Research Statement: Improving the Performance and Correctness of Parallel Programs

Joseph Devietti, University of Pennsylvania

The last two decades have overseen a historic shift in processor design toward heterogeneous designs in which multicore CPUs are augmented with special-purpose graphics hardware (GPUs) and a growing array of fixed-function accelerators for uses such as media codecs and machine learning. The rise of heterogeneous parallel hardware has made the challenge of parallel programming even harder, introducing new kinds of performance and correctness bugs, and new programming models for which debugging tools are missing or lacking.

My research seeks ways to make parallel programming easier, safer and faster via innovations across the system stack, from computer architecture to runtime systems, programming languages and algorithms. I build hardware and software systems that can find and fix performance and correctness bugs in parallel software. As parallelism is a cross-cutting concern across many areas of computer science, my work correspondingly spans from architecture to algorithms and has been published in venues ranging from MICRO to SODA.

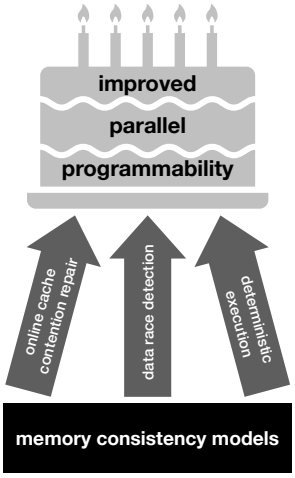

Figure 1: Research overview

A unifying theme throughout my work has been a deep appreciation for memory consistency models. I often work at the limits of these models and expand their boundaries: for example, my work is the first to show the benefits of multi-lingual memory consistency models [1]. More broadly, I have demonstrated the first systems for:

- correct and efficient online cache contention repair
- comprehensive data race detection on GPUs
- new abstractions in high-performance deterministic execution

My work has served as a building block for research at the University of Michigan [2]. I have received funding from Intel, Google and Nvidia, and also an Intel Early Career Faculty Award. At Intel, my work has driven internal conversations about next-generation hardware extensions for performance and security. To transition my work on deterministic containers into commercial reality, I co-founded the startup Cloudseal to bring fundamental improvements to crash reporting technology.

During my time at Penn I've **graduated three PhD students**. Christian DeLozier, my first student, is now an Assistant Professor of Electrical and Computer Engineering at the US Naval Academy. My student Nimit Singhania, co-advised with Rajeev Alur, joined a program analysis team at Google, and my student Yuanfeng Peng joined a network debugging team also at Google.

I've worked to improve the rigor of **computer architecture teaching** at Penn. I introduced hardware design into the graduate computer architecture class for the first time, and have developed a series of interactive computer architecture visualizations to explain core concepts and support self-directed exploration.

My **community service** to the academic community has included organizing workshops at ASPLOS and EuroSys, and serving as Treasurer/Registration Chair for HPCA 2015. I've been on the PC of many first-tier conferences, including ASPLOS, HPCA, PLDI and IEEE MICRO Top Picks.

Below, I outline the progress I've achieved along the three main directions of my research vision, and describe my plans for the future.

Direction 1: Repairing Cache Contention Online

One important class of multicore performance issues is *cache contention bugs*: they arise when true sharing or false sharing repeatedly triggers slow paths in a multicore processor's cache coherence protocol. These bugs have been found in production code like the Boost C++ libraries, MySQL and Linux. Cache contention bugs are hard to identify as they arise from opaque memory layout decisions made by compilers and memory allocators, and their severity is affected by the details of each system like interconnect topology. To address these inherent challenges, we've shown that smart runtime systems can effectively solve many cases of cache contention without programmer intervention, and without even stopping program execution.

Our first system is Laser: Light, Accurate Sharing dEtection and Repair [3]. Laser detects cache contention via a family of performance counters, available in recent Intel architectures, that provide the program counter and memory address of instructions that trigger certain coherence protocol transitions. Laser's repair mechanism implements a software-based store buffer using dynamic binary rewriting. Like a hardware-based store buffer, our software store buffer defers the cost of cache coherence (and thus, contention) to improve performance. Laser-Repair is the first online contention repair scheme to adhere to the TSO consistency model, avoiding semantic pitfalls in previous approaches.

