# Decoupled Architectures as a Low-Complexity Alternative to Out-of-order Execution

Neal C. Crago and Sanjay J. Patel
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL USA
(crago, sjp)@illinois.edu

***Abstract*— In this paper we present OUTRIDERHP, a novel implementation of a decoupled architecture that approaches the performance of contemporary out-of-order processors on parallel benchmarks while maintaining low hardware complexity. OUTRIDERHP leverages the compiler to separate a single thread of execution into *memory-accessing* and *memory-consuming* streams that can be executed concurrently, which we call `strands`. We identify loss-of-decoupling events which cripple performance on traditional decoupled architectures, and design OUTRIDERHP to enable extraction of multiple strands and control speculation which provide superior memory and functional unit latency tolerance. OUTRIDERHP outperforms a baseline in-order architecture by 26-220% and Decoupled Access/Execute by 7-172% when executing parallel benchmarks on an 8-core CMP configuration. OUTRIDERHP performs within 15% of higher-complexity out-of-order cores despite not utilizing large physical register files, dynamic scheduling, and register renaming hardware.**

## I. INTRODUCTION

Execution stalls due to memory and functional unit latency are the most significant limiting factor for performance in parallel workloads in multicore systems. We find in our eight parallel benchmarks that removing stalls due to memory latency increases performance by nearly 230% on average, while idealizing functional unit latency improves performance by an additional 70%. The performance impact of both the memory and functional unit latencies indicates the importance of efficient latency tolerance mechanisms.

Out-of-order (OOO) processors enable functional and memory latency tolerance by executing instructions out-of-order with respect to one another. However, OOO requires significant hardware structures such as associative instruction windows, large register files, register renaming, and load store queues. To combat this large complexity, techniques have been proposed using less complex hardware at some performance cost [1], [2], [3]. However, these approaches still leverage significant hardware such as register renaming and instruction windows to buffer in-flight instructions.

## II. DECOUPLED ARCHITECTURES

Traditional decoupled architectures divide the *memory-access* and *memory-consuming* instructions into separate instruction streams called `strands`, which execute in different hardware contexts and communicate data and control flow decisions with one another [4]. Decoupling enables the memory-accessing strands to continue executing while the memory-consuming strands wait on data from memory, thus tolerating memory latency. Decoupled architectures execute instructions out-of-order, but this parallelism is extracted by the compiler rather than in hardware, leading to significantly simpler hardware similar to in-order designs.

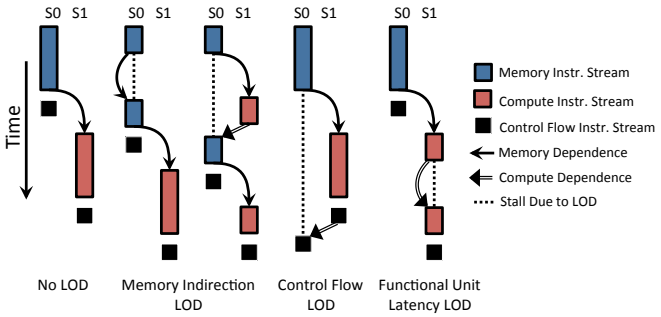


Fig. 1. Loss-of-decoupling (LOD) events that can severely limit performance for traditional two-strand decoupled architectures such as DAE.

Although decoupled architectures enable memory latency tolerance, potential performance improvement is limited due to *loss-of-decoupling* (LOD) events which result from inter- and intra-strand dependences. Past work has investigated handling memory indirection LOD [5]. However, data-dependent control flow causes memory-accessing strands to block and wait for the control flow decision, reducing the benefit of decoupling. We also identify LOD caused by exposed functional unit latencies, which reduce the rate at which a memory-consuming strand can consume data from the FIFO queue, causing the producing memory-accessing strand to eventually stall when the data queue becomes full. We address these LOD events in our high-performance decoupled architecture OUTRIDERHP.

## III. ARCHITECTURE AND CODE PARTITIONING

Figure 2 presents the OUTRIDERHP architecture. In addition to hardware required for an in-order processor, OUTRIDERHP includes additional low-complexity hardware such as small register files, data queues for communication, and a memory access unit (MAU) for multiple outstanding memory requests. To reduce LOD due to data-dependent control flow, we provide control speculation through checkpointing. Memory indirection and functional unit latency LOD is avoided through extracting additional strands.

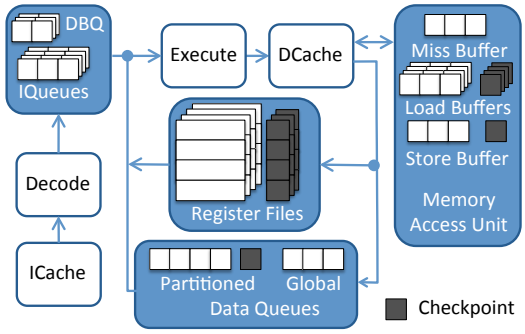


Fig. 2. OUTRIDERHP expands a traditional in-order pipeline with instruction queues, data queues, a memory access unit, and separate register files.

