Harmony: Co-Optimizing Parallelism and Locality to Bound Performance

Jennifer Brana, Nathan Beckmann
jbrana@cs.cmu.edu beckmann@cs.cmu.edu
Carnegie Mellon University

ABSTRACT

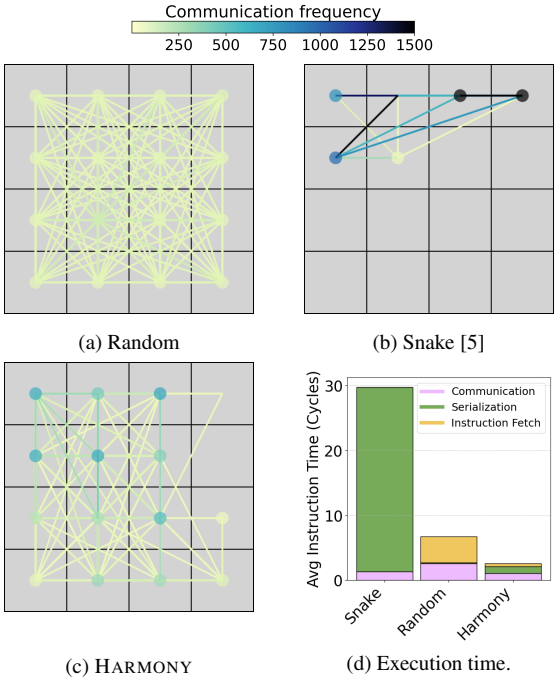
To maximize performance, systems must balance parallelism and locality. Architectures often neglects this tradeoff, optimizing for only parallelism or locality. Prior work has neglected this tradeoff, instead optimizing for only parallelism or locality. We explore an abstract processor, CHAOS, as a limit study to evaluate the bounds of performance when jointly optimizing for parallelism and locality. We find that co-optimizing parallelism, data locality, *and* instruction locality is essential: our optimized scheduling algorithm, HARMONY, outperforms all other methods and achieves near ideal scaling.

1 Introduction and Background

There is a fundamental tradeoff in computing between parallelism and locality. To improve performance, processors distribute instructions across parallel processors. As parallelism increases, locality is lost as instructions are distributed far apart, causing data movement from communication and instruction fetch (ifetch) to dominate execution. Architectures limit data movement by exploiting locality, i.e., by co-locating instances of the same instruction or instructions that communicate with each other. However, co-location can increase processor contention, serializing independent operations. To maximize performance, systems must find the optimal balance between parallelism and locality.

Limitations of existing architectures. Unfortunately, existing architectures fail to consider parallelism and locality together. Traditional out-of-order (OoO) superscalars improve parallelism by eliminating false dependencies to expose dynamic instruction-level parallelism (ILP) *within* and *across* loops. However, this approach compromises locality, as rewriting registers destroys dataflow locality (i.e. the predictability of data dependencies), forcing cores to distribute instructions irrespective of dependencies (increasing communication latency) and switch instructions every cycle (increasing ifetches) [8, 9]. In contrast, spatial architectures optimize for locality by unrolling program dataflow graphs (DFG) spatially in hardware such that dependent instructions are close together (to reduce communication latency) [2, 6, 9]. This requires each static instruction is bound to a processing element (PE) such that all dynamic instances are executed on the same PE (eliminating instruction fetches). Binding instructions statically can serialize dynamic instances of the instruction, limiting parallelism [1].

Goal: Understanding the limits of parallelism & locality. This work seeks an instruction placement method for parallel processors that finds the optimal balance between parallelism and locality. Fig. 1 illustrates how instruction execution is impacted by varying levels of parallelism and locality. A naïve way to expand parallelism is to assign dynamic instructions to random PEs. While Random achieves high parallelism, it ignores locality, causing data communication and ifetch to dominate execution time. By contrast, Snake (adapted from [5]) optimizes for locality by binding each *static* instruction to a single PE and placing dependent instructions close together, following a "snake" pattern. However, optimizing for locality by decreasing parallelism leads to instruction serialization. Our method, HARMONY, jointly considers parallelism and locality to increase parallelism without degrading performance.