While sound, Laser's repair mechanism is slow so we developed the Thread Memory Isolation (TMI) system [1] that implements software-based store buffering using virtual memory techniques instead. This results in radically lower performance overheads – compared to ideal manual repair of false sharing, LASER provides just 24% of that manual speedup while TMI provides 88%. Once contention is detected, TMI employs a novel multithreaded *fork* primitive[1] that upgrades each thread to live in its own process. Now, the same virtual page can map to different physical pages for different threads, resulting in an efficient store buffer mechanism controlled by software. While our software store buffers have very weak consistency semantics, we show their use is still compatible with the C/C++ memory model. Along the way we discovered that while existing memory models assume *monolingual* programs, many programs (often due to super-optimized libraries like libc) are a *mix* of languages like C and assembly. TMI accounts for the vastly differing semantics of x86 assembly and C/C++, to ensure that TMI is correct even for complex *multilingual* programs. TMI [1] and Laser [3] are featured as building blocks in follow-on work from the University of Michigan [2].

We next explored the potential for automatic cache contention repair in managed runtime systems with Remix [4], the first system to detect or repair cache contention in a language virtual machine. Remix is a modified version of the Oracle HotSpot JVM that can automatically discover all forms of cache contention bugs and repair false sharing bugs in programs running on the JVM. REMIX repairs false sharing issues as efficiently as manual fixes, and sometimes even better: one expert-tuned benchmark ran faster with Remix than without because the expert had coded defensively against false sharing that did not materialize on our hardware platform. Remix avoided inserting any padding, and achieved better cache locality as a result.

Current and future work

We are currently exploring ways to mitigate the performance impact of limited instruction cache (I$) capacity, by adapting code layout at runtime to the paths a program is currently using. Changing code at runtime raises consistency model issues, though with *instruction memory* which is largely ignored in the literature. Establishing the semantics of instruction fetches and writes will expand our understanding of memory consistency and enable future research in runtime code transformations.

[1] Regular *fork*, in a multithreaded process, creates a new process with only the calling thread and no others.

More broadly, there are many untapped opportunities in "self-driving" optimization based on performance counters. Performance counter feedback can be used to tune application-level parameters, like the size of internal caches, beyond adjusting behavior solely at the ISA level. Heavyweight optimizations, like memory prefetchers powered by deep learning, can be trained with the latest runtime data; concerns about overfitting are also minimized in our online setting where only the current execution is of interest. There are also likely opportunities to reduce training overheads by transferring optimizations between machines or even applications. I believe automatic runtime optimizations like these are key to scaling performance in future architectures.

Direction 2: Concurrency Bug Detection

Multithreaded programming suffers from a host of well-known correctness challenges. One of the key challenges is the potential for programs to contain *data races*. Data races can introduce non-sequentially-consistent and even undefined behavior into programs, making programs hard or impossible to understand. The possibility of data races also complicates other analyses we often want to perform on multithreaded programs, such as recording and replaying multithreaded execution, verifying program correctness, and enforcing determinism.

Our work on concurrency bug detection has taken many forms. On the theoretical side, we've worked on new algorithms for richer parallel programming models [5], and our work mechanizing proofs of correctness of existing algorithms [6] has helped identify small issues in the paper proof in a famous piece of related work [7]. We've shown how to improve on the state-of-the-art FastTrack race detection algorithm by deduplicating the massive amounts of redundancy in race detection metadata [8]. We've also explored a new hybrid hardware-software solution for data race detection to reduce its performance overheads [9].

With the rise of GPU architectures and the massive amounts of parallelism that these accelerators support, our attention turned to providing efficient race detection for GPU programs. CPU race-detection algorithms do not fare well on the GPU, as they have many components that consume time or space linear in the number of threads. This is fine for multithreaded applications that sport tens of threads, but is a huge stumbling block with GPU programs that may literally spawn millions of concurrent threads. Our Barracuda system [10] proposes a new, scalable race detection algorithm that leverages the hierarchical structure of the GPU programming model to keep time and space overheads in check. Barracuda is also the first race detector to support non-trivial parts of the GPU memory consistency model like fences and atomics.