We build upon the partitioning scheme from [5]. OUTRIDERHP extracts strands from the original thread by examining the dependency graph and partitioning the program along memory-access and memory-consumption lines. Memory indirection LOD is avoided by splitting the original memory-accessing stream into multiple strands by separating dependent memory accesses, with the goal of having at least one strand without an LOD event. To facilitate functional unit latency tolerance, we split chains of floating point instruction into separate strands along producing and consuming lines, similar to the memory dependence partitioning.

We enable control speculation through checkpointing the register file, data queue tail pointers, and MAU tail pointers using SRAM shadow-bitcells [6] to reduce overhead. When an instruction in a memory-accessing strand reaches a branch that requires data from a memory-consuming strand, The memory-accessing strand enters a speculative mode. The waiting branch instruction and the prediction is deferred to a special FIFO queue known as the *deferred branch queue (DBQ)*, a checkpoint is made of the strand's state and the strand speculatively continues on the predicted path provided by the branch predictor. When the correct control flow decision is available from the memory-consuming strand, the prediction is compared and the state is restored from the checkpoint if a misprediction occurred.

## IV. EVALUATION

We compare OUTRIDERHP with baseline in-order, DAE [4], and both contemporary high-performance (OOO-HP) and complexity effective (OOO-CE) architectures [1]. The simulator is execution-driven and models an 8-core CMP architecture with three levels of cache hierarchy. Each baseline core is four-wide issue in-order with private 32KB L1 caches and a RISC instruction set. The eight cores share a unified 512KB L2 cache and the 4MB L3 Cache which is connected to memory controllers and offchip DRAM. The OOO processors have a 256-entry physical register file, a 128-entry ROB, and a 128-entry instruction window either unified associative lookup (OOO-HP) or 8-way dependence-base steering (OOO-CE). Both DAE and OUTRIDERHP has a 16-entry register file per strand and a 64-entry partitioned data queue, while OUTRIDERHP supports up to 4 strands and has 2 checkpoints for control speculation.

For evaluation, we use eight optimized parallel kernels written using the task queue model from Rigel [7]. The benchmarks include conjugate gradient linear solver (`cg`), columbic potential with cutoff (`cutcp`), 2D fast fourier transform (`fft`), 2D stencil (`heat`), k-means clustering (`kmeans`), mergesort (`merge`), mri reconstruction (`mri`), and edge detection (`sobel`).

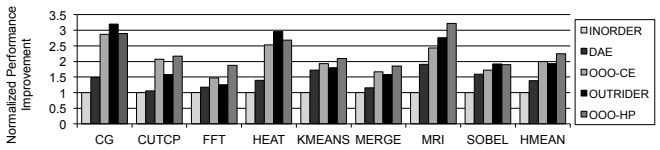


Fig. 3. Overall performance of 4-wide in-order baseline, DAE, OUTRIDERHP architecture, and OOO designs, both high-performance (OOO-HP) and complexity-effective (OOO-CE).

Figure 3 presents the overall performance of the evaluated architectures. On average, OUTRIDERHP outperforms DAE by 54% and the in-order baseline by 92%. Additionally, OUTRIDERHP performs within 15% of OOO-HP, and on par with OOO-CE, despite not having hardware register renaming, dynamic scheduling, and large instruction windows. OUTRIDERHP actually outperforms OOO on `cg` and `heat` by utilizing non-blocking strands instead of instruction windows which fill up quickly in cache-miss intensive applications. We find that OOO instruction windows mostly benefit control and compute intensive codes with long floating-point dependency chains such as in `mri`, and is the single-most the reason for the gap between OOO processors and OUTRIDERHP. However, by extracting additional strands, OUTRIDERHP can still tolerate some degree of floating-point unit latency and even improves performance greatly over DAE on the `cutcp` and `mergesort` benchmarks through utilizing control speculation.

## REFERENCES

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proc. of the 24th Intl. Symp. on Computer Architecture*, 1997, pp. 206–218.

[2] F. Tseng and Y. N. Patt, "Achieving out-of-order performance with almost in-order complexity," in *Proc. of the 35th Intl. Symp. on Computer Architecture*, 2008, pp. 3–12.

[3] A. Hilton, S. Nagarakatte, and A. Roth, "iCFP: Tolerating all-level cache misses in in-order processors," *IEEE Micro*, vol. 30, no. 1, pp. 12–19, 2010.

[4] J. E. Smith, "Decoupled access/execute computer architectures," in *Proc. of the 9th Intl. Symp. on Computer Architecture*, 1982, pp. 112–119.

[5] N. C. Crago and S. J. Patel, "Outrider: efficient memory latency tolerance with decoupled strands," in *Proc. of the 38th Intl. Symp. on Computer Architecture*, 2011, pp. 117–128.

[6] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose, "Increasing processor performance through early register release," in *Proc. of 22nd IEEE Intl. Conf. on Computer Design*, 2004, pp. 480 – 487.

[7] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An architecture and scalable programming interface for a 1000-core accelerator," in *Proc. of the 36th Intl. Symp. on Computer Architecture*, 2009, pp. 140–151.