Massively Parallel Server Processors

Varun Agrawal, Mina Abbasi Dinani, Yuxuan Shui, Michael Ferdman, and Nima Honarmand

Abstract—Modern data centers enjoy massive degrees of request-level parallelism with significant cross-request similarity. Although similar requests follow similar instruction sequences, conventional processors service them individually and do not take full advantage of cross-request similarity. Single-Instruction Multiple-Thread (SIMT) architectures can leverage this similarity, however, existing SIMT processors—chief among them, GPUs—are ill-suited for server applications, as they are specifically designed to maximize throughput at the expense of latency, preventing them from meeting server QoS requirements. We advocate a new approach to SIMT server processors, namely Massively Parallel Server Processors (MPSPs), which we outline in this paper. To begin to understand their architectural needs, we measure the degree of control-flow and memory-access divergence encountered when running *unmodified* server applications on MPSP-style processors. Our preliminary results indicate that a software scheduler that bundles together similar requests can minimize control-flow divergence, making SIMT execution of unmodified server code feasible. Moreover, we find that memory-access divergence, although significant in raw numbers, can be tackled with changes in stack and heap layouts. Overall, our results encourage further consideration of MPSPs as a promising architecture for server processors.

Index Terms—Parallel processing, Single Instruction Multiple Thread, servers, data centers

1 Introduction

Modern data centers enjoy massive degrees of request-level parallelism. High-throughput servers routinely receive thousands of requests per second. These independent requests often result in repeated tasks, such as web servers serving the same pages or databases processing the same queries, with identical or similar parameters. Such repetition results in the execution of many nearly-identical instruction sequences [1], [2].

At the same time, reducing energy consumption is a major goal for warehouse-scale data centers, as electricity and cooling comprise much of their total cost of ownership [3]. A significant portion of this energy is used by the conventional super-scalar out-of-order processors that constitute, almost exclusively, the processing engines of modern data centers servers [4].

Conventional processors execute independent requests separately and cannot take advantage of the similarity of their instruction streams. This is unfortunate because the majority of the energy spent by modern processors is not in the functional units, but in the logic responsible for fetching, decoding, and scheduling instructions and memory accesses [5]. Cross-request instruction-stream similarity can be leveraged to amortize this overhead over a large number of similar independent requests to significantly reduce processor energy.

A promising architectural style to leverage cross-request similarity is *Single-Instruction Multiple-Threads* (SIMT). In particular, a server can bundle together groups of similar requests and execute them simultaneously and in lockstep on a SIMT processor. As long as these threads of execution remain in lockstep, each instruction needs to be fetched, decoded, renamed, scheduled, and issued only once for the entire group of threads. This allows much of the complex logic in the processor pipeline to be shared by all the threads in a group.

The potential of SIMT servers has led researchers to investigate using GPUs. Most notably, Rhythm [6] re-implemented server software from scratch, including web server and network stack, targeting the GPU programming model. Although its results demonstrated that server requests can be processed in a SIMT fashion, it also exposed the fundamental problem of GPUs in this domain: GPUs cannot meet the stringent Quality of Service (QoS) requirements of server applications.

This finding is not surprising, as GPUs are fundamentally ill-suited for server workloads. First, GPUs use throughput-oriented architectures that rely on *fine-grained context switching* among *massive numbers of threads* to hide instruction latencies, greatly sacrificing single-thread performance. High single-thread performance is required to meet QoS targets. Therefore, effective SIMT server processors must employ latency-hiding techniques—such as branch prediction, out-of-order and speculative execution, and advanced cache organizations—which are absent from GPUs. Moreover, server workloads have far fewer threads compared to data-parallel GPU workloads, ruling out fine-grained context switching for hiding latency.

Second, GPU memory systems provide high bandwidth only when accesses from different threads can be coalesced into large blocks. However, server workloads are notorious for random pointer-chasing memory access patterns [1] and are therefore not a good match for a GPU memory system. Instead, they require large instruction caches and support for many parallel small requests from different threads.