In follow-on work, we designed the CURD system [11] that performs race detection on the GPU itself (Barracuda streamed events to the CPU to perform race detection there), providing a massive performance boost. CURD is 17x faster than Barracuda and 2x faster than Nvidia's own CUDA-Racecheck race detector, despite the fact that CURD detects many more classes of races than CUDA-Racecheck does. Having received over a dozen requests for access to the CURD source code, we are working with Nvidia to open-source it and upgrade the current state of CUDA race detection.

Synergistic work

In a related line of work we've built a series of static analyses for GPU programs to identify potential performance issues. Our GPUDrano system [12] detects uncoalesced memory accesses which can artificially inflate a program's memory bandwidth demands by an order of magnitude or more. We also developed the first analysis for *block size independence* of GPU programs [13] (best paper award winner at SAS '18), a new correctness property that guarantees that changes to the *block size* parameter of a GPU program will not affect the program's semantics. Tuning the block size is a common performance optimization, as the optimal block size varies across GPUs. However, without block size independence, changing the block size can silently alter the computation being performed. Our analysis puts the common practice of block size tuning on a safe semantic footing for the first time.

Direction 3: Improving Deterministic Execution

Another core source of complexity in multithreaded programming is the presence of nondeterminism. Research into deterministic execution seeks to alleviate many of the correctness issues that plague multithreaded programming, by making program behavior repeatable. Determinism makes it much simpler to debug a program, makes it possible to replicate programs for reliability, and facilitates archiving of computational artifacts. Through both research and commercialization efforts, we've expanded the boundaries of determinism to encompass new and richer classes of programs.

Deterministic Shared Memory

One key bottleneck in all prior deterministic execution systems is that they require a global *total ordering* of synchronization. Even when two threads acquire distinct locks, both acquires are forced into a deterministic total order. The total order is only really needed when two threads contend on the same lock – there the total order deterministically orders one contender before the other. While threads often acquire distinct locks, it is undecidable which lock a thread will acquire next so the total order must be enforced all the time, just in case.

Our Consequence system [14] was the first to demonstrate the irony of previous deterministic systems that adopted very relaxed memory consistency models but retained a total order on synchronization. Consequence was able to outperform these prior systems *and* provide stronger memory consistency through careful engineering and the first implementation of deterministic *blocking* synchronization. Previous deterministic synchronization mechanisms are all polling-based which is wasteful of processor resources. Unlike in a nondeterministic system, deterministic blocking is non-trivial because unblocking must be performed deterministically with respect to all threads in the system. As a result of these and other innovations, the Consequence system is more than 2x faster than previous deterministic execution systems.

We also found a way to smash through the total-order bottleneck completely with the LazyDet system [15]. We identified many instances in which the imposition of a total order on synchronization is unnecessarily pessimistic. LazyDet uses deterministic speculation to avoid stalling due to the total order, and later validates that the speculative execution was indistinguishable from what the total order would have enforced. Maintaining the illusion of a total order is needed to preserve determinism, but makes deterministic speculation much subtler than its nondeterministic counterpart. The upside is that deterministic speculation yields a big performance win when threads acquire distinct locks, as the total order is unnecessary in such cases. Compared to our prior Consequence work, LazyDet is nearly an order of magnitude faster on microbenchmarks with fine-grained locking, and 2x faster on several real applications.

Deterministic OS Abstractions

Most of the existing work on deterministic execution (ours included) has focused on shared memory parallelism. However, to bring the benefits of determinism to a wider audience we need to broaden the scope to support the complex multi-process jobs that people run, like software builds, data analytics pipelines, and computational science workflows. Our DetFlow system [16] takes an initial step in this space, providing a deterministic process group abstraction without requiring a custom OS like previous systems. Instead, DetFlow uses lightweight library call interception (via LD_PRELOAD) to intercept system calls and ensure that parallel processes are isolated from one another with respect to shared resources like the filesystem. Software builds and bioinformatics workflows running under DetFlow incur just a 2% performance overhead.

