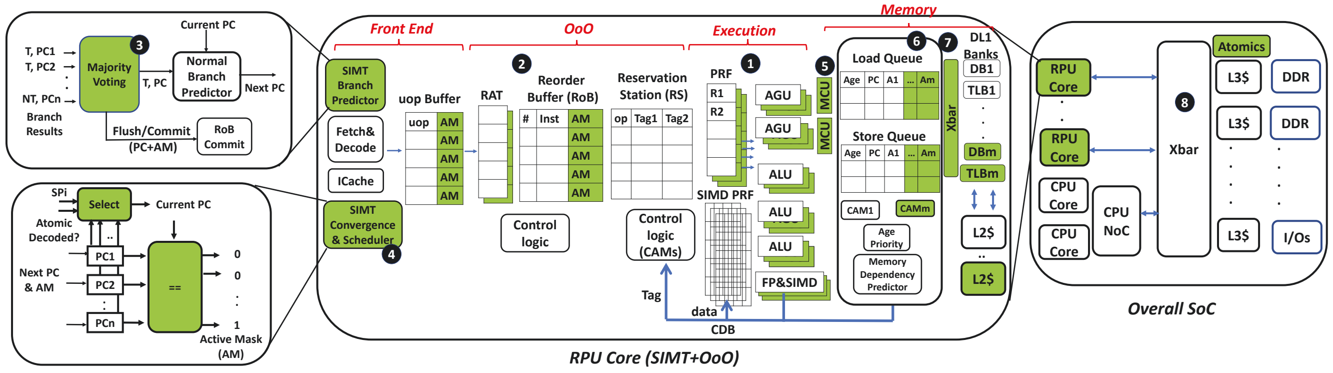
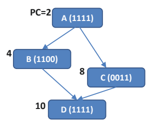


Figure 6: RPU hardware overview. Changes to the OoO core needed to support SIMR execution are highlighted in green.

```
1. / / BBA Basic Block "A"
2. if ( x > 0)
3. {
4.     / / BBB
5. }
6. else
7. {
8.     / / BBC
9. }
10. / / BBD
```

Divergent code example



Control Flow with Active Mask

PC1	PC2	PC3	PC4	Current PC (min)	Active mask	Next PC (BP)
2	2	2	2	2	1111	4
4	4	4(F)	4(F)	4	1111	6
6	6	8	8	6	1100	10
10	10	8	8	8	0011	10
10	10	10	10	10	1111	12

MinPC selection policy

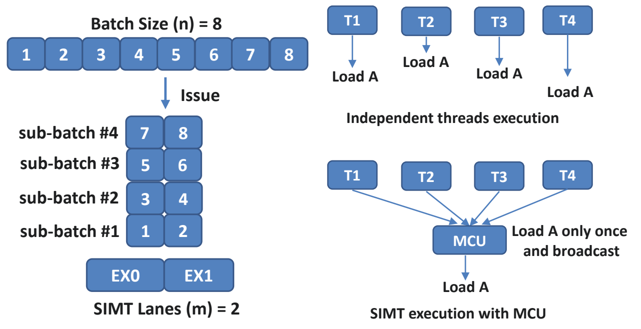
Figure 7: Stack-less convergence analysis example with MinPC policy. Assuming instruction size = 2, and 1-cycle miss penalty, F:flush.

own Program Counter (PC) and Stack Pointer (SP), however, only one current PC (i.e., one path) is selected at a time. The selected PC is given to the basic block whose entry point has the lowest address. The *MinPC* heuristic relies on the assumption that reconvergence points are found at the lowest point of the code they dominate [66]. For function calls we assume a *MinSP* policy [64] which gives priority to the deepest function call or we set a convergence barrier at the instruction following the procedure call.

Figure 7 contains a simple example of *MinPC* policy analysis and shows how PC selection interacts with divergent control flow. When threads execute divergent control flows, the paths are serialized, and each path is associated with the current PC and corresponding active mask. This control flow divergence serialization overhead is minimized by intelligent batching techniques which we describe in Section III-B1. The *MinPC* strategy has been found to achieve 100% accuracy [64] to determine correct reconvergence points for GPGPU workloads and up to 94% for CPU SPECint workloads [65]. Even in the rare cases where the policy misses the correct reconvergence points, it still reconverges not too far behind and achieves overall good SIMT control efficiency (Section III-B1). The stack-less reconvergence approach is transparent to the compiler and ISA, and can handle indirect branches without profiling or virtual ISA support. This differs from stack-based approaches that are widely used in modern GPUs [63], [67] which require compiler-assisted static analysis to determine correct reconvergence points and ISA support to update the hardware stack [68] and list all the targets of indirect branches [69].

Running threads in lock-step execution and serializing divergent paths can induce deadlocks when programs employ inter-thread synchronization [70]–[72]. There have been several proposals to alleviate the SIMT-induced deadlock issue on GPUs. All of the proposed solutions rely on multi-path execution to allow control flow paths that are not at the top of the SIMT stack to make forward progress. In the RPU, when an active thread's PC has not been updated for k cycles and there have been at least b atomic instructions decoded within the k-cycle window (an indication for spin locking by other selected threads), then the waiting thread is prioritized and we switch to the other path for t cycles. Otherwise, the default *MinSP-PC* is applied. Multi-path interleaving requires partitioning the return address stack (RAS) in the branch prediction unit to support multiple control flows [73]. Another issue is that the *MinSP-PC* selection policy can increase the branch prediction latency, hindering pipeline utilization. To mitigate this issue, we can leverage techniques proposed for complex, multi-cycle branch history structures, such as hierarchical or ahead pipelining prediction [74].

Sub-batch Interleaving: Previous work [20], [23]–[25] show that data center workloads tend to exhibit low IPC per thread (a range of 0.5-1 out of 5, and up to 1.75 IPC when SMT enabled [20]), due to long memory latency at the back-end and instruction fetch misses at the frontend [20], [26]. To increase our execution unit utilization and ensure a reasonable backend execution area, we implement sub-batch interleaving [75] as depicted in Figure 8a. By decreasing the number of SIMT lanes (m) per execution unit to be a fraction of batch size (n), we issue threads over multiple cycles. Sub-batch interleaving along with OoO scheduling can hide nanosecond-scale latencies efficiently, increasing IPC utilization. Another advantage of sub-batch interleaving is that we can skip issue slots of non-active threads to mitigate control divergence penalties and support smaller batches of execution [75]. To hide longer microsecond-scale latencies [76]–[78], multiple batches can be interleaved via hardware batch scheduling in a coarse-grain, round-robin manner with zero-overhead context switching. However, studying multi-batch scheduling to hide microsecond-scale latency is beyond the scope of this work.



(a) Sub-batch interleaving (b) MCU

Figure 8: Sub-batch interleaving and MCU to improve latency hiding and memory throughput efficiency respectively.

Memory Coalescing: To improve memory efficiency, a low-latency memory coalescing unit (MCU) is placed before the load and store queues (❺). As described in Figure 8b, the MCU is designed to coalesce memory accesses to the same cache line from threads in a single batch, making better use of cache throughput and avoiding cache access serialization. The MCU filters out accesses to shared inter-request data structures that might exist in the heap or data segments [25]. To balance the need for a low cache hit latency with avoiding divergent access serialization, the MCU only detects the two most common memory coalescing scenarios: when all threads access the same word, or when threads access consecutive words from the same cache line. This is unlike the complex sub-batch sharing in GPU data coalescing [63], [79] which increases memory access latency to detect more complex locality patterns [80].

LD/ST Unit: In our MCU, if neither simple pattern is detected, the number of accesses generated will equal the number of active SIMT lanes. All accesses from the same instruction will allocate one row in the load or store queue (❻), sharing the same PC and age fields/logic, and thus amortizing the memory scheduling and dependence prediction [81] overhead. Figure 9 depicts the LD/ST unit (LSU) structure in more detail. The entries of the RPU's LD/ST queues are expanded such that each row can contain as many addresses as there are SIMT lanes. Further, we assign an independent content-address memory (CAM) for each lane to account for in-parallel store-to-load forwarding. For coalesced accesses, only one slot in the entry (entry#0) is allocated and broadcast for CAM comparisons. To save area, we do not preserve the loaded value in the load queue; instead, we write the return value to the register file directly and set the corresponding valid bit. Therefore, the load instruction is completed, and the tag is broadcast when all the slots in the entry are valid and completed.

Cache and TLB: To serve the throughput needs of many threads while achieving scalable area and energy consumption, the RPU uses a banked L1 cache. The load/store queues are connected to the L1 cache banks via a crossbar (❼). To ensure TLB throughput can match the L1 throughput,

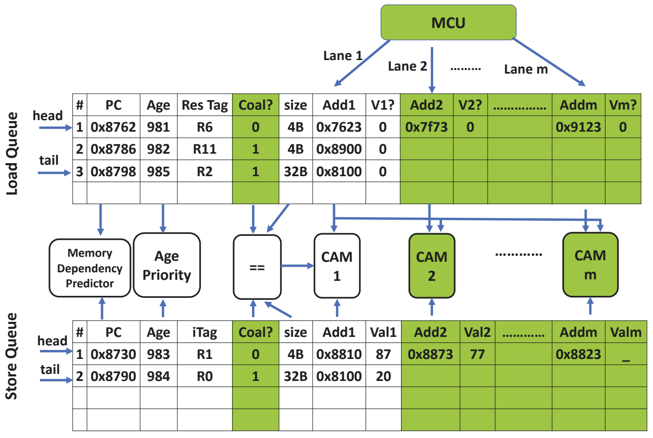

MCU
Lane 1
Lane 2
....
Lane m
Load Queue
head
tail
# PC Age Res Tag Coal? size Add1 V1? Add2 V2? .... Addm Vm?
1 0x8762 981 R6 0 4B 0x7623 0 0x7f73 0 0x9123 0
2 0x8786 982 R11 1 4B 0x8900 0
3 0x8798 985 R2 1 32B 0x8100 0
Memory Dependency Predictor
Age Priority
==
CAM 1
CAM 2
....
CAM m
Store Queue
head
tail
# PC Age iTag Coal? size Add1 Val1 Add2 Val2 .... Addm Valm
1 0x8730 983 R1 0 4B 0x8810 87 0x8873 77 0x8823 _
2 0x8790 984 R0 1 32B 0x8100 20

Figure 9: RPU's LD/ST Unit

each L1 data bank is associated with a TLB bank. Since the interleaving of data over cache banks is at a smaller granularity than the page size, TLB entries may be duplicated over multiple banks. This duplication overhead reduces the effective capacity of the DTLBs, but allows for high throughput translation on cache+TLB hits. As a result of the duplication, all TLB banks are checked on the per-entry TLB invalidation instructions [82]. Sections III-B3 and III-B4 discuss how we alleviate contention to preserve intra-thread locality and achieve acceptable latency via batch size tuning and SIMR-aware memory allocation.

