SHADOW: Simultaneous Multi-Threading Architecture with Asymmetric Threads

Ishita Chaturvedi*
Princeton University
Princeton, NJ, USA

Bhargav Reddy Godala
Ahead Computing
Portland, OR, USA

Abiram Gangavaram
Princeton University
Princeton, NJ, USA

Daniel Flyer
Princeton University
Princeton, NJ, USA

Tyler Sorensen
University of California, Santa Cruz and Microsoft
Santa Cruz, CA, USA

Tor M. Aamodt
University of British Columbia
Vancouver, BC, Canada

David I. August
Princeton University
Princeton, NJ, USA

Abstract

Many important applications exhibit shifting demands between instruction-level parallelism (ILP) and thread-level parallelism (TLP) due to irregular sparsity and unpredictable memory access patterns. Conventional CPUs optimize for one but fail to balance both, leading to underutilized execution resources and performance bottlenecks. Addressing this challenge requires an architecture that can seamlessly adapt to workload variations while maintaining efficiency.

This paper presents SHADOW, the first asymmetric SMT core that dynamically balances ILP and TLP by executing out-of-order (OoO) and in-order (InO) threads simultaneously on the same core. SHADOW maximizes CPU utilization by leveraging deep ILP in the OoO thread and high TLP in lightweight InO threads. It is runtime-configurable, allowing applications to optimize the mix of OoO and InO execution. Evaluated on nine diverse benchmarks, SHADOW achieves up to 3.16× speedup and 1.33× average improvement over an OoO CPU, with just 1% area and power overhead. By dynamically adapting to workload characteristics, SHADOW outperforms conventional architectures, efficiently accelerating memory-bound workloads without compromising compute-bound performance.

Keywords

Asymmetric CPU microarchitecture, Simultaneous multithreading (SMT), Heterogeneous thread execution, Dynamic ILP-TLP balancing, Software work stealing, Sparse workloads, Memory-bound workload acceleration, Thread-level parallelism (TLP), Instruction-level parallelism (ILP), Sparse matrix multiplication (SpMM), Low-overhead microarchitectural design

1 Introduction

Memory-bound applications, such as deep learning workloads [25] and high-performance computing kernels [42], often exhibit irregular memory access patterns and data sparsity. These characteristics cause fluctuations between instruction-level parallelism (ILP) and thread-level parallelism (TLP) during execution [33]. Traditional CPUs are typically optimized for either ILP or TLP, with out-of-order (OoO) cores focusing on ILP and lean in-order (InO) cores on TLP. This specialization can lead to underutilization of resources and performance inefficiencies when handling applications with dynamic parallelism requirements.

To address CPU inefficiencies, domain-specific accelerators have been developed to improve memory-bound performance [20, 46, 47, 62, 65]. While these accelerators achieve high throughput, they often sacrifice programmability, making them unsuitable for general-purpose workloads. GPUs provide greater flexibility but struggle with irregular memory access patterns, limiting their efficiency in sparse workloads [15]. Meanwhile, CPUs remain the most programmable option but struggle to adapt dynamically to shifting ILP and TLP demands. While OoO CPUs effectively exploit ILP by reordering instructions, their performance deteriorates in sparse and memory-bound workloads, where ILP opportunities diminish. This results in inefficiencies when handling high-latency workloads such as sparse matrix-vector multiplication (SpMV) in iterative solvers [51], graph traversal for social network analysis [37], database query processing [48], stencil computations in high-performance computing [17], and deep learning workloads with large embeddings for recommendation systems [22].

Evolving Beyond Traditional SMT. Simultaneous multithreading (SMT) improves execution unit utilization by allowing multiple threads to share resources. However, conventional SMT architectures trade off between ILP and TLP, either supporting a few OoO threads for ILP [39] or many in-order threads for TLP [26]. For example, Intel's SMT designs prioritize ILP by supporting only two OoO threads per core. Prior work has explored adaptive SMT execution. MorphCore [56] switches between a 2-thread deep OoO

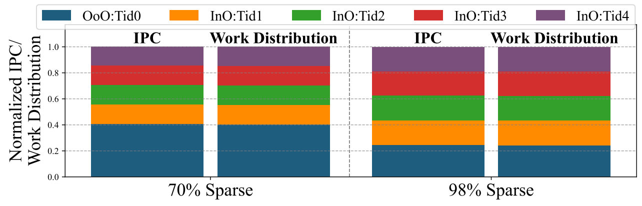

Figure 1: SHADOW dynamically redistributes work as IPC changes. High ILP skews execution toward the OoO thread, while work distributes equally among threads when the ILP is low. SHADOW adapts to the application without software intervention.

*Also with Cerebras Systems (current address).

MICRO 2025, October 18–22, 2025, Seoul, Korea
2025. ACM ISBN 978-X-XXXX-XXXX-X/XX/XX
https://doi.org/XXXXXXX.XXXXXXX

mode and an 8-thread wide in-order mode but lacks simultaneous ILP-TLP execution, leading to underutilization when workloads do not fit neatly into one mode. FIFO Shelf [53] and FIFOrder [6] speculatively routes instructions between OoO and in-order paths, but at the cost of increased hardware complexity eg, cross-path tracking, speculative wakeup, recovery.

This paper proposes a fundamentally different approach to balancing ILP and TLP: SHADOW executes OoO and InO threads concurrently within a single core, dynamically redistributing work without speculative instruction steering. Unlike prior approaches that switch execution modes, SHADOW enables simultaneous deep ILP and wide TLP execution, maximizing resource utilization.

Figure 1 illustrates how SHADOW dynamically adjusts work distribution. At low sparsity, minimal cache misses allow OoO threads to achieve high IPC, greedily taking more iterations via software-driven work-stealing. As sparsity increases and L1 D-cache misses rise, OoO efficiency drops. When memory stalls limit OoO throughput, InO threads, free from speculative overhead, continue stealing work to sustain execution. This adaptation occurs without explicit hardware coordination, threads independently steal work as it becomes available, naturally distributing tasks based on ILP and TLP demands. While SHADOW, like MorphCore, partitions the physical register file among threads, a key distinction is that MorphCore operates in either OoO or InO mode at a given time, whereas SHADOW enables the simultaneous execution of both (See Section 3.3). This co-execution allows SHADOW to dynamically balance ILP and TLP within a single core, adapting more effectively to workload demands.

A key advantage of this hybrid approach is its efficiency in terms of area and power. Adding additional in-order threads to an existing core increases area by only 1%, whereas adding a separate core would incur a significant increase in area overhead. Furthermore, the base core itself is often underutilized, making it more cost-effective to fully utilize existing resources before introducing additional cores. By enabling in-order and out-of-order execution within a single core, SHADOW maximizes utilization while minimizing power and area costs, offering an efficient, general-purpose solution for memory-bound workloads without the complexity of speculative instruction steering.