Third, GPUs require compile-time static analysis of code written for a GPU-specific programming model. Running server workloads on a GPU requires a complete re-implementation of the software in environments like CUDA or OpenCL [6], which would be a major barrier to adoption. Server software comprises millions of lines of complex code, numerous third-party libraries, and frequent interaction with the OS, rendering compilation and execution of traditional server software impossible on GPUs.

Massively Parallel Server Processors. Leveraging SIMT in the server domain requires a de novo approach. We advocate a new class of SIMT CPUs called Massively Parallel Server Processor (MPSP). The goal of MPSP is to run server software *with minimal source modifications* while balancing SIMT efficiency with *stringent server QoS requirements*. Fig. 1 depicts our vision for MPSP, coupling several conventional cores with many SIMT cores optimized for server workloads, with all cores sharing the LLC and memory interface.

The SIMT cores are responsible for processing server requests. Each core consists of a *shared frontend* and *replicated execution lanes* which execute groups of threads in lockstep.

The frontend fetches, decodes, renames, schedules, and issues instructions from a single instruction stream shared by all the threads in the group, amortizing the cost of these complex operations among the threads. To improve single-thread performance and meet QoS requirements, MPSP can leverage conventional latency-hiding techniques such as instruction prefetchers, branch prediction, and out-of-order issue.

The SIMT execution lanes resemble conventional pipelines, replicated for each thread in the group and executing the fetched instructions in parallel for all thread contexts. The lanes can be simple—without complex forwarding paths or wide super-scalar

• *The authors are with the Stony Brook University, Stony Brook, NY 11794.*
E-mail: {vagrawal, mabbasidinan, syuxuan, mferdman, nhonarmand}@cs.stonybrook.edu.

Manuscript received 19 Dec. 2018; revised 23 Feb. 2019; accepted 21 Mar. 2019. Date of publication 14 Apr. 2019; date of current version 12 June 2019.
(Corresponding author: Mina Abbasi Dinani.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2019.2911287

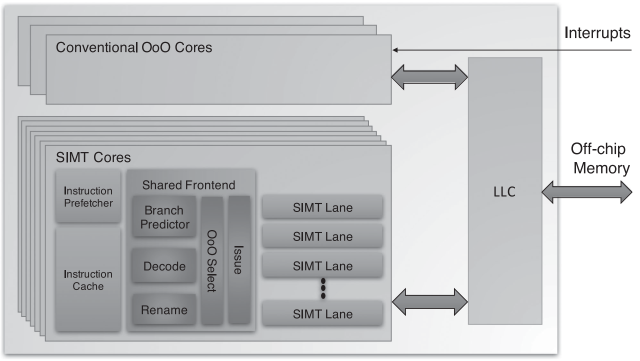

Fig. 1. MPSP architecture, a combination of conventional and SIMT cores that share a last-level cache and memory interface.

pipelines—while achieving high throughput due to large thread groups (32-128 threads in our current thinking). We envision the combined benefits of shared frontend and simplified backend to dramatically reduce the energy-per-request of MPSP compared to existing server processors.

In MPSP, the conventional cores are responsible for running the request thread scheduler and handling I/O, system calls, and interrupts. The scheduler is responsible for examining the incoming requests, grouping them based on their similarity, and dispatching the thread groups to the SIMT cores. In some cases, to form large groups, the scheduler may introduce slight delays before dispatching requests. Introducing scheduling delays will have minimal impact on the response latency, as they will be small compared to the overall request processing time.

The MPSP memory system is distinct from both conventional processors and GPUs. Unlike conventional processors, which see limited memory-level parallelism (MLP) due to frequent pointer chasing in server workloads, MPSP will have high bandwidth demands due to concurrent accesses from a large number of SIMT threads. However, unlike GPUs, which cater to a large number of concurrent accesses by coalescing adjacent requests, MPSP must support and track a large number of distinct in-flight memory requests. Moreover, the MPSP memory system must carefully cater to instruction fetch. Server workloads have large instruction working sets that cannot fit into L1-I caches [7], in contrast to GPUs which run small kernels with tiny instruction working sets. The MPSP memory pipeline must be optimized for instruction-fetch access patterns and will likely include an L2-I cache for shared instructions [8].