Weak Consistency Model+NMCA: To exploit the fact that requests rarely communicate and exhibit low coherence, read-write sharing or locking [24], [25], as well as extensive use of eventual consistency in data center [83], we design the memory system to be similar to a GPU, i.e., weak memory consistency with non-multi-copy-atomicity (NMCA) [1]. RPU implements a simple, relaxed coherence protocol with no-transient states or invalidation acknowledgments, similar to the ones proposed in HMG [88] and QuickRelease [89]. That is, cache coherence and memory ordering are only guaranteed at synchronization points (i.e., barriers, fences, acquire/release), and all atomic operations are moved to the shared L3 cache. Therefore, we no longer have core-to-core coherence communication, and thus we replace the commonly-used mesh network in CPUs with a higher-bisection-bandwidth, lower-latency core-to-memory crossbar (❽). Furthermore, NMCA permits threads on the same lane to share the store queue, reducing the complexity of having a separate store queue per thread [85]. This relaxed memory model allows our design to scale the number of threads efficiently, improving thread density by an order of magnitude.

1) CPU vs GPU vs RPU: Table II lists the key architectural differences between CPUs, GPUs and our RPU. The RPU takes advantage of the latency-optimizations and

[1]In fact, some CPU ISAs, like ARMv7 [84], [85] and POWER [86], already support a weak consistency model with non-multi-copy-atomicity in their specifications [87].

Table II: CPU vs GPU vs RPU architecture differences

Metric	CPU	GPU	RPU
Core model	**OoO**	In-Order	**OoO**
Freq	**High**	Moderate	**High**
ISA	**ARM/x86**	HSAIL/PTX	**ARM/x86**
Programming	**General-Purpose**	CUDA/OpenCL	**General-Purpose**
System Calls	**Yes**	No	**Yes**
Thread grain	**Coarse grain**	Fine grain	**Coarse grain**
TLP per core	Low (1-8)	Massive (2K)	Moderate (8-32)
Thread model	SMT	**SIMT**	**SIMT**
Consistency	Variant	**Weak+NMCA**	**Weak+NMCA**
Coherence	Complex	**Relaxed Simple**	**Relaxed Simple**
Interconnect	Mesh	**Crossbar**	**Crossbar**

Table III: CPU inefficiencies in the data center

Data center characteristics & CPU inefficiency	**RPU's mitigation**
Request similarity [37] & high frontend power consumption [11]	SIMT execution to amortize frontend overhead
Inter-request data sharing [25]	Memory coalescing and an increase in the number of threads sharing private caches
Low coherence/locks [24], [25] and eventual consistency [83]	Weak memory ordering, relaxed coherence with non-memory-copy-atomicity & higher bandwidth core-to-memory interconnect
Low IPC due to frequent frontend stalls and memory latency [20], [23]–[26]	Multi-thread/sub-batch interleaving
DRAM & L3 BW are underutilized, data prefetchers are ineffective [21], [24], [25], [27]	High thread level parallelism (TLP) to fully utilize BW
Microservice/nanoservice have a smaller cache footprint [17]	High TLP and decrease L1&L2 cache capacity/thread

programmability of the CPU while exploiting the SIMT efficiency and memory model scalability of the GPU. Finally, Table III summarizes a set of data center characteristics that create inefficiencies in CPU designs and how the RPU mitigates them.

2) An Examination of SMT vs SIMT Energy Efficiency: This subsection examines why the RPU's SIMT execution is able to outperform MIMD SMT hardware for data center workloads. Equation 1 presents an analytical computation of the RPU's energy efficiency (EE) gain over the CPU. In Equation 1, *n* is the RPU batch size, *eff* is average RPU SIMT efficiency, and *r* is the ratio of memory requests that exhibit inter-thread locality within a single SIMR batch. CPU energy is divided into frontend+OoO overhead (including fetch, decode, branch prediction, OoO control logic and load/store queue), execution (including register reading/writing and instruction execution), memory system (including private and L3 caches), and static energies.

$$EE = \frac{CPU_{Energy}}{RPU_{Energy}} = \frac{Exec_{Energy} + Mem_{Energy} + FE\_OoO_{Energy} + Static_{Energy}}{Exec_{Energy} + (1-r)Mem_{Energy} + \frac{1}{n*eff} * [r*Mem_{Energy} + FE\_OoO_{Energy} + Static_{Energy}] + SIMT_{Overhead}} \quad (1)$$

In Equation 1, the RPU's energy consumption in frontend and OoO overheads are amortized by running threads in lock step; hence the energy consumed for instruction fetch, decode, branch prediction, control logic and CAM tag accesses [90] for register renaming, reservation station,

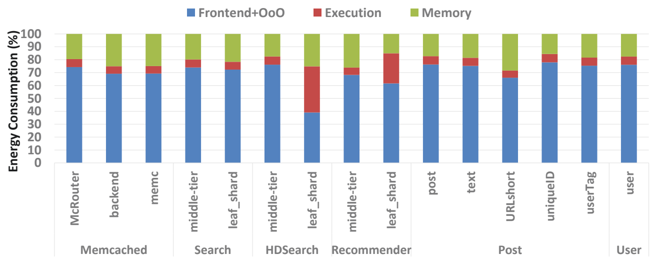


Figure 10: Dynamic energy consumption breakdown per pipeline stage as a percentage of total CPU core energy.

register file control, and load/store queue are all consumed only once for all the threads in a single batch (see Figures 6 and 9). In scalar CPU designs, the frontend and OoO overheads have to be consumed for each thread. Even with SMT, the entire CPU pipeline is partitioned among the simultaneous threads. Threads on the same core are executed independently [34], [35], [91], which fails to exploit thread similarity and increases single thread latency.

Coalesced memory accesses are also amortized in the RPU by generating and sending only one access for the batch to the memory system. While private cache hits and MSHR merges can filter out some of these coalesced accesses in a SMT design, the programmer must guarantee that simultaneous threads are launched and progress together in order to capture this inter-thread data locality [92], [93] and still must pay the energy cost of multiple cache accesses. Furthermore, since SIMT can execute more threads/core given the same area constraints, the reach of its locality optimizations is wider.

The final metric SIMT execution amortizes is static energy. The RPU improves throughput/area and has a smaller SRAM budget/thread compared to an SMT core. The RPU introduces an energy overhead ($SIMT_{Overhead}$ in Equation 1) to account for the SIMT convergence optimizer, majority voting circuit, active mask propagation, MCUs, larger caches and multi-bank L1/L2 arbitration. However, at high SIMT efficiency, the energy savings from the amortized metrics greatly outweigh the SIMT management overhead.

Figure 10 shows the energy consumption breakdown per pipeline stage of our studied microservices when running on CPU (Section IV details our experimental methodology). As shown in the figure, workloads consume a considerable amount of energy at the frontend and OoO stages, with an average of 73% [11], [15]. The *HDSearch-leaf* and *Recommender-leaf* are the exception with 39% and 60% of energy consumed on frontend+OoO respectively. These workload contain fully SIMD vectorized functions; therefore, the backend consumes a large fraction of the energy. The memory subsystem consumes 20% of energy on average. By substituting the values in Equation 1 with the amortized components, which consume 50-90% of the total CPU energy, an anticipated 2-10x energy efficiency gain can be achieved with the RPU when SIMT efficiency is high and

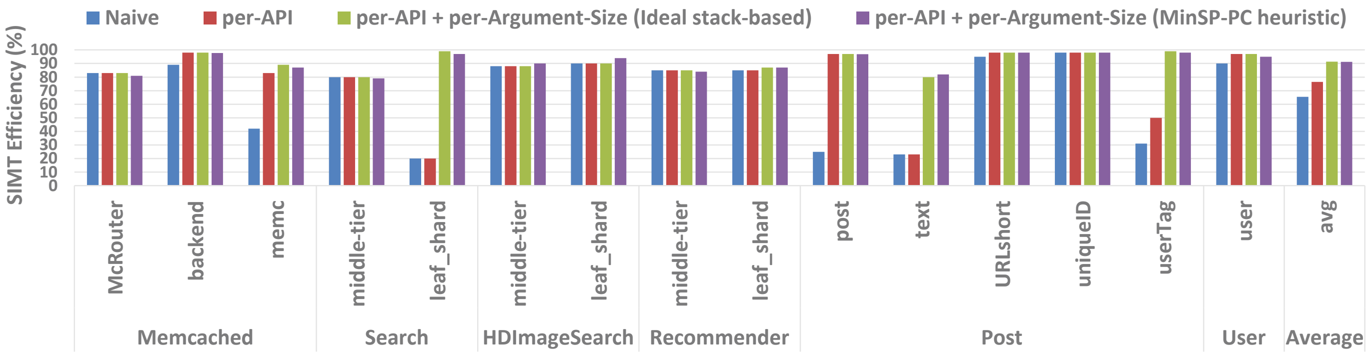
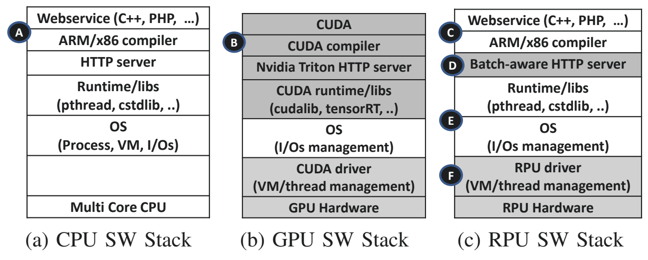


Figure 11: SIMT control flow efficiency with different request batching policies (Batch Size = 32)



Webservice (C++, PHP, ...)
ARM/x86 compiler
HTTP server
Runtime/libs (pthread, cstdlib, ..)
OS (Process, VM, I/Os)
Multi Core CPU
CUDA
CUDA compiler
Nvidia Triton HTTP server
CUDA runtime/libs (cudalib, tensorRT, ..)
OS (I/Os management)
CUDA driver (VM/thread management)
GPU Hardware
Webservice (C++, PHP, ...)
ARM/x86 compiler
Batch-aware HTTP server
Runtime/libs (pthread, cstdlib, ..)
OS (I/Os management)
RPU driver (VM/thread management)
RPU Hardware
A
B
C
D
E
F
(a) CPU SW Stack
(b) GPU SW Stack
(c) RPU SW Stack

Figure 12: Hardware/Software Stack of CPU vs GPU vs RPU for microservices programming

accesses are frequently coalesced. This anticipated energy efficiency is aligned with previous work [94] which studied energy efficiency when vectorizing data-parallel workloads (PARSEC) on CPU hardware.

In the next section we experimentally show the amount of SIMT control and memory efficiency present in microservice workloads and explore the effect of different batch sizes.

B. SIMR Software Stack

Figure 12 compares the RPU's software (SW) stack, to that of the CPU and GPU. GPU computing (B in Figure 12) generally requires the programmer to use a specialized language, like CUDA, and (in the case of NVIDIA) uses a closed-source compiler, runtime, driver, and ISA. These all restrict programmer productivity. While GPUs have been successful accelerating deep learning inference, they are poorly suited for workloads with middling parallelism and tight deadlines.

Microservice developers typically use a variety of high-level open-source programming languages and libraries (A). For the RPU, we maintain the traditional CPU software stack (C, E), changing only the HTTP server, driver and memory management software. The RPU is ISA-compatible with the traditional CPU.

The role of our HTTP server (D) is to assign a new software thread to each incoming request [95], [96]. The SIMR-aware server groups requests in a batch based on each request's Application Program Interface (API) similarity and argument size (see Section III-B1), then sends a *service* launch command for the batch to the RPU driver with pointers to the thread contexts of these requests.