Contributions of This Work.

- A comprehensive study of ILP-TLP tradeoffs in modern CPUs, demonstrating the inefficiencies of traditional SMT architectures and highlighting the need for asymmetric multithreading (Section 2).
- Through SHADOW, we demonstrate that an asymmetric SMT core can efficiently balance ILP and TLP by enabling simultaneous execution of in-order and out-of-order threads, maximizing performance with minimal area and power overhead (Section 3).
- An extensive evaluation of SHADOW across diverse workloads, spanning compute-bound and memory-intensive applications, demonstrates its adaptability to varying cache pressures, with a detailed study on Sparse Matrix Multiplication (SpMM) (Section 5).

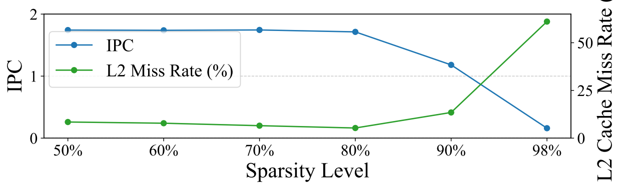

Figure 2: Change in the IPC and L2 cache misses of the SpMM application with varying sparsity.

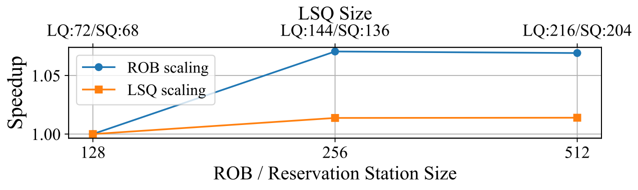

Figure 3: Speedup of 95%-sparse SpMM on a single OoO thread, showing diminishing returns from enlarging the ROB/RS or LSQ sizes.

2 Background and Motivation

Modern CPUs struggle to balance ILP and TLP, leading to underutilization. OoO execution extracts ILP but stalls on cache misses in memory-bound workloads, while InO execution exploits TLP but cannot hide long-latency memory operations. This section examines SMT's limitations in adapting to shifting ILP/TLP demands and introduces SHADOW, an asymmetric SMT architecture that integrates OoO and InO threads within a single core, dynamically redistributing execution resources without speculative overhead.

2.1 CPU Underutilization in Memory-Bound Workloads

Memory-bound applications fluctuate between ILP- and TLP-friendly phases, leading to CPU underutilization when execution resources sit idle due to limited ILP or TLP extraction.

Challenges in Out-of-Order Execution. OoO processors mitigate memory stalls using a large instruction window via the reorder buffer (ROB) and reservation stations (RS). However, these resources deplete quickly in memory-bound workloads. Figure 2 shows that in SpMM, increasing sparsity leads to rising L2 cache misses, severely degrading IPC as instruction windows remain underutilized.

Expanding the ROB and RS increases the instruction window, potentially improving performance in memory-bound workloads [40]. Figure 3 isolates this effect by (a) scaling only the ROB and RS and (b) scaling only load and store queue (LSQ) entries, while keeping all other microarchitectural structures constant, using a gem5 simulator configured as described in Table 1. A larger ROB yields a modest 7% IPC gain but significantly increases wakeup-select CAM complexity, leading to high area and power overhead. The diminishing returns suggest that ROB scaling alone is insufficient for mitigating stalls in memory-bound scenarios. Similarly, increasing LSQ capacity has minimal impact: once the ROB is saturated by in-flight loads and their dependences, new instructions stall before

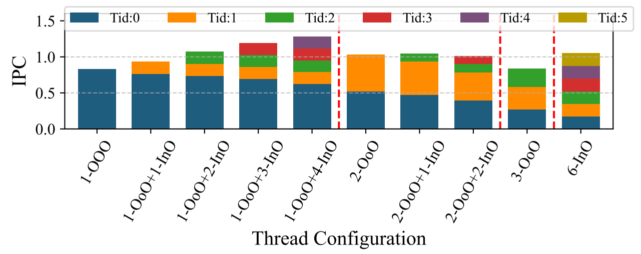

Figure 4: Breakdown of IPC contributions from all threads for 95% sparse SpMM.

the LSQ fills, keeping LSQ structural hazards off the critical path. While increasing the instruction window improves performance in ILP-heavy workloads, it does not effectively leverage the TLP present in memory-bound workloads like SpMM. To fully exploit available parallelism, CPUs must also incorporate mechanisms to distribute work across multiple threads.

Challenges in Thread-Level Parallelism. Memory-bound workloads like SpMM exhibit inherent TLP, as matrix rows can be processed independently. However, traditional SMT struggles with resource allocation, causing contention among threads. Figure 4 isolates the impact of increasing thread count while keeping the ROB, LSQ, and physical registers fixed, modeled using the gem5 simulator (detailed later in Table 1) on 95% sparse SpMM. The results show that adding OoO threads initially improves IPC, but gains diminish beyond two threads due to ROB saturation and register file pressure.

The results indicate that while adding more OoO threads initially boosts IPC, the benefits diminish beyond two OoO threads due to ROB saturation and increased register file pressure. In contrast, the 1 OoO + 4 InO configuration achieves higher IPC than a 3 OoO setup by leveraging in-order threads, which rely solely on the architectural register file (ARF) without adding speculative execution overhead. However, a fully in-order configuration with 6 InO threads struggles in workloads requiring high ILP, highlighting the importance of SHADOW's hybrid execution model, which dynamically balances ILP and TLP to optimize performance.

2.2 Limitations of Traditional SMT Architectures

Existing SMT implementations do not dynamically adjust ILP/TLP execution balance, leading to suboptimal utilization. Two major prior SMT approaches have attempted to address this issue:

MorphCore [56]: Mode-Switching SMT. MorphCore allows a CPU core to switch between two execution modes based on the number of threads used by the application: (1) A deep OoO mode (up to 2-wide SMT) optimized for ILP. (2) A wide InO mode (up to 8-wide SMT) optimized for TLP. However, this approach does not allow ILP and TLP execution to coexist, leading to underutilization when workloads do not fit neatly into one execution mode.

FIFO Shelf [53] and FIFOrder [6]: Speculative SMT. FIFO Shelf and FIFOrder show that decode-stage logic can steer instructions within a single thread between an OoO engine and an in-order FIFO path, enabling fine-grained ILP/TLP adaptation at modest hardware cost. However, this requires mechanisms for correctness—cross-path dependency tracking, speculative wakeup, and misprediction recovery. SHADOW instead partitions at thread granularity, assigning each software thread wholly to either an OoO or an InO lane.