(a) Random (b) Snake [5] (c) HARMONY (d) Execution time.

Figure 1: Frequency of communication between PEs and breakdown of average instruction execution time for SpMV. See Fig. 2 for performance results. Lines represent inter-PE communication, and circles depict intra-PE communication. Random has significant communication due to frequent messaging between a large number of PEs with no clear locality. Snake, adapted from [5], has sparse and localized communication, however, instructions execute on a limited number of PEs, causing significant serialization delay. Harmony co-optimizes for parallelism and locality to find the balance that maximizes performance.

Problem. Although prior work has recognized a tradeoff between parallelism and locality, prior work has explored the tradeoff only within specific architectures [5, 7], where inessential design choices cloud the fundamental challenges. To explore the tradeoff, architects need a simple model that captures parallelism, data locality, and instruction locality, free of other constraints.

CHAOS*: Our processor model where anything goes.* To facilitate exploration of the tradeoff space, we develop a new spatial processor model, CHAOS, that enables exploiting

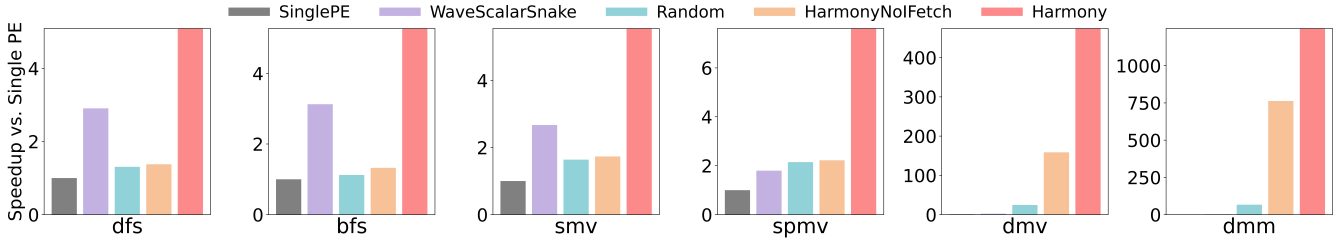


Figure 2: Speedup over a single PE across all apps and placement methods. Fabric sizes are selected based on the maximum IPC achievable in the app. HARMONY vastly outperforms all other placement methods. HARMONY considers parallelism, instruction locality, and dataflow locality during scheduling, allowing it to achieve high performance for apps with variable levels of parallelism and locality.

parallelism, dataflow locality, and instruction locality in a single framework. CHAOS incorporates the dynamism of OoO scheduling to throttle parallelism and a spatial execution model to expose control of dataflow and instruction locality. CHAOS uses placement methods to assign instructions to a PE, and executes them once their input operands have arrived at the PE, their instruction has been fetched, and the PE is idle. CHAOS models the breakdown of instruction execution time by measuring the factors that delay instruction firing.

HARMONY*: Our method to co-optimize parallelism & locality.* To evaluate the potential of co-optimizing for parallelism and locality, we develop a new placement method, HARMONY. HARMONY places instructions by performing a local search over space and time, scheduling the instruction at the earliest available PE, according to locality and already-scheduled instructions. HARMONY maximizes performance by exploiting parallelism without sacrificing locality.

2 Preliminary Study and Results

2.1 CHAOS Processor Model

CHAOS models a spatial fabric of simple PEs (each containing some instruction cache) connected by a dynamic on-chip network. We observe that performance of instruction placement is influenced by three factors: distance from operand producers, whether the instruction is cached in the PE or not, and serialization due to contention for PE execution resources. Instruction execution is broken into three factors: communication delay, ifetch delay, and serialization delay.