The RPU driver (F) is responsible for runtime batch scheduling and virtual memory management. The driver overrides some of the OS system calls related to thread scheduling, context switching, and memory management, optimizing them for batched RPU execution. For example, context switching has to be done at the batch granularity (Section III-B5), and memory management is optimized to improve memory coalescing opportunities at runtime (Section III-B2).

To ensure efficient SIMT execution, the software stack's primary goals are to: (1) minimize *control flow divergence* by predicting and batching requests' control flow (Section III-B1), (2) reduce *memory divergence* and alleviate cache/memory contention with batch tuning and SIMR-aware virtual memory mapping (Sections III-B2, III-B3, III-B4), and (3) alleviate *network/storage divergence* through system-wide batch splitting (Section III-B5).

1) SIMR-Aware Batching Server: A key aspect to achieve high energy efficiency is to ensure batched threads follow the same control flow, and thus minimize control divergence. To achieve this, we need to group requests that have similar characteristics. Thus, we employ two heuristic-based proof-of-concept batching techniques. First, we group requests based on API or RPC calls. Some microservices may provide more than one API, for example, *memcached* has *set* and *get* APIs, *post* provides *newPost* and *getPostByUser* calls. Therefore, we batch requests that call the same procedure to ensure they will execute the same source code. Second, we group requests that have similar argument/query length. For example, when calling the *Search* microservice, requests that have long search query (i.e., more words) are grouped together as they will probably have more work to do than the smaller ones. Figure 11 shows the SIMT efficiency (i.e., = #scalar-instructions / (#batch-instructions $\times$ batch-size)) for naive batching (based on arrival time) and an optimized per-API and per-argument batching. We demonstrate both the ideal reconvergence with stack-based IPDOM analysis [63], [67] and *MinSP-PC* heuristic policy [64]. We assume a batch size of 32 requests for all microservices and we calculate the average over 75 batches

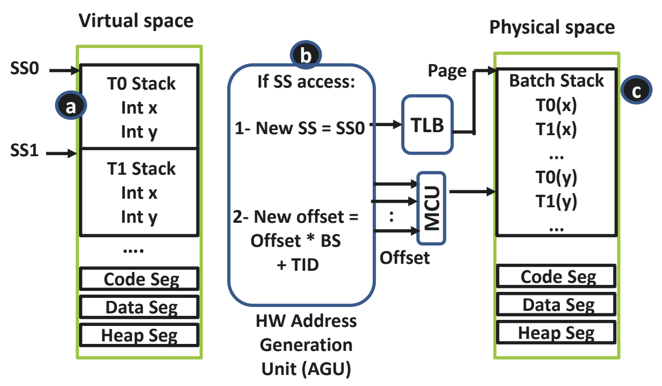


Figure 13: Stack Segment (SS) coalescing (physical stack page size = virtual page size * batch size) with 4-byte interleaving. BS:batch size, TID: thread ID.

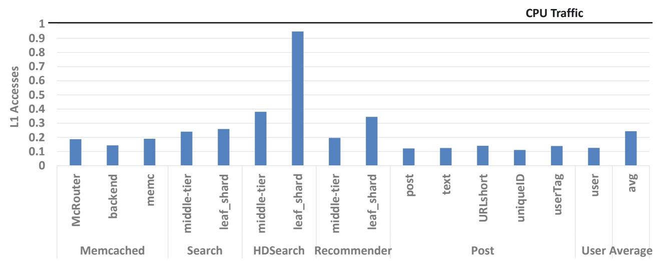


Figure 14: RPU L1 accesses, normalized to CPU accesses

(2400 requests). As shown in Figure 11, batching per-API improves SIMT efficiency for many microservices, up to 2x improvement in *memcached*, and 4x in *Post* microservices. When taking into account per-argument length batching, the overall SIMT efficiency is further improved by 20% on average and up to 5x better on the *Search-leaf* and *post-text* microservices. In total, the stack-based analysis is able to achieve 92% SIMT efficiency. Interestingly, *MinSP-PC* is not far behind with an efficiency of 91% on average. In some microservices the heuristic even shows 1-2% higher efficiency due to eliminating the redundant execution of reconvergnce instructions in the stack-based approach [64].

We achieve this SIMT efficiency while making the following assumptions. First, some of these microservices are not well optimized and employ coarse-grain locking which affects our control efficiency negatively due to critical section serialization and lock spinning. In practice, optimized data center workloads rely on fine-grain locking to ensure strong performance scaling on multi-core CPUs [25], [96]. In our experiments we assume threads that access different memory regions within a data structure use fine-grained locks for synchronization. We also assume that a high-throughput, concurrent memory manager is used for heap segment allocation [97]–[99] rather than the C++ glibc allocator that uses a single shared mutex. Finally, *HDSearch-midtier* contains a data-dependent control flow in which one side of a branch contains more expensive code. To improve SIMT efficiency in such scenarios, we use speculative reconvergence [100] to place the IPDOM synchronization point at the beginning of the expensive branch.

2) Stack Segment Coalescing: Similar to the local memory space in GPUs [63], [101], Figure 13 depicts how the RPU driver and TLB hardware allocate and map stack memory from different threads in the same batch to minimize memory divergence. The interleaving is static and transparent to the compiler and the programmer. When the runtime system calls *mmap* to allocate a new stack segment for a thread [102], [103], we ensure that the stack segments for all the threads in a batch are contiguous (ⓐ in Figure 13). In hardware, we detect accesses to stack addresses and apply an interleaved data mapping (ⓑ), such that stack segments from different threads are interleaved every 4 bytes in the physical address space (ⓒ). The RPU's address generation unit overrides the stack base of all active threads with the stack base of thread 0, thus we only need one TLB translation per stack access. A hardware offset mapping uses the thread ID (TID) of the accessing thread as an index into the SS0 space to determine where the value resides in physical memory. However, this hard mapping prevents threads from accessing other threads' stack data, which is allowed in CPU programming. To resolve this issue, we calculate the target stack segment TID of each access based on the access' virtual segment address, i.e. $TargetTID = (SSi - SS0)/StackSize$, exploiting the fact that stacks are allocated consecutively in the virtual space. If the accessing thread has permission to access the target thread's stack (discussed further in Section VI-C), then the TargetTID is used, allowing inter-thread stack accesses. Note that GPU programming languages avoid this issue by making stack values thread-local.

Coalescing Results: Figure 14 demonstrates the effectiveness of our stack interleaving and heap memory coalescing policies (previously described in Section III-A and Figure 8b). Figure 14 plots the total number of L1 accesses in the RPU, normalized to a MIMD CPU, when both are executing 640 threads. The RPU's 32-thread batches generate on average 4x fewer accesses than the CPU. The causes of this traffic reduction are two-fold. First, many of our middle tier microservices contain significant stack segment accesses (up to 90% in the Post microservices) caused by frequent procedure/system calls, push/pop argument passing, and reading/writing local variables. Our stack segment interleaving technique coalesces all these accesses and generates less traffic compared to the CPU. For example, pushing an 8-byte address in each thread of a 32-thread batch onto the stack generates 8 accesses (8B x 32 threads / 32B cache lines); however, in the CPU 32 accesses are generated.

Second, microservices typically share some global data structures and constant values in the heap and data segments [25] respectively. In the RPU, accesses to this shared data are coalesced within the MCU and loaded once for all the threads in a batch, improving L1 data throughput.

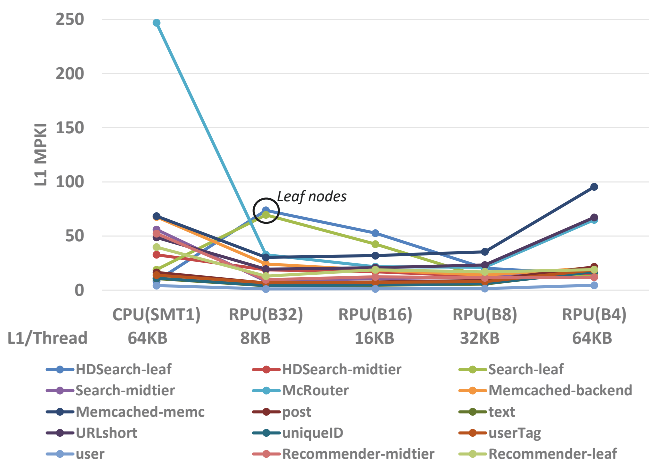
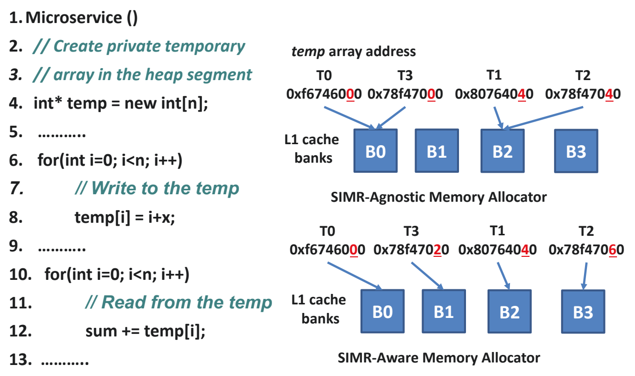


Figure 15: L1 MPKI of a single threaded CPU vs RPU with different batch sizes (32, 16, 8, 4).

While traffic reduction is significant in many cases, back-end data-intensive microservices, like *HDSearch*, still exhibit high traffic as each thread contains private data structures in the heap with little sharing, resulting in frequent divergent heap accesses.

3) Batch Size Tuning and Memory Contention : Previous work [17] shows that micro and nanoservices typically exhibit a low cache footprint per thread, as services are broken down into small procedures and read-after-write interprocedure locality is often transferred to the system network via RPC calls. To exploit this fact, we increase the number of threads per RPU core compared to traditional CPUs. Figure 15 shows the L1 MPKI of a single threaded CPU with 64KB of L1 cache and an RPU with different batch sizes (32, 16, 8, 4) and 256KB of L1 cache. Interestingly, many of our microservices can run at a batch size of 32 threads and require only 8KB/thread without thrashing the L1 cache. More importantly, for these microservices, the L1 MPKI is significantly improved compared to the CPU. This is because memory coalescing reduces the overall number of L1 accesses as well as the number of misses. As the batch size decreases, the coalescing efficiency is reduced.

On the other hand, some leaf node microservices, like *HDsearch-leaf* and *Search-leaf*, have high L1 MPKI compared at a batch size of 32. These are data-intensive services, exhibiting a larger intra-thread locality footprint due to divergent heap segment accesses, read-after-write temporary data and prefetch buffer to hide long memory latency. However, they show low MPKI when we throttle the batch size to 8 (see Figure 15). We make similar observations for TLB and memory system contention when applying batch size tuning. Therefore, we run all our microservices at a batch size of 32, except for these data-intensive services, which are executed with a batch size of 8. Due to sub-batch interleaving, running at this smaller batch size does not affect our execution unit utilization. Regardless of batch size, our RPU hardware is configured with 8 SIMT lanes (Section IV), as such, an


```
1. Microservice ()
2.  // Create private temporary
3.  // array in the heap segment
4.  int* temp = new int[n];
5.  ..........
6.  for(int i=0; i<n; i++)
7.        // Write to the temp
8.        temp[i] = i+x;
9.  ..........
10. for(int i=0; i<n; i++)
11.       // Read from the temp
12.       sum += temp[i];
13. ..........
```

(a) A divergent heap segement accsses of temp data (b) SIMR-aware memory allocation assuming 4 banks 32B interleaving

Figure 16: SIMR-aware memory allocator.