2.3 Why a Hybrid SMT Core is Needed Over Separate OoO and InO Cores

An alternative to asymmetric SMT is using separate InO and OoO cores instead of merging them into a single hybrid core. However, this approach is inefficient due to the following reasons:

(1) **Area Efficiency:** As we will show, when done judiciously, adding extra in-order threads to an existing core increases area by only 1%, whereas adding a separate core incurs a significant area overhead. (2) **Power Efficiency:** Hybrid SMT allows better resource sharing within a single core, reducing redundant hardware replication. (3) **Execution Resource Utilization:** The base core is often underutilized, so SHADOW maximizes its available resources instead of relying on additional cores. By integrating OoO threads and multiple InO threads within the same core, SHADOW achieves higher utilization at minimal cost.

2.4 SHADOW: A Balanced ILP-TLP Execution Model

The limitations of prior SMT approaches highlight the need for a hybrid execution model that can simultaneously exploit ILP while efficiently scaling TLP. SHADOW achieves this through:

- OoO threads to extract ILP where possible.
- Multiple lightweight InO threads to exploit TLP without incurring OoO execution overhead.
- A simple, software-based work-stealing mechanism to dynamically redistribute work based on execution demand.

By concurrently leveraging ILP and TLP, SHADOW achieves higher efficiency than existing SMT designs, making it a scalable solution for modern memory-bound workloads.

3 SHADOW

SHADOW is the first **asymmetric simultaneous multithreading (SMT) architecture** to integrate OoO and InO threads within a single core, dynamically balancing ILP and TLP without speculative instruction steering or execution mode switching. InO threads execute strictly sequentially without speculation or register renaming, allowing SHADOW to efficiently adapt to workload demands, particularly in memory-bound applications.

Figure 5 illustrates SHADOW's microarchitecture. Each cycle, a thread fetches ① and decodes instructions ②. OoO instructions are renamed ③ and placed in the RS④, while InO instructions bypass renaming and enter a FIFO queue. The oldest InO instructions are inserted into the RS. Ready instructions execute⑤, write back ⑥, and commit ⑦, though only OoO threads commit, as InO threads execute non-speculatively.

SHADOW is runtime-configurable, with configurations limited by total physical register count. InO threads use integer and floating-point registers, maximizing TLP without renaming or

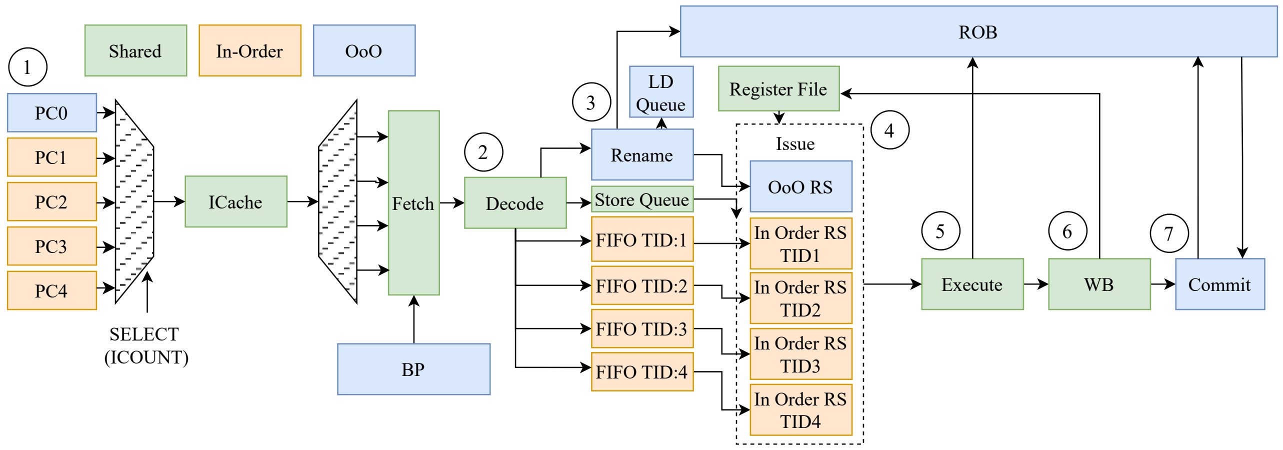

Figure 5: Microarchitecture design of SHADOW configured with 1 OoO and 4 inO threads.

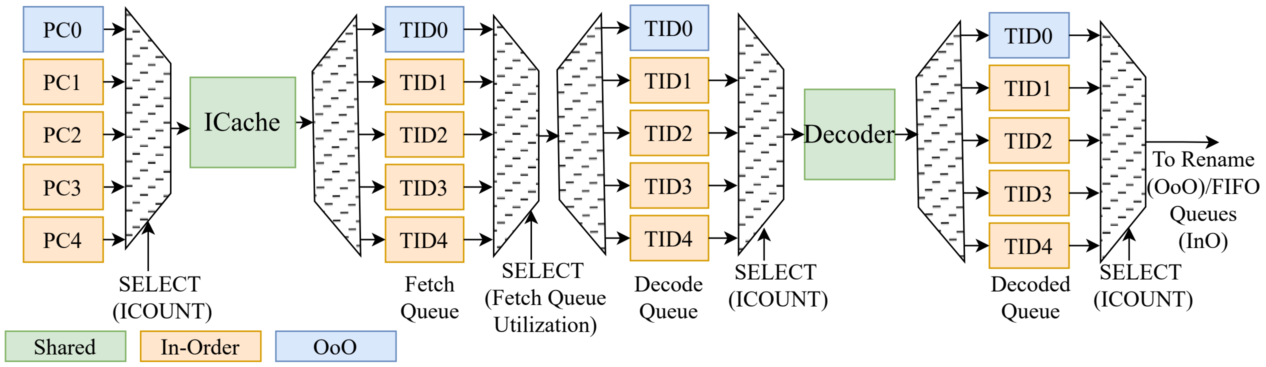

Figure 6: Microarchitecture of the fetch and decode stage configured with 1 OoO and 4 inO threads.

speculation, while OoO threads access all registers, including vector registers. Registers are split between the ARF and the physical register file (PRF). InO threads execute non-speculatively, each receiving a number of registers equal to the ARF. The remaining registers are shared among OoO threads, with architectural state tracking similar to the ARM Cortex-A9 [7], which uses a Register Alias Table to manage the subset of physical registers holding architectural state. With a 256 integer register and 192 floating point register setup modeling the Grace ARM CPU core, SHADOW enables multiple configurations, including 1–3 OoO threads, up to 6 InO threads, and hybrid setups such as 1 OoO + 4 InO and 2 OoO + 2 InO.