In ongoing work, our DetTrace system provides the first deterministic container abstraction. DetTrace uses Linux's ptrace mechanism to provide an airtight deterministic guarantee; in contrast, DetFlow's library call interception could be circumvented in various ways such as making system calls via inline assembly. While DetTrace's performance overhead is higher at around 4-5x, the

container abstraction makes DetTrace much more readily deployable. We've used DetTrace to build over 17,000 Debian packages deterministically. As a point of comparison, the Debian Reproducible Builds (DRB) community with dozens of volunteers took nearly two years to accomplish a similar feat, as they patch individual packages by hand to make them build deterministically. Our generic approach enforces determinism for anything inside the container. Moreover, 100% of the packages we build come out deterministic, while the DRB effort currently has only 92% of Debian packages building deterministically due to a long tail of corner cases, regressions, and so on.

The DetTrace technology also underpins the work at our startup, Cloudseal, which is building the world's first deterministic container service. Cloudseal containers seek to banish heisenbugs forever: any software that crashes or goes awry in our container can be effortlessly reproduced due to the container-level determinism guarantee. With SBIR Phase 1 funding from the NSF, we have hired our first two employees and anticipate an alpha product release by year end.

Conclusion

All of our projects share a consistent focus on improving the experience of writing software, whether it be automatically fixing performance bugs to save developer effort, or helping developers uncover subtle concurrency errors, or providing determinism to make it straightforward for a developer to reproduce bugs. As the processor industry turns to specialization to compensate for a lack of transistor scaling, computing systems have become increasingly complex and difficult to reason about. This makes work like ours ever more relevant and necessary to allow developers to remain productive in the face of massively heterogeneous hardware.

References

[1] C. DeLozier, A. Eizenberg, S. Hu, G. Pokam and J. Devietti, "TMI: Thread Memory Isolation for False Sharing Repair," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[2] T. A. Khan, Y. Zhao, G. Pokam, B. Mozafari and B. Kasikci, "Huron: Hybrid False Sharing Detection and Repair," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[3] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. Newburn and J. Devietti, "LASER: Light, Accurate Sharing Detection and Repair," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[4] A. Eizenberg, S. Hu, G. Pokam and J. Devietti, "Remix: Online Detection and Repair of Cache Contention for the JVM," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[5] J. Fineman, K. Agrawal, J. Devietti, I.-T. A. Lee, R. Utterback and C. Xu, "Race Detection and Reachability in Nearly Series-Parallel DAGs," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2018.

[6] W. Mansky, Y. Peng, S. Zdancewic and J. Devietti, "Verifying Dynamic Race Detection," in *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2017.

[7] C. Flanagan and S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[8] Y. Peng, C. DeLozier, A. Eizenberg, W. Mansky and J. Devietti, "SlimFast: Reducing Metadata Redundancy in Sound and Complete Dynamic Data Race Detection," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2018.

[9] Y. Peng, B. Wood and J. Devietti, "PARSNIP: Performant Architecture for Race Safety with No Impact on Precision," in *ACM IEEE International Symposium on Microarchitecture (MICRO)*, 2017.

[10] A. Eizenberg, Y. Peng, T. Pigli, W. Mansky and J. Devietti, "BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA Programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[11] Y. Peng, V. Grover and J. Devietti, "CURD: A Dynamic CUDA Race Detector," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[12] R. Alur, J. Devietti, O. S. N. Leija and N. Singhania, "GPUDrano: Detecting Uncoalesced Accesses in GPU Programs," in *Computer-Aided Verification (CAV)*, 2017.

[13] R. Alur, J. Devietti and N. Singhania, "Block-Size Independence for GPU Programs," in *Static Analysis Symposium (SAS)*, ***Radhia Cousot Young Researcher Best Paper Award***, 2018.

[14] T. Merrifield, J. Devietti and J. Eriksson, "High-Performance Determinism with Total Store Order Consistency," in *European Conference on Computer Systems (EuroSys)*, 2015.

[15] T. Merrifield, S. Roghanchi, J. Devietti and J. Eriksson, "Lazy Determinism for Faster Deterministic Multithreading," in *International Conference on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2019.

[16] R. G. Scott, O. S. Navarro Leija, J. Devietti and R. R. Newton, "Monadic Composition for Deterministic, Parallel Batch Processing," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2017.