8-thread batch can fully utilize the pipeline, even though amortization suffers versus a 32-thread batch. After inspecting the *HDsearch-leaf* source code, we found that we could reduce the L1 cache footprint of the workload by eliminating some unnecessary data copies and employing function fusion (analogous to kernel/layer fusion in GPU and DL); however, we decided not to alter the program in our experiments.

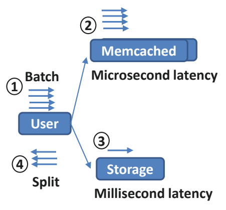
Selecting the right batch size is influenced by many other factors, e.g. the request arrival rate, desired QoS, and system configuration. As widely practiced by data center providers [20], an offline configuration can be applied to tune the batch size for a particular microservice. The time overhead to form a batch size of 8-32 requests is well tolerated by data center providers and matches those used in Google and Facebook's batching mechanisms for deep learning inference [9], [46].

4) SIMR-Aware Memory Allocation : Divergent accesses to the heap have the potential to create bank conflicts in the RPU's multi-bank L1 cache. Figure 16a depicts a frequent code pattern in our microservices. The program dynamically allocates a thread-private temporary array on the heap (line#4), fills the array with intermediate results in a linear fashion (line#8), and reads from this array to process the data (line#12). The top section of Figure 16b shows the behavior of the default C++ SIMR-agnostic CPU allocator. We assume a virtually-indexed L1 cache as is widely employed by CPU designs. Thus, the memory allocator may assign addresses to the *temp* array that result in significant bank conflicts. One solution for this is to change the address mapping of the heap segment [104] to interleave elements accessed by parallel threads, similar to our stack segment interleaving. However, this type of interleaving is ill-suited for heap accesses, which are less structured than stack accesses. Another solution is to rely on hardware-based xor-ing hashing [105], [106], however our experiments show that it is ineffective to alleviate bank conflicts.

To this end, we address this problem by proposing a new SIMR-aware memory allocator that the RPU driver can provide as an alternative and overrides the memory allocator

```
1. Procedure get_user(int userid)
2.   /* first try the cache */
3.   data = memcached_fetch("userrow:" + userid)
4.   if not data       /* SIMT Divergence*/
5.        /* not found : request database */
6.        data = db_select("SELECT * FROM users
7.         WHERE userid = ?", userid)
8.        /* then store in cache until next get */
9.        memcached_add("userrow:" + userid,
10.        data)
11.  end          /* SIMT Reconvergence Point*/
12.  return data
```

(a) Code snapshot for network divergence

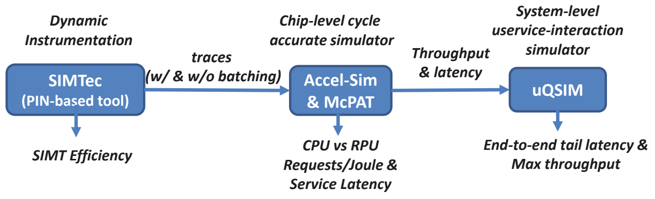


(b) Batch split

Figure 17: Batch split technique for control flow divergence when a path contains long network/storage blocking event.

used by the run-time library through *LD_PRELOAD* Linux utility [107], [108]. Our proposed memory allocator, demonstrated in the bottom image of Figure 16b, avoids data interleaving for the heap segment. Instead, the key idea is to take into account that data are already interleaved every n bytes over L1 banks (n=32B in our baseline). Therefore, if we ensure that the start address of every new memory allocation per thread follows the condition *(start_address%(n*tid) = 0)*, then accesses to the private data structure will be conflict-free for all consecutive data accesses, as shown in Figure 16b. The overhead of this method is the unused few bytes at the start of each data allocation to ensure the alignment constraint (around 896 bytes for an 8-thread allocation). This memory fragmentation is amortized with large memory allocation sizes.

5) System-Level Batch Splitting : In the RPU, context switching is done at the batch granularity, either all threads in a batch are running or all the threads in the batch are switched out. When RPU threads are blocked due to an I/O event, the RPU driver groups the I/O received interrupts and wakes the all the threads in the same batch at the same time to handle their interrupts and continue lock-step execution. However, requests within the same batch can follow different control paths, and paths may be of different lengths. For memory and nanosecond-scale latencies, the paths synchronize at the IPDOM reconvergence point. However, if one path contains significantly longer millisecond-scale latency (e.g., a request to storage or the network), this can hinder the threads on the other path, extending the average latency. Figure 17a illustrates a frequently-used design pattern in microservice development, in which back-end storage accesses are cached in a fast in-DRAM key-value store, like memcached (line#3 in Figure 17a). If the user request hits in the microsecond-scale latency memcached, the request returns immediately to the client (line#12); otherwise, it has to access the millisecond-scale storage, update the cache, and send the result back (lines#5-10). If the hit requests have to wait for the misses at the reconvergence point (line#11), then the storage latency will



Dynamic Instrumentation
SIMTec (PIN-based tool)
SIMT Efficiency
traces (w/ & w/o batching)
Chip-level cycle accurate simulator
Accel-Sim & McPAT
CPU vs RPU Requests/Joule & Service Latency
Throughput & latency
System-level uservice-interaction simulator
uQSIM
End-to-end tail latency & Max throughput

Figure 18: End-to-end experimental setup.

dominate the total average latency.

To avoid this issue we propose a batch splitting technique, depicted in Figure 17b, in which we split the batch and allow multi-path execution [109] for hit and miss requests. The batch is subdivided into two batches, one for the hit requests to continue execution beyond the reconvergence point (④ in Figure 17b), and the other for blocked requests accessing storage (③). The driver copies and saves the architectural state and call stack for the blocked requests. Note that in cycle-level multipath execution on GPUs [109]–[111], divergent paths still ultimately converge and resources are not freed until all paths are complete. In SIMR batch splitting the fast completing path can be allowed to continue and finish execution, while the slower blocked path is context switched out, freeing up resources for other requests.

A hardware-based timeout or software-based hint can be used to determine the splitting decision. Although batch splitting reduces control efficiency, as the miss requests will continue execution alone, we can still batch these orphan requests at the storage microservice and form a new batch to be executed with a full SIMT active mask. We believe there is a wide space of future work to analyze the microservice graph for splitting and batching opportunities.

IV. Experimental Setup

Workloads: We study a microservice-based social graph network as depicted in Figure 3, similar to the one represented in the DeathStarBench suite [17]. *Search*, *HDImageSearch*, *Recommender*, and *McRouter* are adopted from the μsuite benchmarks [16]. We use the input data associated with the suite. The microservices use diverse libraries, including C++ stdlib, Intel MKL, OpenSSL, FLANN, Pthread, zlib, protobuf, gRPC and MLPack. The *Post* and *User* microservices are adopted from the DeathStarBench workloads [17] and social graph is from SAGA-Bench [112]. The microservices have been updated to interact with each other via Google's gRPC framework [113], and they are compiled with the -O3 option and SSE/AVX vectorization enabled. While the RPU can also execute other HPC/GPGPU applications that exhibit the SPMD pattern, like OpenMP and OpenCL, we limit the focus of our study to microservice workloads. Section VI-D discusses running GPGPU workloads on RPU in further detail.

Simulation Setup: We analyze our RPU system over multiple stages and simulation tools. Figure 18 shows our end-to-end experimental setup. First, we analyze the SIMT

Table IV: CPU vs RPU Simulated Configuration

Metric	CPU	CPU SMT	RPU
Core Pipeline	8-wide 256-entry OoO	8-wide 256-entry OoO	8-wide 256-entry OoO
ISA	x86-64	x86-64	x86-64
Freq	2.5 GHZ	2.5 GHZ	2.5 GHZ
#Cores	98	80	20
Threads/core	1	SMT-8	SIMT-32 (1 batch)
Total Threads	98	640	640
#Lanes	1	1	8
Max IPC/core	8	8	64 (issue x lanes)
ALU/Bra Exec Lat	1-cycle	1-cycle	4-cycle
#Stages (ALU-load)	9-12	9-12	14-18
L1 Inst/core	64KB	64KB	64KB
Reg File (PRF)/core (scalar+ FP SIMD)	6KB	48KB	192KB
LSU (read/write)	128/64	128/64	128/64 (8x wide)
L1 Cache	64KB, 8-way, 3-cycle, 1-bank 32B/cycle	64KB, 8-way, 3-cycle, 8-bank 256B/cycle	256KB, 8-way, 8-cycle, 8-bank 256B/cycle
L1 TLB	48-entry	64-entry	256-entry, 8-bank (32-entry/bank)
L2 Cache	512KB, 8-way, 12-cycle, 1-bank	512KB, 8-way, 12-cycle, 2-bank	2MB, 8-way, 20-cycle, 2-bank
L3 Cache	32MB, 16-way	32MB, 16-way	32MB, 16-way
DRAM	8x DDR5-3200, 200 GB/sec	10x DDR5-7200, 576 GB/sec	10x DDR5-7200, 576 GB/sec
Interconnect	9x9 Mesh	11x11 Mesh	20x20 Crossbar
OoO entries/thread	256, 8-wide	32, 1-wide	256, 8-wide
L1 capacity/thread	64KB	8KB	8KB
TLB entries/thread	48	8	8
L1B/cycle/thread	32B/cycle	32B/cycle	32B/cycle
memBW/thread	2 GB/sec	0.9 GB/sec	0.9 GB/sec

Table V: Per-component area and peak power estimates

Component	Area				Peak Power			
	CPU		RPU		CPU		RPU	
	mm^2	% Core	mm^2	% Core	Watt	% Core	Watt	% Core
Fetch&Decode	0.27	24.3	0.3	4.3	0.39	15.6	0.4	3.6
Branch Prediction	0.01	0.9	0.01	0.1	0.02	0.8	0.02	0.2
OoO	0.11	9.9	0.17	2.4	0.85	34	1.45	12.9
Register File	0.14	12.6	2.52	35.8	0.49	19.6	4.26	38
Execution Units	0.25	22.5	2.31	32.8	0.34	13.6	2.51	22.4
Load/Store Unit	0.07	6.3	0.34	4.8	0.13	5.2	0.41	3.7
L1 Cache	0.04	3.6	0.22	3.1	0.09	3.6	0.2	1.8
TLB	0.02	1.8	0.08	1.1	0.06	2.4	0.4	3.6
L2 Cache	0.2	18	0.71	10.1	0.13	5.2	0.24	2.1
Majority Voting	0	0	0.02	0.3	0	0	0.03	0.3
SIMT Optimizer	0	0	0.03	0.4	0	0	0.05	0.4
MCU	0	0	0.02	0.3	0	0	0.01	0.1
L1-Xbar	0	0	0.31	4.4	0	0	1.23	11
Total-1core	1.11		7.04		2.5		11.21	
	mm^2	% Chip	mm^2	% Chip	Watt	% Chip	Watt	% Chip
Total-Allcores	108.8	77.2	140.8	81	245	72.5	224.2	73.7
L3 Cache	7.82	5.5	7.82	4.5	0.75	0.2	0.75	0.2
NoC	9.78	6.9	1.72	1	36.52	10.8	7.02	2.3
Memory Ctrl	14.64	10.4	23.59	13.6	6.85	2	19.27	6.3
Static Power					49	14.5	53	17.4
Total Chip	141		173.9		338.1		304.2	

efficiency of our microservice workloads with an in-house x86 PIN [114]-based tool, named SIMTec [115]. The tool traces the dynamic control flow of CPU threads running in a batch, and calculates the associated active mask and overall SIMT efficiency. SIMTec traces the whole SW stack, including user code, libraries, and frameworks. The PIN tool operates in userspace mode so we were not able to trace system calls; however, they only represent 20% of the microservices executed instructions [17], and we expect they should exhibit high SIMT control efficiency.