SHADOW balances ILP and TLP through a software-based **dynamic work-stealing mechanism** that emerges from a **greedy scheduling strategy** on asymmetric hardware threads. Each thread independently steals work as soon as it becomes available, without centralized control. OoO threads, benefiting from a deep execution window, greedily take more iterations when ILP is high. When memory-bound stalls limit OoO throughput, InO threads, unaffected by speculative bottlenecks, continue stealing work to sustain execution. This adaptive redistribution occurs **without explicit intervention or complex scheduling logic**; instead, the software work-stealing mechanism allows threads to claim available work in a decentralized manner, ensuring efficient workload balancing as an emergent property of execution.

The specific modifications SHADOW introduces at each pipeline stage are described below:

3.1 Fetch and Decode

The fetch ① and decode ② stages in SHADOW operate similarly to SMT-enabled cores (Figure 6). The architecture supports up to six SMT contexts, each with its own program counter (PC). Return Address Stacks (RAS) are exclusive to OoO threads for speculative execution, while InO threads execute without speculation, eliminating rollback mechanisms. Since InO threads do not predict branches, they stall fetch until branch resolution, but SHADOW offsets this with higher TLP from multiple InO threads.

Each cycle, a single thread accesses the instruction cache (ICache), selected via the ICOUNT policy [59]. Decode prioritizes threads with the most instructions in the fetch queue, ensuring a steady supply. Rename and issue selection follow ICOUNT, favoring OoO threads when occupancy is equal to maximize ILP and prevent

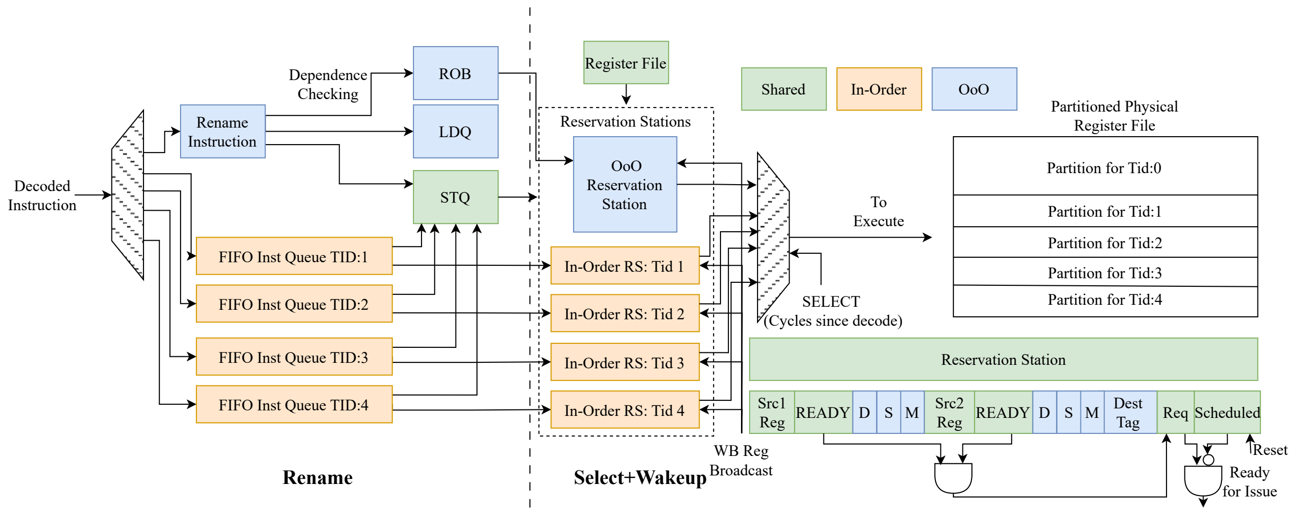

Figure 7: Microarchitecture of the Rename and Wakeup+Select stages configured with SHADOW having 1 OoO and 4 inO threads.

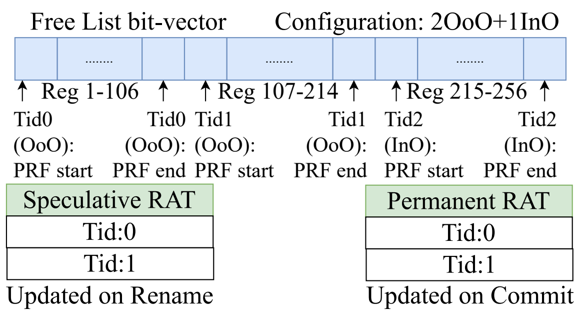

Figure 8: Register File partitioning in SHADOW.

starvation. This scheduling strategy dynamically balances resource allocation for efficient backend utilization.

3.2 Rename

Instruction Placement in the ROB. Figure 7 details SHADOW's rename ③, select, and wakeup ④ stages. After decoding, OoO instructions are renamed to eliminate false dependencies, checked for remaining dependencies, and placed in the ROB and RS, statically partitioned when multiple OoO threads are active to prevent contention. InO instructions bypass renaming, entering per-thread circular FIFO queues that power down when inactive to save energy.

Instruction Placement in the RS. After renaming (for OoO threads) or static PRF mapping (for InO threads), instructions enter the RS. At runtime, the RS is partitioned between OoO and InO threads, with each InO thread allocated one entry, while the rest are evenly split among OoO threads. The oldest InO instructions check dependencies via a per-thread scoreboard, a multi-ported 448-bit table supporting two reads and two writes per thread per cycle. Dependencies are tracked on source registers, and if present, the RS entry waits until sources are ready.

3.3 Context Switching and Register File Allocation

SHADOW uses a software "delegate thread" to apply asymmetric-SMT settings transparently at pthread spawn or context switch [4]. The delegate issues the new `shdw_cfg <#OoO>,<#InO>` instruction, which (1) **Resets the Front End**, activating `<#OoO>` OoO and `<#InO>` InO PCs; (2) **Reconfigures Rename Tables**, assigning OoO threads to the first `<#OoO>` hardware threads and InO threads to the remaining ones, with each OoO thread maintaining an independent free list of physical registers (PRs); and (3) **Partitions Execution Resources**, configuring the ROB and RS based on Section 3.2.

The OS stores an application's thread configuration alongside its process state in the process control block (PCB). To avoid complex resource partitioning across concurrent workloads, SHADOW currently supports either (a) several single-threaded applications or (b) one multi-threaded application. Supporting multiple multi-threaded applications would force the OS to divide shared structures (register file entries, LSQ entries, etc.) among different asymmetric configurations, complicating scheduling and context-switch logic. We leave full multi-program, multi-threaded support for future work.