To explain CHAOS, we describe an instruction execution. A dynamic instruction instance is scheduled to a PE when the first producer of the instruction generates data. Producers explicitly send data to the PE the instruction is scheduled on. Communication delay is measured as the difference between when the last input was generated and when the instruction receives all its inputs. When all inputs have been received, the icache is checked and the instruction is fetched from memory if necessary. The time to fetch the instruction is recorded as ifetch delay. Once an instruction is ready to execute (has all input operands and the instruction), it enqueues to the PE's FIFO readylist. The difference between when the instruction enters the readylist and when it executes constitutes serialization delay. If the instruction generates data operands for other instructions, it sends the data after executing and schedules consumers if necessary.

2.2 HARMONY Scheduling Algorithm

HARMONY uses knowledge of instruction delays to optimize performance by throttling parallelism and locality. When scheduling an instruction, HARMONY iteratively builds a list of candidate PEs to schedule to. During the first pass, the PEs reachable fastest (by best-time) by an instruction's operand producers are entered into the list at the earliest time the PE can execute the instruction. PEs compute when they can execute an instruction based on the potential ifetch delay and if there are other instructions in its readylist that will cause serialization delay. If no PEs are available at best-time, PEs one cycle from current candidates are added to the list and the next cycle is checked. This is repeated until a PE is found.

2.3 Experimental Methodology

We perform trace-driven simulation on six common applications: DFS, BFS, SMV, SpMV, DMV, and DMM. We compare HARMONY against: *(i)* SinglePE, *(ii)* WaveScalarSnake, *(iii)* Random, and *(iv)* HarmonyNoIFetch. Placement methods fall into two categories: static placement where static instructions are bound to a PE such that every dynamic instance is executed on the same PE and dynamic placement where dynamic instances are scheduled independently. SinglePE represents a baseline single-core system which we compare against throughout the paper. WaveScalarSnake, based on [5], is a static placement method that fills PEs sequentially and places dependent instructions close together. Random is a dynamic placement method that distributed instructions randomly. Finally, HarmonyNoIFetch is a dynamic placement method similar to HARMONY except is disregards instruction locality during scheduling.

2.4 Evaluation

Fig. 2 shows the speedup for every app on each system. Starting from the baseline of a single PE, the next three bars demonstrate the importance of each element Harmony optimizes. WaveScalarSnake exploits locality, but ignores parallelism, limiting performance and scalability. Random adds in parallelism, but ignores locality, and performance suffers. HarmonyNoIFetch exploits parallelism and dataflow locality, but ignores instruction locality, and performance is still suboptimal. Only HARMONY considers all aspects of the tradeoff, and achieves by far the best performance as a result, averaging a speedup of 1.98× over the next best method.

3 Future Work

Expanded Limit Study. In the future, we plan to evaluate larger, real world applications such as SPEC CPU2017 and

Mediabench [4]. Additionally, we plan to compare against prior schedulers used in OoO cores and spatial architectures.

Instruction Bottleneck Analysis. To quantify the tradeoff between parallelism and locality, we plan to model how different instruction delays contribute to performance by developing an analytical model such as [3, 5].

Application to Future Hardware Design. We plan to use the insights from our limit study to design an architecture that can optimize parallelism, data locality, and instruction locality.

REFERENCES

[1] N. Agarwal, M. Fream, S. Ghosh, B. C. Schwedock, and N. Beckmann, " The TYR Dataflow Architecture: Improving Locality by Taming Parallelism ," 2024.

[2] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder, "Scaling to the end of silicon with edge architectures," *Computer*, 2004.

[3] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," 2006.

[4] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," 1997.

[5] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers, "Modeling instruction placement on a spatial architecture," 2006.

[6] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A design space evaluation of grid processor architectures," 2001.

[7] R. Nagarajan, S. Kushwaha, D. Burger, K. McKinley, C. Lin, and S. Keckler, "Static placement, dynamic issue (spdi) scheduling for edge architectures," 2004.

[8] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," 1997.

[9] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proc. of the 36th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-36)*, 2003.