Second, we use the trace-driven, cycle-level Accel-Sim v1.1 [116] simulator and BookSim [117] for interconnect simulation to obtain chip-level throughput and service latency for the CPU and RPU. We updated Accel-Sim's frontend to execute x86 traces generated by SIMTizer. CISC instructions with memory operands are broken down to multiple RISC-like instructions with separate loads and stores [118]. Further, Accel-Sim's performance model has been extended to model a CPU-like pipeline with superscalar, OoO issue. Table IV lists the simulator configuration for our CPU vs. RPU analysis. We model a many-core x86-based single-threaded CPU similar to the ones found on the market today and commonly used in data centers [30]–[33]. We also model an 8-way simultaneous multi-threading CPU (SMT8), to reflect the highest SMT degree found in the market today from IBM POWER9 [35].

We ensure both CPU and RPU have the same pipeline configuration and frequency. For SMT8, we maintain the same number of total threads and memory resources/thread as the RPU (see the last four entries in Table IV). Cache latency is calculated based on CACTI v7.0 [119]. The multi-bank caches and MCU increase the L1/L2 hit latency from 3/12 cycles in the CPU to 8/20 cycles in the RPU. For other execution units, the ALU/Branch execution latency is increased to 4 cycles in the RPU to account for the extra wiring and capacitance of adding more lanes [43] and the majority voting circuit. We assume an idealistic cache coherence protocol for the CPU with zero traffic overhead, in which atomics are executed as normal memory loads in a private cache. In the RPU, atomic instructions bypass private caches and execute at a shared L3 cache.

Third, to study batching effects on a large scale and capture the system implications of context switching, queuing delay, and network/storage blocking, we harness uqsim [120], an accurate and scalable simulator for interactive microservices. The simulator is configured with our social graph network along with the latency and throughput values obtained from Accel-Sim simulations to calculate system-wide end-to-end tail latency.

Energy&Area Model: We use McPAT [121] and some elements from GPUWattch [80] to configure the CPU and RPU as described in Table IV and to estimate per-component area, peak power and dynamic energy. For the RPU, we consider the additional components and augmentation required to support SIMT execution described in Figure 6. The majority voting circuitry is modeled as a CAM structure (32-way comparator) to count the taken and non-taken results and a reduction tree to calculate the most selected destination address. The SIMT optimizer is modeled as 2x reduction tree to calculate the minimum PC and SP [65] and a CAM structure to calculate the active mask. A 2x 32-way CAM structure is used to model the memory coalescing units [80]. The RAT, ROBs, and uop buffers are extended to include the 4-byte active mask and its associated logic.

Table V shows the calculated area and peak power for the RPU and single-threaded CPU at 7-nm technology [122]. To support SMT-8 in the CPU, a 14% area and power increase per core is required (not shown in the table for brevity). From analyzing the table results, we can make the following observations. First, the CPU's frontend+OoO area and power overhead are roughly 40% and 50% respectively, which are aligned with modern CPU designs [11], [15]. The table shows that the RPU core is 6.3x larger and consumes 4.5x more peak power than the CPU core; however, the RPU core supports 32x more threads. Second, in the RPU core, most of the area is consumed by the register file and execution units, 68% of the area vs. 35% in the CPU. The additional overhead of the RPU-only structures consume 11.8% of the RPU core. Most of this overhead comes from the 8x8 crossbar that connects the L1 banks to the LD/ST queues. Third, the dynamic energy per L1 access and L2 access in RPU is higher by a factor of 1.72x and 1.82x respectively than in CPU, due to the larger cache size, L1-Xbar and MCU. However, the generated traffic reduction and other energy savings in the frontend will outweigh this energy increase as detailed in the next section.

In Section V, we use the per-access energy numbers generated from our McPAT analysis with the simulation results generated by Accel-Sim to compute the runtime energy efficiency of each workload (Figure 19).

V. Experimental Results

A. Chip-Level Results

Figure 19 and Figure 20 show energy efficiency (requests/joule) and service latency of RPU and CPU-SMT8 normalized to single threaded CPU. All the hardware executes the same number of requests (2400). On average, the RPU can achieve 5.7x higher energy efficiency compared to CPU, while still coming within 1.44x of its service latency, with the worst service latency of 1.7x at *HDSearch-midtier*. Overall, the RPU's service latency remains under the 2x higher latency limit defined by data center providers [10], [52]. The main causes of RPU's energy-efficiency are: (1) reducing the number of issued instructions by a factor of 30x, amortizing the frontend and OoO dynamic energy overhead that accounted for up to 73% in the scalar heavily-integer microservices, (2) generating 4x less traffic on average, therefore decreasing the memory energy consumption, and (3) running 6x more requests at almost the same service latency vs. the CPU, and thus amortizing the static energy. The *HDSearch-leaf*, *HDSearch-midtier* and *Search-leaf* microservices exhibit less energy-efficiency than the average. These workloads run at a smaller batch size and/or exhibit high memory divergence (Figure 14). In *HDSearch-leaf*, the frontend+OoO only accounts for 39% of the CPU's energy, limiting the SIMT energy efficiency as discussed in Equation 1 and Section III-A2.

On the other hand, CPU-SMT8 only improves energy efficiency by 5% at a 5x higher service latency cost. This is because the number of issued instructions and the generated accesses are the same as in single threaded CPU. Further, SMT8 partitions the frontend resources per thread and causes cache serialization of stack segment accesses and shared heap variables, hindering service latency, whereas RPU avoids all these issues through SIMT execution.

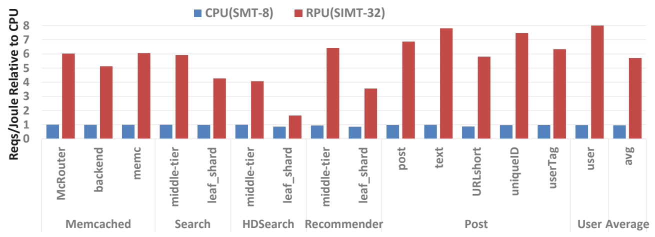


Figure 19: RPU and CPU-SMT8 energy efficiency (requests/joule) relative to single threaded CPU (higher is better)

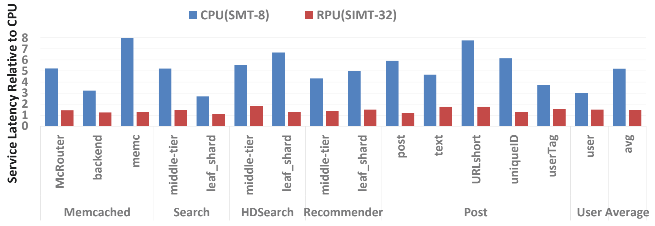


Figure 20: RPU and CPU-SMT8 service latency relative to single threaded CPU (lower is better)

The main causes of RPU's 1.44x higher service latency are four-fold. First, the control SIMT efficiency of some microservices is below 90% (see Figure 11) in which the RPU serializes the divergent paths and increases service latency. Second, when CPU threads run consecutively, they prefetch some shared data to the L1 cache for the incoming threads running on the same core. In the RPU, many threads are run in parallel and incur these compulsory misses at the same time. Third, the L1 access of the RPU is longer (3 vs 8 cycles) as a result of a larger L1 cache size, and ALU instruction latency is 4x higher due to vector pipeline, increasing data dependency latency. Fourth, there remains a slight cache contention occurring in the RPU's L1 cache for *HDSearch-leaf* and *Search-leaf* even after applying batch tuning.

1) Sensitivity Analysis: We evaluate RPU's sensitivity to a number of system parameters:

- *Sub-batch interleaving*: In the CPU, IPC per thread is limited, with a range of 0.3-1, similar to those reported in data center studies [20], [23]. In the RPU, due to sub-batch interleaving, we are able to improve IPC utilization up to 4x by issuing threads over multiple cycles to the SIMT lanes. Although we reduced the number of SIMT lanes by 4x with sub-batch interleaving (i.e., from 32 to 8 lanes), we only noticed 4% performance loss on average, and up to 10% in *UniqueID*, compared to full width SIMT lanes.
- *Moving atomics to L3*: We did not experience slow down from moving atomics to the L3 cache in the RPU, likely because our microservices exhibit few atomic locks per instruction.
- *SIMR-aware heap allocation*: Our SIMR-aware heap segment improves the L1 cache throughput for fre-

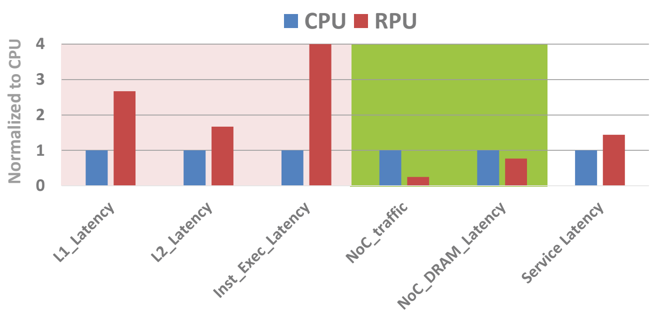
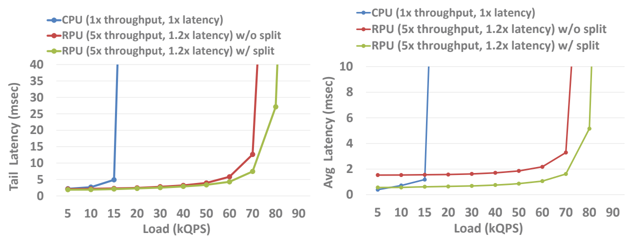


Figure 21: Metrics that contribute to total service latency.



(a) End-to-end 99% tail latency (b) End-to-end average latency

Figure 22: End-to-end tail and average latency for CPU- vs RPU-based system with and without batch split.

quently divergent heap segments in *HDSearch*, where a 1.8x higher throughput was achieved versus the SIMR-agnostic heap allocations.

- *Majority voting*: Majority voting optimizes the branch prediction for common control flow (92% of the time threads traverse the same control flow). Still, 8% control divergence causes some threads to have different predictions per-batch than they would with a per-thread prediction (i.e., as in CPUs). Since we predict next PC per entire batch, we will always have misprediction for the divergent threads of the other path (see Figure 7 example). Majority voting mitigates the flushes caused by these inevitable branch mispredictions by optimizing for the common control flow, and thus improving overall energy efficiency. However, the majority voting policy has little impact on performance, as in the case of divergence both paths are visited anyways, and thus the branch predictor is always correct.