Register file allocation. Similar to GPUs, which assign registers per thread to manage warp distribution [14], SHADOW dynamically allocates registers at context switches based on the application's configuration. As shown in Figure 8, Register allocation uses a bit-vector free list with a priority decoder, as in BOOM [10]. (1) Each I InO thread is allocated A architectural registers, totaling $A \cdot I$ across all active InO threads. (2) The remaining registers are divided among O OoO threads, with each receiving $(P - A \cdot I)/O$ registers. The free

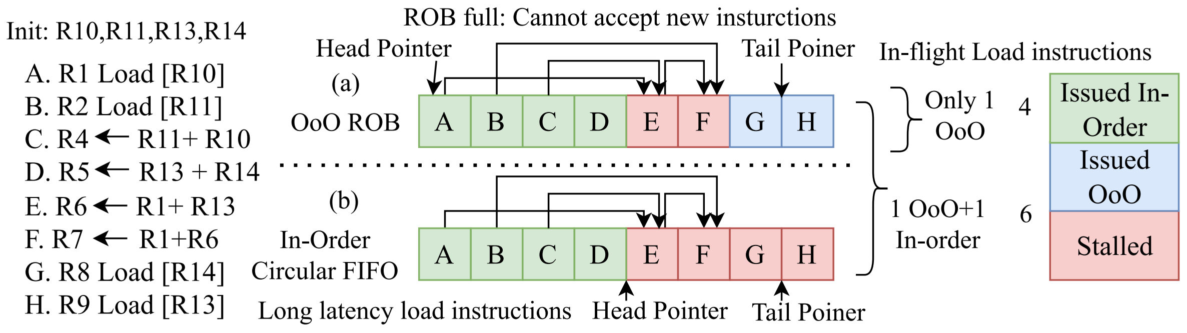

Figure 9: Impact of Adding an In-Order Thread to an OoO System: An Illustrative Assembly Example.

list for each OoO thread is determined using bitwise AND on the available bit vector for the range $[(P-A\cdot I)/O, (i+1)(P-A\cdot I)/O-1]$.

OoO threads use a Pentium 4-style renaming approach [27], similar to MorphCore [56], storing both speculative and architectural data in the PRF. Each OoO thread maintains a per-thread Register Alias Table (RAT) with a Speculative-RAT for speculative execution and a Permanent-RAT for architectural state tracking. This design is preferred over rename checkpoints, as deeper and wider instruction windows (320 entries in this configuration) require more checkpoints—up to 48 for a 256-entry ROB—incurring high RAT restore latency and increased power and performance costs [45]. During context switches, the OS saves the register state and hardware configuration in the PCB, including the stack pointer and Current Program Status Register (CPSR) for each thread before executing the `shdw_cfg` operation.

3.4 Select and Wakeup

The wakeup logic ④ remains unchanged and operates exactly as in a traditional out-of-order core. Figure 7 shows the structure of the RS entry [55, 56].

OoO Wakeup. SHADOW's out-of-order wakeup process follows traditional OoO cores. An operand is marked ready (R bit set) once its dependencies are resolved. For OoO threads, this also occurs when the MATCH bit remains active for the number of cycles specified in the DELAY field. During execution, an instruction broadcasts its destination tag, which is compared against source tags in the RS. A match sets the MATCH bit and updates the DELAY field with the instruction's execution latency, stored in its RS entry. The DELAY value transfers to the SHIFT field associated with the source tag, decrementing each cycle while the MATCH bit stays active. Once the SHIFT field reaches zero, the R bit is set, marking the operand as ready. The RS entry then raises a request for execution once all operands are available.

OoO Select. The select logic scans the RS for the oldest instructions with Req Exec set, generating a Grant bit vector to mark entries for execution. The selected instruction sets its SCHEDULED bit, preventing further competition in later cycles.

In-Order Wakeup. In-order instructions issue sequentially, respecting all dependencies. Each in-order thread has a single RS entry, reducing hardware complexity. If no dependencies exist, the instruction asserts Req to signal readiness; otherwise, it waits until dependencies resolve. When a dependent instruction completes, its destination register is checked against waiting instructions' sources—if matched, the ready bit is set. Execution proceeds once both sources are ready.

This wakeup mechanism is designed for efficiency, eliminating the need for costly broadcasts and tag comparisons across the RS. To maintain correctness, in-order execution stalls and the load instruction entry is not removed from the RS until it has a hit in the translation lookaside buffer (TLB). This precaution prevents an incorrect register state due to potential load re-execution caused by TLB misses.

In-Order Select. With only one RS entry per in-order thread, no additional selection logic is required.

Selecting Ready Instructions for Issue. From the ready instructions in the RS, instructions are selected for issue based on the cycles the instructions have spent in the pipeline since decode, and sent to the issue queues.

3.5 Execute, Writeback, and Commit

SHADOW's execute ⑤, writeback ⑥, and commit ⑦ stages operate like a traditional OoO core. Both OoO and InO threads read from the PRF, execute in the ALU, and use writeback bypass. However, only OoO threads commit, as InO threads execute sequentially without speculation. OoO instructions commit in order, updating the Permanent-RAT.

3.6 Branch Prediction

Only OoO threads perform branch prediction in SHADOW. Experiments show that ILP from the OoO thread and TLP from InO threads effectively mask branch resolution latencies, making branch prediction for InO threads unnecessary. Thus, SHADOW, like the Sun Niagara T1 [34], eliminates branch prediction for InO threads. Mispredictions in the OoO thread affect only its own instructions, leaving InO threads unaffected.

3.7 Load and Store Queues

SHADOW configures load and store queues at runtime. Each OoO thread has a dedicated partition and can execute loads speculatively. Loads update the load queue, check the store queue for forwarding, and enable speculative execution. The store queue detects violations by tracking speculative loads against pending stores.

In contrast, InO threads do not use the load queue, execute conservatively, and each receives five store queue entries. Loads check only the store queue for correctness, avoiding speculation. To prevent incorrect execution, InO threads stall until the issues load registers a TLB hit to avoid incorrect execution if a load needs to be replayed. The TLB is shared among threads.

3.8 Memory Consistency and Coherence

SHADOW targets data-parallel applications that use locks, providing a **sequential consistency for data-race-free (SC for DRF)** model [1]. Lock acquisitions and releases flush relevant pipelines, preventing unintended reordering across lock boundaries. With a shared L1 cache, coherence is maintained without explicit intra-core messaging. While SHADOW currently supports coarse-grained synchronization with SC for DRF, its design can naturally extend to finer-grained memory consistency using ARM-style memory ordering if needed.