To prove the feasibility of the MPSP approach, and to inform the details of its architectural design, the most fundamental question one needs to address is the degree of control-flow and memory-access divergence that one can expect when running *existing* server programs on an MPSP. An MPSP is only effective if threads in a thread group remain in lockstep for long periods (i.e., show little control-flow divergence) and if there is enough parallelism and bandwidth in the memory system to accommodate the memory accesses of the running threads.

Measuring control-flow and memory-access divergence in server workloads is the subject of the rest of this paper. We conduct a study using a functionally-simulated MPSP system, running an Apache PHP server on Linux. Our results demonstrate that more than 99% of the user space instructions of Apache and PHP can run in lockstep. Moreover, despite abundant memory-access divergence, we find that concurrent accesses across SIMT threads follow rigid patterns that enable coordinated issue and tracking of the large number of concurrent accesses required for MPSP. Overall, these results underline the promise of MPSP as a breakthrough paradigm in server architecture design and encourage further research in this area.

2 Methodology and Experimental Setup

To explore the control-flow divergence and memory system behavior of server workloads in a SIMT environment, we built a full-system functional MPSP simulator based on Wind River Simics [9]. A key feature of our simulator is that application, workload, and compiler functionality need not be modified. We dynamically schedule software threads for SIMT execution, requiring only minor code annotations in the *lock* and *unlock* functions to schedule threads during critical sections. To assist in analyzing the memory accesses, we also add code hints to the memory allocator, which do not alter application behavior.

Our simulator runs multi-threaded server applications in SIMT fashion. To model SIMT behavior, our simulator executes instructions from the SIMT threads in a round-robin fashion. All SIMT threads execute a common instruction simultaneously. One instruction on each MPSP thread completes before any of the threads proceed to the next instruction. For this study, all instructions execute in one simulated cycle and there is no timing model of the processor pipeline or the memory system.

We use the prototypical server application, Apache web server under Linux, running the SPECweb2009 "Support Suite" reference PHP implementation with its default opcode cache accelerator. The workload simulates a vendor support website that serves dynamic pages, such as pages that list, search, and extract product details. Each server access requests one of six different dynamic PHP pages. The system is instrumented to bundle together requests to similar pages, simulating the behavior of the MPSP software thread scheduler.

We configured Simics with $n+1$ simulated cores. One core is simulated unconstrained to mimic a conventional core. We developed a Simics module to coerce the remaining cores to run in lockstep, simulating an MPSP SIMT core with n hardware threads. We sequester these n cores for SIMT execution using the Linux *isolcpu* feature, allowing only explicitly-pinned threads to run on them. We configure the Apache web server with n worker threads and pin each thread to one of the sequestered cores, simulating the MPSP SIMT core. To mimic the behavior of all interrupts and system calls executing on the conventional core, we freeze the MPSP SIMT lockstep execution when kernel code is invoked, resuming SIMT lockstep execution when control-flow returns from the kernel.

Our configuration achieves functional correctness and is sufficient to demonstrate the advantages of the MPSP hardware, so long as the OS code running on the conventional cores does not become a bottleneck. Although a noticeable fraction of the instructions in our system is currently in the OS (8.4 percent), the overwhelming majority of these instructions belong to the TCP/IP network stack, code that we plan to migrate to the SIMT cores using user-space networking [10]. The remaining code (thread scheduling, interrupts, and OS background management tasks) is not amenable to SIMT execution, but constitutes a negligible fraction of the instructions.