2) Service Latency Analysis: Despite RPU's higher L1 access (2.3x), sub-batch interleaving, and ALU and branch execution latency (4x), some microservices are still able to achieve service latency close to the CPU, and on average only 1.44x higher latency. This is because our workloads are limited by memory latency, with only 20% of the time with successful instruction retirement [23]. In the RPU, the memory coalescing reduces on-chip memory traffic, alleviating contention and minimizing memory latency. Figure 21 depicts several metrics that explain the relatively little increase in service latency for the RPU. The average memory latency has been reduced by 1.33x because 4x less traffic is generated and a single-hop crossbar interconnect is employed which help to offset the latency increases in instructions and cache hits.

3) GPU Performance: We also run our simulation experiments on an Ampere-like GPU model [123] with the same software optimization as the RPU (e.g., stack memory coalescing and batching) and assume that the GPU supports the same CPU ISA and system calls. For the sake of brevity, we did not include the per-app results in the figures. On average, the GPU achieves 28x higher energy efficiency than the CPU but at 79x higher latency, which aligns with previous work [36], [37]. This high latency is unacceptable for QoS-sensitive data center applications [52], [54], [124]. The lower clock frequency and lack of OoO / speculative execution contribute to GPU's higher service latency.

B. System-Level Results

Figure 22 shows the system-level, end-to-end 99% tail and average latency for CPU-based system and RPU-based system with and without our batch splitting technique described in Section III-B5. We scale the QPS load until reaching the highest maximum throughput at acceptable QoS and the system saturates. We configure uqsim with the end-to-end *User* microservice scenario passing from *Web Server* to *User* to *McRouter* to *Memcached* and *Storage* in Figure 3. We simulate three CPU server machines with 40 cores. We assume a 90% hit rate of *Memcached* with 100, 20, 25, 1000 and 60 microseconds latency for *User*, *McRouter*, *Memcached*, *Storage* and network respectively. In the RPU configuration, we replace the CPU servers with RPU machines consuming the same power budget, i.e., assuming 5x higher requests/joule and 1.2x higher latency as were obtained from chip-level experiments for these services. Request batching is employed for *memcached* in the CPU configuration for *epoll* system calls to reduce network processing [120]. To focus our study on processing throughput we assumed unlimited storage bandwidth for both CPU and RPU configurations.

From analyzing the end-to-end results in Figure 22 we can make the following observations. First, the RPU (with batch splitting) can achieve 4x higher throughput compared to the CPU (60 vs 15 kQPS) with almost the same tail and average latency. Second, the batching formation time is amortized and incurs negligible overhead at both low and high traffic load. This is due to the fact that the CPU system employs batching already for memcached. Third, without batch splitting on millisecond-scale storage accesses the RPU exhibits higher average latency than the CPU, as blocked threads are waiting on the reconvergence point for the other threads that access the storage. However, RPU without batch splitting can still attain an acceptable tail latency. Although tail latency is more important than average latency for QoS measurements, the batch splitting technique can be beneficial to ensure predictable response time when unpredictable high latency episodes occur in large online services [124].

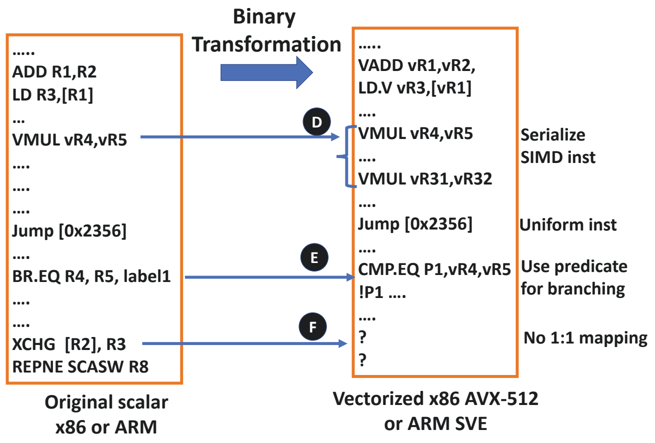


Figure 23: Potential binary transformation of a scalar binary to a vector version

VI. Discussion

This section discusses alternative solutions, other use cases, shortcomings, and pitfalls of the SIMR system.

A. RPU vs CPU's SIMD

A possible alternative to the RPU would be to recompile scalar CPU binaries for execution on the CPU's existing SIMD units, e.g., x86 AVX [125], [126] or ARM SVE [127]. Each request could be mapped to a SIMD lane, amortizing the frontend overhead, leveraging the latency optimizations of the CPU pipeline, and executing uniform instructions on the scalar units [42]. Such a transformation could be done using a SPMD-on-SIMD compiler, like Intel ISPC [42], or at the binary-level, as depicted in Figure 23. However, this solution has three primary shortcomings. First, it requires a complete recompilation of the microservice code, libraries, and OS system calls. Second, SIMD units on contemporary CPUs are designed to accelerate computationally-dense inner loops. The memory system and vector ISA are not optimized for the branch- and memory-heavy microservices we focus on in the RPU. As a result, energy-efficiency and service latency will be negatively affected. For instance, binary transformation requires serializing existing SIMD instructions in the scalar binary (D in Figure 23) and predicate computation cannot take advantage of branch prediction (E). There are 2x more scalar units than SIMD units [30], [31] on existing CPUs, which will go unused if the code can be fully vectorized. Finally, many existing scalar instructions lack a 1:1 mapping with any vector instruction (F), e.g., complex string manipulation, atomic and OS operations. Based on a manual investigation in x86 ISA [125], there are 129 AVX instructions, and 463 scalar instructions, thus only a maximum of 27% of the scalar instructions are represented in the vector ISA.

B. Multi-threaded vs Multi-process Services

Our proposed SIMR system focuses on multi-threaded services, which are widely used in data centers [25], [128]. However, the rise of serverless computing has made multi-process microservices more common [17], [129]. In multi-process services, the separate virtual address spaces can cause both control flow and memory divergence, even if the processes use the same executable and read the same data, which also causes cache-contention issues on contemporary CPUs. We believe that with user-orchestrated inter-process data sharing and some modifications to the RPU's virtual memory these effects can be mitigated. However, since the contemporary services we study are all multi-threaded, we leave such a study as future work.

C. Security Implications

The grouping of concurrent requests for SIMT execution may enable new vulnerabilities. For instance, a malicious user may generate a very long query that could affect the QoS of other short requests or leak control information. Such attacks can be mitigated in our input size-aware batching software by detecting and isolating maliciously long requests, as described in Section III-B1. Another security vulnerability is the potential for parallel threads to access each other's stack data (exploiting the fact that threads' stack data are adjacent in the physical space). However, as described in Section III-B2, the RPU's address generation unit is able to identify inter-thread stack accesses and throw an exception if such sharing is not permitted.

D. GPGPU Workloads on RPU

The RPU can seamlessly execute other HPC, GPGPU, and DL applications that exhibit the SPMD pattern, written in OpenMP, OpenCL, or CUDA. GPUs are 2-5x more energy efficient than CPUs [130]–[133], thanks to their simpler in-order pipeline, lower frequency, and software-managed caches. However, this energy efficiency comes at the cost of easy programmability. Developers need to rewrite their code in a GPGPU programming language and make a heroic effort to get the most out of the GPU's compute efficiency [134], [135]. Recently Nvidia has written its back-end libraries in hand-tuned machine assembly to improve instruction scheduling [116] and has proposed complex asynchronous programming APIs [136] to hide memory latency via prefetching. Such optimizations are likely necessary due to the lack of OoO processing. In CPUs, the hardware supports OoO scheduling with a large instruction window, which removes these performance burdens from the programmers. Furthermore, CPU programming supports system calls naturally and does not require CPU-GPU memory copies.

We believe that the RPU takes the best of both worlds. It can execute GPGPU workloads with the same easy-to-program interface as CPUs while providing energy efficiency comparable to a GPU. CPUs typically contain 1-2x 256-bit (assuming Intel AVX) SIMD engines per core [31], [32] to amortize the frontend overhead. In the RPU, 8x lanes running in lock step, each with a dedicated

256-bit SIMD engine, can provide a wider 2048-bit SIMD unit per core, amortizing the frontend overhead even further and reducing the energy efficiency gap with the GPU. GPUs will likely remain the most energy efficient for GPGPU workloads, but we claim RPUs will not be far behind. We leave an evaluation of the achievable energy efficiency of GPGPU workloads on RPU as future work.

E. Pitfall: The RPU can be bottlenecked by I/O throughput

A high throughput processor like the RPU requires high throughput storage and I/O to perform optimally. In Figure 5, we demonstrate that the off-chip memory bandwidth will dramatically scale in future years with the introduction of DRR5, DDR6, and HBM in the data center. We observe similar trends for I/O standards like PCIe5 and PCIe6 [137]. 128x PCIe6 lanes per single socket can provide 2 TB/sec of bidirectional I/O bandwidth. Ethernet 400/800 Gb/sec and recent NVMe interface advances will enable substantial network and storage throughput improvements.

F. Pitfall: SIMR requires a lot of changes

We design the SIMR system so that the SW stack changes are as minimal as possible. As demonstrated in Section III-B and Figure 12, only the HTTP server and some OS system calls are required to change in the software stack while we keep the programming interface, compiler, runtime, and ISA unaltered. In fact, data center providers adopted DL accelerators [9], [46] and changed the entire software stack for similar outcomes and efficiency.

G. Pitfall: On-premise cost is of paramount importance in TCO

The capital expenses of servers can be a more important factor in the Total Cost of Ownership (TCO) of data centers than power consumption [10]. While we focused our work on energy efficiency, the RPU system also improves cost efficiency. As depicted in Table V, the RPU improves thread density by 5.2$\times$ compared to the CPU. A single socket RPU matches the throughput of six single-threaded CPU sockets while maintaining acceptable latency.

H. Pitfall: Batching interferes with the TCP protocol

Applying batching on the incoming requests before the TCP/IP processing can falsely indicate congestion for the end users, causing an adverse effect on the TCP congestion avoidance algorithm. To mitigate this issue, we can start applying batching at the entry of the logic layer (Figure 3) and bypass the web server (or at least its TCP/IP processing) to send acknowledgments to the end users as soon as possible.

I. Pitfall: Batching is not desired for interactive services

If the QoS demands of a particular service are not being met, we can tune the batch size and set a time-out to decrease latency. In general, if the service cannot tolerate batching of any size or shows low SIMT efficiency, they can be executed on the CPUs at lower throughput.

Table VI: GPU vs RPU Terminology

GPU	RPU
Grid/Thread Block (1/2/3-dim)	SW Batch (1-dim)
Warp	HW Batch
Thread	Thread/Request
Kernel	Service
GPU Core / Streaming MultiProcessor	RPU Core / Streaming MultiRequest
SIMT	SIMR
CUDA Core	Execution Lane

J. RPU vs GPU Terminology

Although RPU and GPU are both SIMT-based hardware, we decided to use different hardware terminology for the RPU. Table VI compares Nvidia's GPU and our terminology.