3.9 Dynamic Work Distribution and Stealing Mechanism

SHADOW employs a software-based work-stealing mechanism to dynamically distribute workloads between OoO and InO threads. Algorithm 1 demonstrates its implementation using Pthreads. The user specifies `CHUNK_SIZE` to control the granularity of work stolen by each thread (line 2). Worker threads are then spawned (line 3), each executing in a continuous loop (line 4) until all work is completed. A thread acquires the lock (line 6) to safely access and update the global `currentChunk` counter, which tracks the next available workload index. It records its assigned starting index (line 7), updates `currentChunk` so other threads are aware of the remaining work (line 8), and then releases the lock (line 9). If the assigned index exceeds the total workload, the thread terminates (line 11) and waits for others to complete (line 18). Otherwise, it determines the range of iterations it will process (line 13) and executes its assigned work (line 15). The thread then loops back to steal more work (line 5) until no work remains.

The OS scheduler treats each pthread as a normal kernel thread and is unaware of SHADOW's internal OoO/InO asymmetry; it only intervenes on context switches. Fine-grained load balancing happens entirely in user space via work-stealing locks within each pthread, handling uneven progress without OS support. SHADOW also integrates with high-level runtimes (e.g., Intel TBB [50], Cilk [9], OpenMP) that already provide similar work-stealing abstractions.

This evaluation assumes uniformly distributed workloads, where each chunk has the same amount of work. Work stealing adapts to ILP-TLP characteristics: when cache misses are low, OoO threads achieve higher IPC and process more iterations, while at high miss rates, InO threads take on a larger workload, improving resource utilization. Contention on the lock used for work-stealing can arise when multiple threads steal work simultaneously, particularly with small `CHUNK_SIZE` or when many threads finish their assigned work around the same time. In our evaluation, OoO threads stall when no work remains, though implementations could allow them to reclaim unfinished iterations from InO threads for improved efficiency.

Work stealing can affect cache performance, particularly in workloads with high L1 D-cache hit rates, due to increased data movement across threads (Section 5.1). While SHADOW is designed for a single core, its principles extend to multi-core architectures. For example, an ARM big core in a big.LITTLE configuration could be adapted into a SHADOW core, and dynamic work stealing between big and little cores can be used to maximize resource utilization. Mapping OoO and InO threads across multiple SHADOW-enabled cores—e.g., allocating two OoO and four InO threads among many cores—requires complex load balancing under shared-resource constraints and is left for future work.

3.10 Impact on CPU Frequency

SHADOW is expected to have a similar impact on CPU core frequency as MorphCore [56], which estimates a 2.5% slowdown due to added multiplexers in critical pipeline stages. SHADOW introduces comparable multiplexers to support the selection between OoO and InO instructions in the rename stage, RS scheduling, and operand bypass logic. Additionally, SHADOW integrates per-thread fetch queues and in-order wakeup/select logic, similar to MorphCore. Like MorphCore, SHADOW ensures that newly added in-order scheduling logic is placed and routed to avoid extending the critical path beyond these minor multiplexer delays.

When configured with only InO threads, SHADOW does not boost core frequency as InO threads share execution resources with OoO threads, thus changing frequency would require decoupling their pipeline, adding complexity and undermining SHADOW's simplicity. Even with a frequency boost, 6 InO threads would underperform a single OoO thread. Using IBM POWER6 [36] (5 GHz, InO) and POWER7 [21] (4.25 GHz, OoO) as reference points, experiments show that 6 InO threads are 1.08× slower in high L1 D-cache miss workloads and 1.78× slower in ILP-heavy workloads.

3.11 Security Considerations

Sharing core resources among multiple threads can introduce side- and covert-channel vulnerabilities, as demonstrated by extensive prior work on SMT security. Approaches such as SecSMT [57], defense mechanisms against transient-execution attacks on SMT cores [31], and SMT-COP's execution-unit arbitration [58] show how resource partitioning, arbitration, and flush policies can mitigate contention-based leaks. While SHADOW's InO lanes eliminate speculation and rename-table sharing—reducing several common leakage vectors, comprehensive protection (e.g., against subtle microarchitectural or transient attacks) requires integrating dedicated defenses, which is beyond this paper's scope. We leave the exploration of tailored hardware/software co-designs (e.g., fine-grained port throttling, cache partitioning, speculative masking) for future work.

3.12 Example of SHADOW's Impact on TLP and ILP Performance

In OoO execution, long-latency loads with high L2 miss rates stall execution by occupying the ROB for hundreds of cycles, preventing new instructions from entering. Completed instructions cannot retire due to in-order commit, further filling the ROB. Figure 9 uses a small instruction sequence (A–H) to contrast this bottleneck with SHADOW's asymmetric OoO+InO setup, which alleviates ROB pressure.

In Figure 9(a), each OoO instruction allocates a ROB entry, with arrows indicating data dependencies for wake-up, while head and tail pointers track the oldest retire slot and the next allocation position respectively.Long-latency loads retain their ROB entries until they retire; even if younger instructions finish early, their slots remain occupied until all previous entries retire, filling the ROB and blocking new issue due to no free entries.

Figure 9(b) illustrates a SHADOW asymmetric configuration with 1-OoO+1-InO thread: the OoO thread allocates ROB entries with dependency pointers, while the InO thread issues from a lightweight FIFO that bypasses the ROB, allowing the InO thread to utilize execution units alongside the OoO thread without consuming ROB slots. Omitting register renaming and speculation keeps hardware overhead minimal while increasing in-flight memory operations (e.g., from 4 to 6), thereby boosting memory-level parallelism and throughput compared to a lone OoO thread.

Algorithm 1: Dynamic Work Stealing Mechanism in SHADOW

```
Initialize global counter index $currentChunk \leftarrow 0$ and mutex;
Define CHUNK_SIZE as the amount of work each thread steals at a time;
Spawn $N$ worker threads;
foreach worker thread $t$ in parallel do
    while true do
        Lock mutex;
        Set $startChunk \leftarrow currentChunk$;
        Update $currentChunk \leftarrow currentChunk + \texttt{CHUNK\_SIZE}$;
        Unlock mutex;
        if $startChunk \geq TotalWork$ then
            break;
        end
        Set $endChunk \leftarrow \min(startChunk + \texttt{CHUNK\_SIZE}, TotalWork)$;
        foreach iteration $i$ in $[startChunk, endChunk)$ do
            Execute assigned work;
        end
    end
end
Synchronize and join worker threads;
Return completed workload;
```