A major challenge in a SIMT system is handling control-flow divergence in a SIMT thread group. The shared front-end of SIMT architectures implies that all threads concurrently execute the same instruction. Our Simics module mimics this behavior by executing one instruction per cycle for all threads in the MPSP SIMT core *only if all of the cores are pointing to the same instruction in memory*. However, real software threads can follow different paths after encountering data-dependent conditional branches. As a result, the thread group splits into smaller subgroups that no longer execute in lockstep; instead, the subgroups must be executed either serially or in an interleaved fashion on the SIMT processor, significantly degrading processor resource utilization. For SIMT to be beneficial, it is crucial for the subgroups to *re-converge* at the earliest possible rendezvous point. One common approach is to identify the re-convergence point at compile time and pass this information

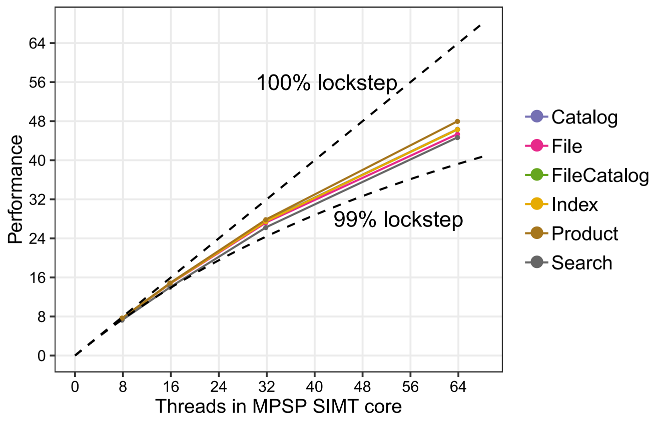

Fig. 2. Control-flow divergence for different request types.

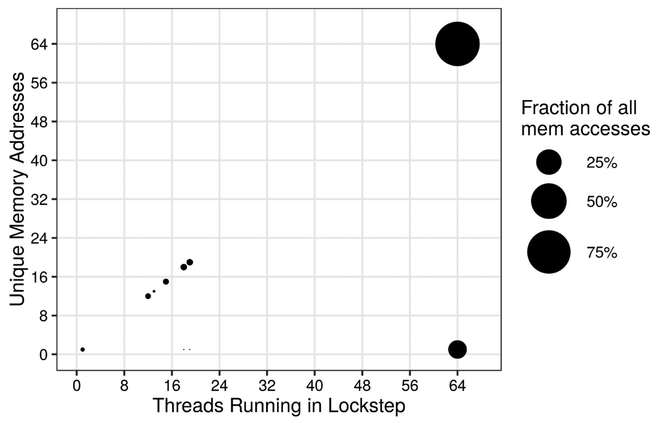

Fig. 3. Memory divergence of 64 threads for *Search* requests.

to the hardware as part of the binary. The compiler analyzes the static Control-Flow Graph of the program to identify the post-dominator [11] of each divergence point, and uses this post-dominator as the re-convergence point. The SIMT hardware uses a *branch synchronization stack* to keep track of the divergences and their corresponding re-convergence points [12].

Compile-time static analysis can identify most of the re-convergence points in the application source code, but would require modifying the x86 toolchain, a costly undertaking for the purposes of this study. Instead, we developed a *dynamic technique* to identify the appropriate re-convergence points at runtime. Our simulator dynamically constructs the CFG by monitoring the executed instructions, identifying the appropriate rendezvous points for each branch instruction through post-dominator analysis on the generated CFG. We find that, although the threads diverge frequently, the number of variant CFG paths is small in practice. Our analysis indicates that all common paths are typically observed after executing a small number of requests, after which point the CFG is stable and can be used for post-dominator analysis. When threads diverge, our simulator identifies the rendezvous point based on the analysis results, and uses a branch synchronization stack to re-converge the threads after they all reach the post-dominator instruction.

3 Control-Flow Divergence

We used our MPSP simulator to study the control-flow divergence of concurrent requests. Upon divergence, the threads are partitioned into multiple subgroups between the divergence and re-convergence points (when all threads cannot run in lockstep). Threads in each subgroup follow the same path and continue to execute in lockstep. While a subgroup executes, the other subgroup(s) are stalled, waiting either at the divergence or re-convergence point. Serially running the subgroups reduces the benefits of MPSP, amortizing instruction overheads across fewer threads, and decreases the potential efficiency and performance. For a SIMT processor to be beneficial for servers, control-flow divergence must be limited to allow maximum use of SIMT threads in the vast majority of the time.