K. Open Questions

This work has spanned different levels of the system architecture and software stack to execute data center microservice efficiently. While the anticipated results look promising, there remain open questions that require further study:

- How much SIMT control efficiency exists in real-world microservices? Answering this question is critical because while we claim 92% SIMT efficiency in our workloads, this might not be the case for more diverse and complex microservices used by billion-scale users. We hope this work will encourage the data center providers to release more information about the thread similarity and SIMT efficiency of their applications.
- Can we use the existing CPU's SIMD to run the microservices rather than relying on the RPU? What changes are required to make the CPU's SIMD solution feasible? If the compilation and ISA barriers discussed in Section VI-A are resolved, how much performance and energy efficiency can SIMD achieve?
- We left some interesting architecture and software challenges unresolved in this work. For example, improving the SIMT branch prediction to predict the associated active mask along with the next PC is an interesting area to explore. Further, our L1 TLB design suffers from entry duplication, which affects the effective capacity of the DTLBs. Another interesting area to explore is building an efficient and transparent coalescing techniques for the storage and network traffic with hardware and system call support. We left other open questions throughout this paper.

VII. Related Work

Server Workloads on GPUs: The most closely related work to ours are [36], [37], [45]. Agrawal et al. [37] proposed to run data center server workloads, namely SPEC-Web benchmarks, in lock-step execution on GPUs to exploit request similarity. This work achieved significant energy efficiency, but the authors had to rewrite the workloads from PHP to CUDA and reported 6000$\times$ worse latency. Similarly, Hetherington et al. [36], [138] run the memcached workload on a GPU. However the longer request latency,

Table VII: SIMR vs previous SIMT work at a high-level conceptual view

	OoO	CPU ISA	Thread grain	SW & Workloads support
GPUs [123]	✗	✗	Fine	Data-parallel
VT [144]	✗	✗	Fine	Data-parallel
GPU+OoO [145]	✓	✗	Fine	Data-parallel
Simty [140]	✗	✓	Fine	Data-parallel
Vortex [139]	✗	✓	Fine	Data-parallel
DITVA [142]	✗	✓	Fine	Data-parallel
MSPS [45]	✓	✓	N/A	Web server
SIMT-X [61]	✓	✓	Fine	Data-parallel
SIMR	✓	✓	Coarse	Data-parallel & Request-parallel microservices

lack of system calls support, and limited programmability have hindered wide-spread adoption of GPUs for general data center workloads.

RISCV-based GPUs: Recent projects, Vortex [139] and Simty [140], have explored building GPGPUs with a general purpose ISA like RISC-V. While these projects can improve a GPU's applicability to execute CPU binaries, they still suffer from poor latency optimization.

SIMT+OoO: Agrawal et al. [45] study the SIMT efficiency of monolithic SPEC-web workloads [141] written in PHP. Their results indicate the workloads contain promising control and memory efficiency that can be executed on SIMT hardware. They claim GPUs are ill-suited for server applications, so they propose Massively Parallel Server Processors (MSPS) to run server workloads on CPU-like SIMT hardware. However, their work lacks any architectural details and does not tackle the relevant software challenges (Section III-B). Their focus is limited to web serving applications and they require compiler support for reconvergence analysis.

Kalathingal et al. [142], [143] proposed dynamic inter-thread vectorization architecture (DITVA) to leverage the implicit similarity that exists across in-order SMT threads when running data-parallel workloads. Tino et al. [61] introduced SIMT support to an out of order pipeline (SIMT-X) to optimize OpenMP workloads. There are common design concepts between these works and our RPU micro-architecture. Nonetheless, they lack the software stack to support the microservices as they focus on data-parallel workloads. In summery, Table VII compares SIMR versus previous SIMT work at a high-level conceptual view.

GPU+OoO: Previous work [145]–[149] explored adding lightweight out-of-order execution in GPUs to further improve memory latency hiding. Our work is fundamentally different in that we start with an aggressive OoO CPU design, then add GPU-like SIMT elements as necessary to improve energy-efficiency. This approach frees the RPU from the constraints of the GPU programming model, introducing several new challenges we must address to efficiently execute general-purpose pre-compiled microservices. In addition, prior GPU+OoO approaches still relied on massive fine-grained multithreading and focused on throughput, whereas the RPU has significantly fewer threads and balances throughput and latency, addressing unique challenges in the memory system. Furthermore, none of the prior work has made the connection between SIMT and microservices.

Microservices Acceleration: Previous work [18], [19], [150]–[152] have explored using hardware to accelerate microservices, with a focus on remote procedure calls [18], [19], [151], and network data transformations [150]. These proposals are orthogonal to our work and could be applied on top of the RPU. These works focus on helping the CPU remove isolated bottlenecks, whereas the RPU focuses on a full system solution intended to replace the CPU.

Exploiting thread/process locality: Previous studies have proposed exploiting the locality at branch prediction [73], and caches [92], [93], [153] when running different instances of the same thread/process on multi-core CPUs. Still, they incur the same issues of simultaneous multi-threading discussed in Section III-A2.

VIII. Conclusion

Data center computing is experiencing an energy-efficiency crisis. Aggressive OoO cores are necessary to meet tight deadlines but waste energy. However, modern productive software has inadvertently produced a solution hardware can exploit: the microservice. By subdividing monolithic services into small pieces and executing many instances of the same microservice concurrently on the same node, parallel threads execute similar instruction control-flow. We exploit this fact to propose our Single Instruction Multiple Request (SIMR) processing system, comprised of a novel Request Processing Unit (RPU) and an accompanying SIMR-aware software system.

The RPU adds Single Instruction Multiple Thread (SIMT) hardware to a contemporary OoO CPU core, maintaining single threaded latency close to that of the CPU. As long as SIMT efficiency remains high, all the OoO structures are accessed only once for a group of threads, and aggregation in the memory system reduces accesses. Complimenting the RPU, our SIMR-aware software system handles the unique challenges of microservice + SIMT computing by intelligently forming/splitting batches and managing memory allocation. Across 15 microservices our SIMR processing system achieves 5.7x higher requests/joule, while only increasing single thread latency by 1.44x. We believe the combination of OoO and SIMT execution opens a series of new directions in the data center design space, and presents a viable option to scale on-chip thread count in the twilight of Moore's Law.

Acknowledgment

We thank our anonymous reviewers of ISCA 2022 and MICRO 2022 for their feedback and valuable comments. This work was supported, in part, by NSF CCF #1910924 and CCF #1943379 (CAREER).

References

[1] Y. Sverdlik, "Growth of Hyperscale Data Centers," https://www.datacenterknowledge.com/cloud/analysts-there-are-now-more-500-hyperscale-data-centers-world, 2019.

[2] T. Simonite, "Moore's Law Is Dead. Now What?" https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/, 2016.

[3] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, 2019.

[4] C. Trueman, "Why data centres are the new frontier in the fight against climate change," https://www.computerworld.com/article/3431148/why-data-centres-are-the-new-frontier-in-the-fight-against-climate-change.html, 2019.

[5] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "Carbon emissions and large neural network training," *arXiv preprint arXiv:2104.10350*, 2021.

[6] I. King, A. Leung, and D. Pogkas, "The Chip Shortage Keeps Getting Worse. Why Can't We Just Make More?" https://www.bloomberg.com/graphics/2021-chip-production-why-hard-to-make-semiconductors/, 2021.

[7] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, 2020.

[8] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim *et al.*, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[9] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[10] L. A. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines, 3rd edition," *Synthesis lectures on computer architecture*, 2018.

[11] J. Haj-Yihia, A. Yasin, Y. B. Asher, and A. Mendelson, "Fine-grain power breakdown of modern out-of-order cores and its implications on skylake-based systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[12] M. D. Powell, A. Biswas, J. S. Emer, S. S. Mukherjee, B. R. Sheikh, and S. Yardi, "CAMP: A technique to estimate per-structure power at run-time using a few simple parameters," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[13] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient embedded computing," *Computer*, 2008.

[14] J. Sartori, B. Ahrens, and R. Kumar, "Power balanced pipelines," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012.

[15] Z. Xie, X. Xu, M. Walker, J. Knebel, K. Palaniswamy, N. Hebert, J. Hu, H. Yang, Y. Chen, and S. Das, "APOLLO: An Automated Power Modeling Framework for Runtime Power Introspection in High-Volume Commercial Microprocessors," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[16] A. Sriraman and T. F. Wenisch, "μ suite: A Benchmark Suite for Microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[17] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[18] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, N. McKeown, and C. Kim, "The nanoPU: Redesigning the CPU-Network Interface to Minimize RPC Tail Latency," *arXiv preprint arXiv:2010.12114*, 2020.

[19] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[20] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "SoftSKU: Optimizing server architectures for microservice diversity@ scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[21] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[22] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for OLDI microservices," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018.

[23] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a Warehouse-Scale Computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[24] 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 *Proceedings*

of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2012.

[25] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory Hierarchy for Web Search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[26] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[27] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying Memory Access Patterns for Prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[28] D. Gope, D. J. Schlais, and M. H. Lipasti, "Architectural support for server-side PHP processing," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[29] 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," *2012 ACM/IEEE 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[30] K. Lepak, G. Talbot, S. White, N. Beck, S. Naffziger *et al.*, "The next generation AMD enterprise server product architecture," *IEEE hot chips*, 2017.

[31] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman *et al.*, "Cascade Lake: Next generation Intel XEON scalable processor," *IEEE Micro*, 2019.

[32] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala *et al.*, "The ARM Neoverse N1 platform: Building blocks for the next-gen cloud-to-edge infrastructure SoC," *IEEE Micro*, 2020.

[33] "Ampere Computing," https://amperecomputing.com/altra/.

[34] R. Sugumar, M. Shah, and R. Ramirez, "Marvell ThunderX3: Next-Generation Arm-Based Server Processor," *IEEE Micro*, 2021.

[35] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "IBM Power9 processor architecture," *IEEE Micro*, 2017.

[36] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "Memcachedgpu: Scaling-up scale-out key-value stores," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[37] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck, "Rhythm: Harnessing Data Parallel Hardware for Server Workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[38] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a file system with GPUs," in *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[39] M. Silberstein, S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, and E. Witchel, "GPUnet: Networking abstractions for GPU programs," *ACM Transactions on Computer Systems (TOCS)*, 2016.

[40] J. Veselỳ, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt, "Generic system calls for GPUs," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[41] NVIDIA, "NVIDIA GPUDirect," https://developer.nvidia.com/gpudirect.

[42] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in *2012 Innovative Parallel Computing (InPar)*, 2012.

[43] A. Fog *et al.*, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," *Copenhagen University College of Engineering*, vol. 93, p. 110, 2011.

[44] Amazon, "Netflix on AWS," https://aws.amazon.com/solutions/case-studies/netflix/, 2021.

[45] V. Agrawal, M. A. Dinani, Y. Shui, M. Ferdman, and N. Honarmand, "Massively Parallel Server Processors," *IEEE Computer Architecture Letters*, 2019.

[46] M. Anderson, B. Chen, S. Chen, S. Deng, J. Fix, M. Gschwind, A. Kalaiah, C. Kim, J. Lee, J. Liang *et al.*, "First-generation inference accelerator deployment at facebook," *arXiv preprint arXiv:2107.04140*, 2021.

[47] Memchached, "Caching beyond RAM: Riding the Cliff," https://memcached.org/blog/nvm-multidisk/, 2019.

[48] A. Basak, Z. Qu, J. Lin, A. R. Alameldeen, Z. Chishti, Y. Ding, and Y. Xie, "Improving streaming graph processing performance using input knowledge," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[49] D. Meisner and T. F. Wenisch, "DreamWeaver: Architectural Support for Deep Sleep," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[50] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: Eliminating Server Idle Power," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

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

[52] U. Hölzle, "Brawny cores still beat wimpy cores, most of the time," *IEEE MICRO*, 2010.

