POSTER: The Liberation Day of Nondeterministic Programs

Enrico A. Deiana *enrico.deiana@u.northwestern.edu*
Vincent St-Amour *stamourv@eecs.northwestern.edu*
Peter Dinda *pdinda@northwestern.edu*
Nikos Hardavellas *nikos@northwestern.edu*
Simone Campanoni *simonec@eecs.northwestern.edu*

I. Introduction

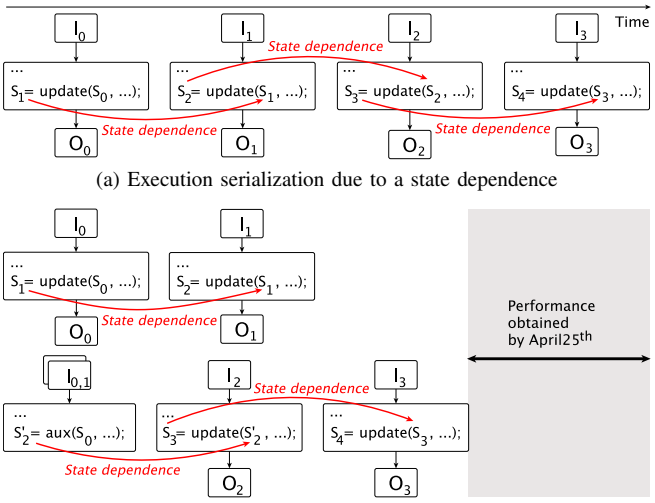
Increasing thread-level parallelism (TLP) is the chief way to maximize core utilization and compute performance on multicore systems. However, inter-thread data movements, which are necessary for satisfying dependences, impose a low limit on TLP and hence on performance. The research community has had tremendous success using techniques like thread-level speculation to reduce the impact of apparent dependences [1], [2], [3], [4]. Uncovering and ignoring apparent dependences, however, has reached the point of diminishing returns. To overcome this parallelism plateau, we must turn our attention to actual dependences.

While, strictly speaking, all actual dependences must be satisfied to preserve program semantics, strict semantics preservation is not always necessary; some programs allow variations in their outputs. Prior work has shown that breaking actual dependences can lead to important performance gains, if one is willing to accept some output distortion [5], [6], [7]. When output quality needs to be preserved, prior approaches are not applicable as they introduce inaccuracies if any actual dependence is broken.

Nondeterministic programs naturally produce different output results from run to run given the same inputs; this defines the program's inherent output variability. Often, this output variability originates from variations in the program's intermediate data. Here, an intermediate datum forwarded from a producer to a consumer may vary across runs for the same input. Hence, consumers are often designed to be resilient to such variations. Our hypothesis is that this resiliency may allow an alternative producer to generate data that are similar enough to satisfy the original consumer. This alternative producer can be decoupled from the original producer, therefore liberating additional TLP. When a producer and a consumer exchange data to share their state, we call the actual producer-consumer dependence a *state dependence*.

State dependences exist at the algorithm level and are out of the reach of automatic tools. Hence, we rely on the developer to make them explicit to our system. The state dependences we identified in the PARSEC benchmarks follow the read-after-write pattern shown in Figure 1a. We call the alternative producer *auxiliary code* because we use it as a substitute in case of need—when there is not enough TLP. The auxiliary code is automatically generated by our compiler (starting from the developer-provided description of the state dependence) and is specialized to a given state dependence. Our experiments show that properly-generated auxiliary code increases TLP while keeping output variations within the bounds of the original, nondeterministic program.

Research supported by NSF CNS-1405756, CCF-1453853, CCF-1533560, the DOE Hobbes project via Sandia National Labs, and Intel IPCC.



(a) Execution serialization due to a state dependence

(b) Additional TLP generated by auxiliary code

Figure 1: Alternative execution model obtained by using auxiliary code to satisfy a state dependence.

To the best of our knowledge, this work is the first to generate automatically additional code to satisfy an actual dependence with alternative data.

II. Our Approach: April25th

The April25th tool-chain liberates additional TLP of nondeterministic C++ programs that exhibit the pattern of Figure 1a, relying on additional algorithm-specific information and representative inputs from developers. It does so by exploring the design space generated by state dependences, and then choosing the configuration with the best profile (e.g., highest performance) whose output is within the program's inherent variability for all provided inputs.

Identifying state dependences requires algorithmic knowledge that is beyond the purview of automatic tools. Hence, developers have to make the state dependence pattern explicit and provide it to April25th, along with a description of algorithm-specific tradeoffs. Tradeoffs are pieces of program text (constants, data types, functions) whose value is chosen from a range given by the developer. Such tradeoffs are

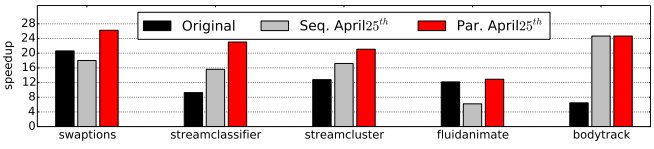


Figure 2: Peak performance.

used by April25^{th} only to balance quality and performance in auxiliary code. The encoding of state dependences and tradeoffs is done through the *State Dependence* and *Tradeoff Interfaces* (respectively SDI and TI) that April25^{th} exposes to the developer. Developers also need to provide representative inputs to determine the program's inherent output variability, and a function to measure the distortion of the output with respect to the original, non-distorted output.