To establish the effectiveness of MPSP, we estimate its performance for different numbers of hardware threads in the MPSP SIMT core by calculating the fraction of cycles that the hardware threads are occupied. We study the instructions that run in lockstep compared to the serially-executing diverged threads. Importantly, all divergences are not equal. In some cases, there are two divergent paths, forming two groups and resulting in an average of $\frac{1}{2}$ MPSP utilization. However, in the worst case, each thread may follow its own unique path, resulting in completely separate serial execution of all threads and $\frac{1}{n}$ utilization. We therefore estimate the impact of divergence on performance by measuring the average number of instructions executing in lockstep:

$$\text{Performance} = \frac{\text{Total number of instructions for all requests}}{\text{Simulated cycles to process all requests}}.$$

Fig. 2 shows MPSP performance for different PHP requests when running 8, 16, 32, or 64 parallel threads. For contrast, dashed lines show 100 percent instructions running in lockstep and 99 percent running in lockstep. As expected, higher thread counts show greater deviation from the ideal behavior. However, even with 64 threads, more than 99 percent of the instructions execute in lockstep across all request types, indicating that MPSP can offer significant performance and efficiency benefits.

To understand the cause of the opportunity loss for MPSP at higher thread counts, we identified the serially-executed functions. Most of the serialized execution is not due to thread divergence, but arises from software locks in the PHP memory allocator. Moreover, although the current PHP code forces serialization for the entire allocator, the instructions executed inside the locked region are actually the same across all threads. While we avoided functional changes to the software for this study, we find that re-engineering the memory allocator to use finer-granularity locks would increase the fraction of instructions that can execute in lockstep to nearly 100 percent even at 64 threads.

4 Memory Access Divergence

In addition to the limited control-flow divergence, SIMT execution requires coordinated memory accesses. Frequent memory access divergence—a large number of uncoordinated and independent accesses—would necessitate a complex memory subsystem, cache, and interconnect to prevent massive performance degradation due to the serialization of memory accesses.

Fig. 3 presents the observed memory access divergence as the number of unique addresses accessed simultaneously. The location of the bubbles in the figure shows the number of unique addresses accessed simultaneously by the corresponding number of threads. The size of each bubble indicates the relative fraction of all memory accesses. Memory accesses across all request types are dominated by the utility functions that interpret PHP scripts and templates (*zendparse*, *lex_scan*, and *zendlex*), therefore, the divergence behavior is similar for all request types. Here, we present results for the *Search* requests.

When running in lockstep, a significant fraction of the memory accesses are to the same address, indicated by the bubble at $(64, 1)$. However, the vast majority of the accesses are to different addresses, indicated by the bubble at $(64, 64)$. Similarly, small

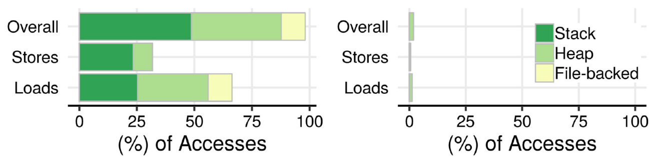

Fig. 4. Access breakdown of 64 threads for $Search$ requests.

bubbles along the diagonal also represent accesses to different addresses, but, in these cases, control-flow divergence occurred and fewer than 64 threads were running in lockstep. Taken in isolation, these results would signal significant challenges in the design of an MPSP memory system.

To better understand the memory access behavior in server workloads, we concentrated on the relationship between simultaneous accesses. We used the Linux */proc/pid/maps* facility to identify the virtual memory region to which each access belongs: file-backed (shared libraries and read-only data), heap, or stack. We found that practically all accesses to the file-backed regions are to the same address across all threads, while the stack and heap accesses are to different, divergent, addresses.