[53] 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 *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[54] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Communications of the ACM*, 2018.

[55] J. Hruska, "DDR5 specifications," https://www.extremetech.com/computing/312730-ddr5-memory-specification-finalized-up-to-6400gt-s-2tb-lrdimms.

[56] S. Schlachter and B. Drake, "Introducing Micron DDR5 SDRAM: More Than a Generational Update," 2020.

[57] R. Press, "DDR5 vs DDR4 – All the Design Challenges & Advantages," https://www.rambus.com/blogs/get-ready-for-ddr5-dimm-chipsets/, 2021.

[58] T. Morgan, "NVIDIA enters the ARMs race with homegrown Grace CPUS," https://www.nextplatform.com/2021/04/12/nvidia-enters-the-arms-race-with-homegrown-grace-cpus/, 2021.

[59] Z. Peterson, "DDR5 vs. DDR6: Here's What to Expect in RAM Modules," https://resources.altium.com/p/ddr5-vs-ddr6-heres-what-expect-ram-modules, 2021.

[60] I. Cutress, "Intel to Launch Next-Gen Sapphire Rapids Xeon with High Bandwidth Memory," https://www.anandtech.com/show/16795/intel-to-launch-next-gen-sapphire-rapids-xeon-with-high-bandwidth-memory, 2021.

[61] A. Tino, C. Collange, and A. Seznec, "SIMT-X: Extending single-instruction multi-threading to out-of-order cores," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2020.

[62] P. H. Wang, H. Wang, R.-M. Kling, K. Ramakrishnan, and J. P. Shen, "Register renaming and scheduling for dynamic execution of predicated code," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[63] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, "General-Purpose Graphics Processor Architectures," *Synthesis Lectures on Computer Architecture*, 2018.

[64] C. Collange, "Stack-less SIMT reconvergence at low cost," in *HAL, Tech. Rep. hal-00622654*, 2011.

[65] J. D. Collins, D. M. Tullsen, and H. Wang, "Control flow optimization via dynamic reconvergence prediction," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004.

[66] Y. Takahashi, "A mechanism for SIMD execution of SPMD programs," in *Proceedings High Performance Computing on the Information Superhighway. HPC Asia'97*. IEEE, 1997, pp. 529–534.

[67] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[68] NVIDIA. CUDA Binary Utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

[69] ——. PTX ISA :: CUDA Toolkit Documentation. [Online]. Available: http://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[70] A. ElTantawy and T. M. Aamodt, "MIMD synchronization on SIMT architectures," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[71] B. Kerbl, M. Kenzel, M. Winter, and M. Steinberger, "CUDA and Applications to Task-based Programming," in *Eurographics (Tutorials)*, 2022.

[72] NVIDIA, "NVIDIA Tesla V100 GPU architecture," 2017.

[73] S. Hily and A. Seznec, "Branch prediction and simultaneous multithreading," in *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*, 1996.

[74] A. Seznec and A. Fraboulet, "Effective ahead pipelining of instruction block address generation," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, 2003.

[75] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[76] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, 2017.

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

[78] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the killer microsecond," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[79] L. Nyland, J. R. Nickolls, G. Hirota, and T. Mandal, "Systems and methods for coalescing memory accesses of parallel threads," Dec. 27 2011, uS Patent 8,086,806.

[80] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: enabling energy optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[81] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997.

[82] felixcloutier.com, "INVLPG — Invalidate Specific TLB Entries," https://www.felixcloutier.com/x86/invlpg.

[83] P. Bailis and A. Ghodsi, "Eventual consistency today: limitations, extensions, and beyond," *Communications of the ACM*, 2013.

[84] ARM, "Overview of memory consistency," https://developer.arm.com/documentation/ddi0406/c/Appendices/Barrier-Litmus-Tests/Introduction/Overview-of-memory-consistency.

[85] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8," *Proceedings of the ACM on Programming Languages*, 2017.

[86] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding power multiprocessors," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 175–186.

[87] L. Maranget, S. Sarkar, and P. Sewell, "A tutorial introduction to the ARM and POWER relaxed memory models," *Technical Report*, 2012.

[88] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "HMG: Extending cache coherence protocols across modern hierarchical multi-gpu systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[89] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "QuickRelease: A throughput-oriented approach to release consistency on GPUs," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[90] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE journal of solid-state circuits*, 2006.

[91] M. Clark, "A new x86 core architecture for the next generation of computing," in *2016 IEEE Hot Chips 28 Symposium (HCS)*, 2016.

[92] J. Meng, J. W. Sheaffer, and K. Skadron, "Exploiting inter-thread temporal locality for chip multithreading," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.

[93] H. Kwak, B. Lee, A. R. Hurson, S.-H. Yoon, and W.-J. Hahn, "Effects of multithreading on cache performance," *IEEE Transactions on Computers*, 1999.

[94] J. M. Cebrian, M. Jahre, and L. Natvig, "ParVec: vectorizing the PARSEC benchmark suite," *Computing*, vol. 97, no. 11, pp. 1077–1100, 2015.

[95] R. T. Fielding and G. Kaiser, "The Apache HTTP server project," *IEEE Internet Computing*, 1997.

[96] A. Wiggins and J. Langston, "Enhancing the scalability of MemCached," *Intel document*, 2012.

[97] E. D. B. K. S. McKinley and R. D. B. P. R. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications," in *Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), Cambridge, MA, November 2000*, 2000.

[98] I. Gelado and M. Garland, "Throughput-oriented GPU memory allocation," in *Proceedings of the 24th symposium on principles and practice of parallel programming*, 2019.

[99] P. Menage, "TCMalloc : Thread-Caching Malloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[100] S. Damani, D. R. Johnson, M. Stephenson, S. W. Keckler, E. Yan, M. McKeown, and O. Giroux, "Speculative reconvergence for improved SIMT efficiency," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020.

[101] NVIDIA, "CUDA C Programming Guide."

[102] C. Wellons, "Raw Linux Threads via System Calls," https://nullprogram.com/blog/2015/05/15/, 2015.

[103] M. Kerrisk, "mmap — Linux manual page," https://man7.org/linux/man-pages/man2/mmap.2.html.

[104] M. Winter, M. Parger, D. Mlakar, and M. Steinberger, "Are dynamic memory managers on GPUs slow? a survey and benchmarks," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 219–233.

[105] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *Proceedings of the 11th international conference on Supercomputing*, 1997, pp. 76–83.

[106] B. R. Rau, "Pseudo-Randomly Interleaved Memory," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 74–83.

[107] M. Kerrisk, "ld.so — Linux manual page," https://man7.org/linux/man-pages/man8/ld.so.8.html.

[108] StackOverflow, "What is the LDPRELOAD trick?" https://stackoverflow.com/questions/426230/what-is-the-ld-preload-trick.

[109] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[110] M. Rhu and M. Erez, "The Dual-Path Execution Model for Efficient GPU Control Flow," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[111] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[112] A. Basak, J. Lin, R. Lorica, X. Xie, Z. Chishti, A. Alameldeen, and Y. Xie, "SAGA-bench: Software and hardware characterization of streaming graph analytics workloads," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[113] Google, "gRPC: A high performance, open source universal RPC framework," https://grpc.io/.

[114] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "PIN: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[115] A. Alawneh, M. Khairy, and T. G. Rogers, "A SIMT Analyzer for Multi-Threaded CPU Applications," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022.

[116] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[117] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2013, pp. 86–96.

[118] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[119] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," *HP laboratories*, 2009.

[120] Y. Zhang, Y. Gan, and C. Delimitrou, "μqsim: Enabling accurate and scalable simulation for interactive microservices," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[121] 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 *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, 2009.

[122] J. Zhai, C. Bai, B. Zhu, Y. Cai, Q. Zhou, and B. Yu, "McPAT-Calib: A Microarchitecture Power Modeling Framework for Modern CPUs," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[123] Nvidia, "A100 Tensor Core GPU architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[124] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, 2013.

[125] D. Kusswurm, *Modern X86 Assembly Language Programming*. Springer, 2014.

[126] "x86/x64 SIMD Instruction List (SSE to AVX512)," https://www.officedaytime.com/simd512e/.

[127] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu *et al.*, "The ARM scalable vector extension," *IEEE micro*, 2017.

[128] "Which are the most used web servers?" https://www.stackscale.com/blog/top-web-servers/.

[129] "What is the python global interpreter lock (gil)," https://realpython.com/python-gil/.

[130] M. P. Puig, L. De Giusti, and M. Naiouf, "Are GPUs Non-Green Computing Devices?" *Journal of Computer Science and Technology*, 2018.

[131] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 451–460.

[132] Xcelerit, "Computing benchmarks: Processors," https://www.xcelerit.com/computing-benchmarks/processors/.

[133] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," in *2019 IEEE international conference on embedded software and systems (ICESS)*, 2019.

[134] (2017) CUDA C Programming Guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[135] D. Kirk and W. Wen-mei, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[136] G. Thomas-Collignon and V. Mehta, "Optimizing CUDA Applications for NVIDIA A100 GPU," https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf.

[137] J. Hruska, "PCI Express 6.0 With 256GB/s Coming in 2022 Because Screw Bandwidth Constraints," https://www.extremetech.com/computing/293451-pci-express-6-0-with-256gb-s-coming-in-2022, 2019.

[138] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, 2012.

[139] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[140] C. Collange, "Simty: generalized SIMT execution on RISC-V," in *CARRV 2017-1st Workshop on Computer Architecture Research with RISC-V*, vol. 6, 2017, p. 6.

[141] SPEC Organization, "SPECweb 2009," https://www.spec.org/web2009/, 2009.

[142] S. Kalathingal, S. Collange, B. N. Swamy, and A. Seznec, "DITVA: Dynamic inter-thread vectorization architecture," *Journal of Parallel and Distributed Computing*, 2018.

[143] S. Kalathingal, C. Collange, B. N. Swamy, and A. Seznec, "Dynamic inter-thread vectorization architecture: extracting DLP from TLP," in *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2016, pp. 18–25.

[144] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The Vector-Thread Architecture," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, 2004.

[145] K. Iliakis, S. Xydis, and D. Soudris, "LOOG: Improving GPU Efficiency With Light-Weight Out-Of-Order Execution," *IEEE Computer Architecture Letters*, 2019.

[146] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[147] X. Gong, X. Gong, L. Yu, and D. Kaeli, "HAWS: Accelerating GPU wavefront execution through selective out-of-order execution," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2019.

[148] P. Xiang, Y. Yang, M. Mantor, N. Rubin, and H. Zhou, "Revisiting ILP designs for throughput-oriented GPGPU architecture," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.

[149] R. Espasa, M. Valero, and J. E. Smith, "Out-of-Order Vector Architectures," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997.

[150] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus Prime: Accelerating Data Transformation in Servers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[151] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC tax in datacenters," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[152] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3:{Energy-Efficient} microservices on {SmartNIC-Accelerated} servers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[153] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong, "Multi-execution: multicore caching for data-similar executions," in *2009 ACM/IEEE 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.