April25^{th} includes an autotuner, a profiler, and three compilers called front-end, middle-end, and back-end. The *front-end* compiler translates C++ extended with the SDI and TI to standard C++ code, encoding April25^{th}-specific information in API calls understood by the other April25^{th} compilers. The front-end also estimates the program's inherent output variability by running the original program's binary. The *middle-end* compiler translates the output of the front-end into our intermediate representation (IR). It embeds the design space in the IR, which is defined by the tradeoffs described using the TI, how often a state dependence is satisfied with auxiliary code, and the number of threads to dedicate to the TLP available in the original program.

The description of the design space is used by the *autotuner*, which explores it by choosing the next configuration to test. The *back-end* compiler translates our IR into the binary that corresponds to the configuration chosen by the autotuner. The *profiler* runs the binary generated by the back-end using the representative inputs, measuring its performance and output quality (using a developer-supplied metric) for consumption by the autotuner. The autotuner then decides whether or not to test other configurations iterating over the loop just now described.

When enough information has been obtained, April25^{th} generates the binary of the most performant configuration it has seen, considering only configurations whose outputs fall within the program's inherent output variability for all the provided inputs.

III. Evaluation

Our evaluation tests the hypothesis behind our work—that state dependences can be satisfied with auxiliary code. To do so, we built April25^{th} on top of LLVM 3.9.1, Racket 6.8, and Opentuner 0.5.0.

We consider PARSEC benchmarks that exhibit nondeterminism: bodytrack, fluidanimate, swaptions, and two variants of streamcluster (clustering, called streamcluster, and classification, called streamclassifier). The state dependences we find are all related to state updates like the human body model in bodytrack, the fluid state in fluidanimate, the swaption price in swaptions, and the clusters of points in streamcluster and streamclassifier. The tradeoffs we encode in auxiliary code are the number of original threads, the number of threads to use for state dependences, and algorithm-specific tradeoffs (e.g., the number of annealing layers used by bodytrack's particle filter, the number of particles to discard from the computation in fluidanimate). To measure output quality we use well-known domain-specific metrics (e.g., average Euclidean distance between the position of the particles in fluidanimate) [8], [9], [10]. We evaluate our system on a machine with two Intel Xeon E5-2695 v3 Haswell processors.

Satisfying state dependences with auxiliary code liberates important additional TLP. Figure 2 compares the speedup of three approaches to parallelizing the benchmarks. The first, "Original", is the out-of-the-box benchmark that has been parallelized by traditional means. The second, "Seq. April25^{th}", uses only the TLP liberated by April25^{th}. The third, "Par. April25^{th}", combines the TLP available in the original code with the one liberated by April25^{th}. All approaches maintain output quality within the original program's output variability. Speedup is computed using the single-threaded version of the out-of-the-box benchmark as baseline. Results shown in Figure 2 empirically support our hypothesis: state dependences can be satisfied with auxiliary code. This enables April25^{th} to liberate additional TLP from multi-threaded, non-deterministic programs.

IV. Conclusion

Actual dependences have been either satisfied or broken by prior work. We propose an intermediate solution for a subset of actual dependencies (i.e., state dependences) common to nondeterministic programs: satisfy a state dependence with auxiliary code. This work is the first step in exploiting state dependences, and does so without introducing distortion in the program's output.

References

[1] L. Hammond *et al.*, "The Stanford Hydra CMP," in *International Symposium on Microarchitecture (MICRO)*, 2000.
[2] W. Liu *et al.*, "POSH: A TLS Compiler That Exploits Program Structure," in *Principles and Practice of Parallel Programming (PPoPP)*, 2006.
[3] G. S. Sohi *et al.*, "Multiscalar processors," in *International Symposium on Computer Architecture (ISCA)*, 1995.
[4] J. G. Steffan *et al.*, "The STAMPede Approach to Thread-level Speculation," in *Transactions on Computer Systems (TOC)*, 2005.
[5] S. Campanoni *et al.*, "HELIX-UP: Relaxing Program Semantics to Unleash Parallelization," in *Code Generation and Optimization (CGO)*, 2015.
[6] M. Samadi *et al.*, "Sage: Self-tuning approximation for graphics engines," in *International Symposium on Microarchitecture (MICRO)*, 2013.
[7] X. Sui *et al.*, "Proactive Control of Approximate Programs," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
[8] D. L. Davies and D. W. Bouldin, "A Cluster Separation Measure," in *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 1979.
[9] H. Hoffmann *et al.*, "Dynamic Knobs for Responsive Power-aware Computing," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
[10] S. Misailovic *et al.*, "Quality of Service Profiling," in *International Conference on Software Engineering (ICSE)*, 2010.