We examined the heap and stack accesses further, treating them as offsets from the allocation base address instead of as absolute addresses. For the heap, we instrument the *malloc()* function to pass the base address of each memory allocation to the simulator and use the offset within the allocated object when comparing the addresses across the simultaneously-executing memory access instructions. For the stack, we calculate the offset from the base of the stack of the respective thread.

Fig. 4 breaks down the simultaneous memory accesses with the same and different offsets. We further separate accesses into loads and stores to highlight their different distributions across the memory regions. To simplify the plots, file-backed accesses are shown as same-offset, although, in reality, they are same-address accesses (a special case of same-offset).

Although the vast majority of simultaneous accesses are to distinct addresses, practically all divergent memory accesses performed by threads running in lockstep are to the same offset within their allocation. The stack region accesses have identical offsets across all threads, for both loads and stores. Only a tiny fraction of divergent accesses, all in the heap, are to different offsets. These differences arise from reading and manipulating character strings of different lengths, owing to the differences in the parameters of the web requests being serviced.

Although the stack and heap accesses initially appeared divergent, their offsets indicate that, across threads, they follow rigid patterns and the majority of the accesses can be coordinated by modifying the memory layout (e.g., interleaving the thread stacks). Further indicating opportunity for software-assisted coordination of accesses, we find that heap objects that are accessed in lockstep are often allocated in lockstep and are almost always accessed simultaneously. Vector memory allocation can be used to allocate heap space for a group of such objects to avoid unnecessary allocation-time synchronization, and to help coordinate future memory accesses to the objects.

5 Conclusions

Servers process similar requests, executing nearly identical instruction sequences. However, existing architectures execute independent requests separately and do not take advantage of the cross-request instruction sequence similarity. To leverage cross-request similarity, we described a SIMT-based Massively Parallel Server Processor (MPSP) architecture to significantly improve server system efficiency. Using a functional simulation of an MPSP system running an Apache PHP server on Linux, we found that more than 99% of the user space instructions can run in lockstep on such an architecture. Furthermore, we find that although simultaneous memory accesses are often to different addresses, they have the same offset from a thread-specific address, suggesting that accesses can be made “regular” by modifying the memory layout. Together, these results demonstrate a significant opportunity for the MPSP architecture that warrants further research toward its development.

Acknowledgments

Varun Agrawal and Mina Abbasi Dinani contributed equally to this publication.

References

[1] M. Ferdman, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Temporal instruction fetch streaming,” in *Proc. 41st Int. Symp. Microarchitecture*, 2008, pp. 1–10.
[2] M. Ferdman, C. Kaynak, and B. Falsafi, “Proactive instruction fetch,” in *Proc. 44th Int. Symp. Microarchitecture*, 2011, pp. 152–162.
[3] L. Barroso and U. Hoelzle, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. San Rafael, CA, USA: Morgan and Claypool Publishers, 2009.
[4] J. Hamilton, “Overall data center costs,” Sep. 2010: [Online]. Available: https://perspectives.mvdirona.com/2010/09/overall-data-center-costs
[5] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” in *Proc. 37th Int. Symp. Comput. Architecture*, 2010, pp. 37–47.
[6] S. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. Lebeck, “Rhythm: Harnessing data parallel hardware for server workloads,” in *Proc. 19th Int. Architectural Support Program. Languages and Operating Syst.*, 2014, pp. 19–34. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541956
[7] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the clouds: A study of emerging scale-out workloads on modern hardware,” in *Proc. 17th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2012, pp. 37–48.
[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, “Reactive NUCA: Near-optimal block placement and replication in distributed caches,” in *Proc. 36th Int. Symp. Comput. Archit.*, 2009, pp. 184–195.
[9] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Comput.*, vol. 35, no. 2, pp. 50–58, 2002.
[10] DPDK Project, “Data plane development kit,” 2018, https://www.dpdk.org/
[11] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Languages Syst.*, vol. 13, no. 4, pp. 451–490, 1991.
[12] J. Hennessy and D. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.