Table 1: CPU Microarchitectural Parameters

Parameter	Value
ISA	ARM
Simulator	Gem5
Reorder Buffer (ROB) Size and Reservation Station size [13, 16]	320
Circular FIFO Queue Size	20
OoO Thread Store Queue Size	68
OoO Thread Load Queue Size	72
In-Order Thread Store Queue Size	5
Integer Registers (ARF+PRF)	258
Floating Point Registers (ARF+PRF)	192
Vector Registers (ARF+PRF)	192
FUs	6 Int ALUs, 2 Int MUL/DIV, 4 FP/Vector, 2 Load/Store, 1 Load-only, 2 Store-only
L1 Data Cache	64 kB, 2-way set associative, 2 cycles pipelined, 24 MSHRs
L1 Instruction Cache	64 kB, 2-way set associative, 2 cycles pipelined, 12 MSHRs
L2 Cache	1 MB, 16-way set associative, 16 cycles pipelined, 24 MSHRs, Strided Prefetcher
DRAM	DDR4-2400 (16x4), 32 GB/channel, 0.833 ns cycle (1.2 GHz), 17.5 ns row hit, 45.8 ns row miss
Issue Width	8 / 4
Commit Width	8 / 4

Table 2: Benchmarks used to evaluate SHADOW

Benchmark	L1-DCache Miss Rate	L2 Cache Miss Rate	Added lines of code
Sparse Matrix Multiplication	High (>25%)	Varies	-
APSP [5]	24%	1%	0
Backprop [12]	45%	84%	8
Heartwall [12]	71%	39%	8
Tiled Dense Matrix Multiplication	5%	1%	-
BC [12]	1.6%	15%	0
TSP [5]	1%	98%	12
nn [5]	2%	96%	10
Pathfinder [5]	1%	37%	10

4 Evaluation Methodology

This section describes the experimental context used to evaluate SHADOW.

Modeling SHADOW. SHADOW is implemented in Gem5 [8, 41] using system emulation mode, modeling a high-performance ARM Grace core (Table 1), with several modifications to support its asymmetric multithreading model. SMT support was extended to enable heterogeneous execution within a single core, integrating OoO and InO threads. The rename stage was modified to let InO threads bypass register renaming while extending dependency tracking for false dependencies. Dispatch logic enforces strict in-order execution for InO threads, prevents speculation, and prioritizes the oldest ready instruction across all threads. The commit stage was modified so that only OoO threads commit instructions, while InO threads execute strictly non-speculatively, eliminating rollback mechanisms. These changes ensure an accurate SHADOW model in Gem5.

Evaluating SHADOW. SHADOW is evaluated on SpMM (Gustavson's algorithm), tiled dense matrix multiplication, and selected CRONO [5] and Rodinia [12] benchmarks, summarized in Table 2. Benchmarks are classified by L1 D-cache miss rates as high ($\geq 25\%$) or low ($< 25\%$). The table also quantifies code modifications required to integrate dynamic work stealing in Pthreads-based implementations, highlighting adaptation effort.

SpMM serves as the primary workload, with varying sparsity levels showcasing SHADOW's ILP-TLP balancing under different memory access patterns affecting OoO IPC. Additional benchmarks cover compute-bound (e.g., BC, NN, Pathfinder) and memory-intensive (e.g., APSP, Backprop, Heartwall) workloads for broad evaluation. Benchmarks were selected for Pthreads compatibility, with a subset used due to software-based dynamic work stealing requiring well-defined workload distribution. No benchmarks were excluded based on performance.

SHADOW is evaluated against MorphCore and FIFOShelf. MorphCore runs upto 2-thread OoO mode and shifts to up to 8 InO threads once more than two threads are active; the study compares 2-OoO and 6-InO cases, the latter matching the register-file limit. FIFOShelf is roof-lined by modeling 3 OoO threads (the register-constrained maximum) with a doubled ROB dedicated to the OoO path, giving an optimistic upper bound.

Equivalent SHADOW configurations are evaluated for direct performance comparison.

5 Evaluation of SHADOW

This section evaluates SHADOW across memory- and compute-bound workloads under various configurations.

5.1 Performance of SHADOW Across Diverse Workloads

Figure 10 shows that TLP-heavy workloads (Backprop, APSP, TSP) gain most from mixed OoO+InO threads, whereas ILP-centric, cache-sensitive kernels (dense MM, nn, Pathfinder) slow down as extra threads raise cache/RF/ROB pressure. Hence the best thread mix varies with each benchmark's ILP-vs-TLP balance and cache sensitivity; no single uniform policy wins everywhere.

The optimal configuration for each benchmark is influenced by three primary factors: **(1) ILP vs. TLP:** ILP-heavy kernels prefer a few wide OoO threads, whereas TLP-rich kernels benefit from adding lightweight InO threads. **(2) Shared Resources:** Pressure on RF, SQ, and ROB dictates whether additional threads enhance performance or introduce bottlenecks. **(3) Cache Sensitivity:** Cache-bound kernels prefer low-contention setups; too many threads exacerbate cache pressure and negatively impact overall performance.

Table 3 links each benchmark's ILP/TLP balance, resource pressure, and cache behavior to its best setting: mixed OoO + InO excels when both ILP and TLP are available (*Backprop*, *APSP*), whereas cache-bound kernels (*Heartwall*, *nn*) peak with one OoO thread. These mappings show how workload traits direct the configuration that delivers the highest speedup.

5.2 Performance of SHADOW Across Workloads with Varying Cache Miss Rates

Impact of SHADOW Configurations on High-Miss-Rate Workloads. Figure 11 shows that cache-insensitive kernels (APSP, Backprop, Heartwall) favor a mixed design. Beyond two OoO threads, ROB/RF/SQ contention erodes gains; pure InO scaling, in turn, starves ILP. A 1-OoO + 4-InO mix best balances ILP and TLP, raising performance by $1.47\times$ while keeping keeping the resource pressure on ROB, RS, LSQ, RF below the contention threshold.

Impact of SHADOW Configurations on Low-Miss-Rate Workloads. Figure 12 shows SHADOW's speedup over the 1 OoO thread baseline for Dense Matrix Multiplication, BC, TSP, NN, and Pathfinder, compute-bound benchmarks with low L1 DCache miss rates. These workloads favor single-threaded execution, as adding threads increases conflict misses, degrading performance. All configurations perform worse than the baseline due to higher cache contention.

High-Miss-Rate Benchmarks.

- **APSP:** Performance scales with thread count as added TLP complements ILP without harming cache performance. The 2 OoO-2 InO configuration outperforms 3 OoO by avoiding register file contention and store queue bottlenecks.
- **Backprop:** The 1 OoO-3 InO configuration best balances ILP and TLP, maximizing TLP while avoiding structural hazards like rename register shortages or ROB constraints, all while maintaining stable cache performance.
- **Heartwall:** The 1 OoO configuration achieves the best performance. Despite a high L1 D-cache miss rate, L2 misses remain low. Adding more threads significantly increases L2 misses, degrading performance.

Low-Miss-Rate Benchmarks.

- **Tiled Dense Matrix Multiplication, NN, and Pathfinder:** Performance declines as more threads increase cache contention. While TLP improves, conflict misses reduce ILP in the OoO thread. NN and Pathfinder are highly sensitive, with any extra thread causing slowdowns. In Dense Matrix Multiplication, 2 OoO threads achieve 1.2× speedup, balancing TLP gains against ILP loss, but adding more threads degrades performance.
- **BC:** The 2 OoO-1 InO configuration performs best, as BC maintains stable cache miss rates, preserving ILP while leveraging additional TLP.
- **TSP:** The 3 OoO configuration is optimal. While D-cache misses slightly increase with more threads, L2 misses drop significantly, improving ILP as TLP aids L2 prefetching, leading to performance gains.

Configuring SHADOW for Optimal Performance. There is no single best SHADOW configuration for all workloads. The effectiveness of additional in-order threads depends on the ILP-TLP tradeoff of each benchmark. Some workloads benefit from more TLP (e.g., Backprop, APSP), while others rely heavily on ILP and suffer from cache contention when additional threads are added (e.g., Dense MM). The configurability of SHADOW allows users to select the

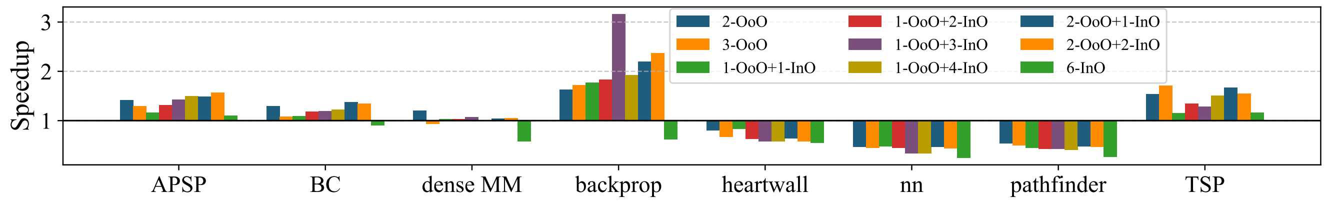

Figure 10: Per-workload speedup for various SHADOW thread configurations.

Benchmark	Best Config	Speedup	Characterization	Rationale for Best Config
TSP [5]	3 OoO	1.7×	TLP-rich	L2 misses drop; TLP helps ILP via prefetching
APSP [5]	2 OoO-2 InO	1.56×	Resource Pressure	Additional threads cause RF and SQ contention
BC [12]	2 OoO-1 InO	1.37×	Resource Pressure	Additional threads cause RF contention
Backprop [12]	1 OoO-3 InO	3.16×	Resource Pressure	Additional threads cause RF, ROB and cache contention
Tiled Dense MM	2 OoO	1.2×	Cache sensitivity	2 OoO balances ILP/TLP; more threads degrade cache
Heartwall [12]	1 OoO	1×	Cache sensitivity	Conflict misses rise with more threads
Pathfinder [5]	1 OoO	1×	Cache sensitivity	Conflict misses rise with more threads
nn [5]	1 OoO	1×	Cache sensitivity	Conflict misses rise with more threads
Average	–	1.36×	–	–

Table 3: Best configuration, speedup, and explanation for each benchmark. OoO = out-of-order, InO = in-order, RF = register file, SQ = store queue, ROB = reorder buffer.

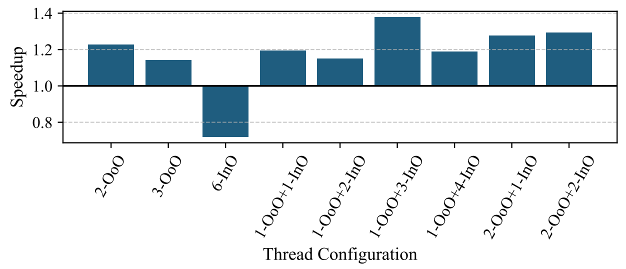

Figure 11: Geometric mean performance of SHADOW configurations on high D-cache miss rate benchmarks from Table 2, normalized to a single-threaded OoO core.

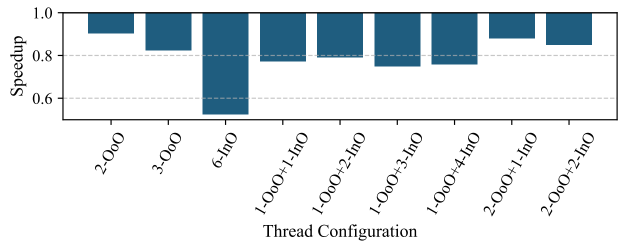

Figure 12: Geometric mean performance of SHADOW configurations on low D-cache miss rate benchmarks from Table 2, normalized to a single-threaded OoO core.

optimal configuration for their specific workload, enabling adaptive performance tuning based on application characteristics.

5.3 Characterization of SHADOW on SpMM

Beyond evaluating SHADOW across diverse workloads, a detailed analysis is conducted on **sparse matrix multiplication (SpMM)**.

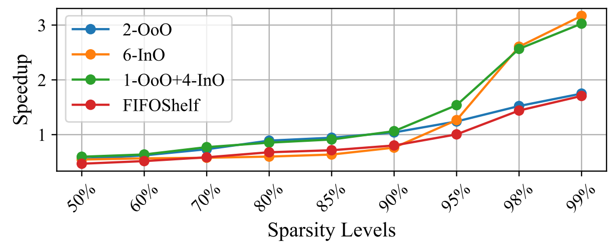

Figure 13: Performance of SHADOW with varying degrees of sparsity for an 8 wide CPU over 1 OoO thread.

SpMM serves as a controlled testbed to assess SHADOW's adaptability across varying cache miss rates by adjusting matrix sparsity and size. While matrices maintain uniform sparsity across rows, individual elements are randomly distributed, disrupting spatial locality and limiting the effectiveness of hardware prefetchers. This results in higher cache miss rates, making SpMM an ideal workload to evaluate SHADOW's ability to balance ILP and TLP under memory constraints. Using Gustavson's algorithm [24], we systematically vary sparsity and matrix size to characterize SHADOW's response to different memory access patterns.

We compare four SHADOW configurations: (1) 1 OoO, (2) 2 OoO, (3) 1 OoO + 4 InO, and (4) 6 InO. This analysis demonstrates how SHADOW dynamically balances ILP and TLP to optimize performance across different sparsity levels.

5.3.1 Varying Degrees of Sparsity. SHADOW's performance was evaluated on SpMM using 600 × 600 matrices with sparsity ranging from 50% to 99% (Figures 13 and 14 for 8- and 4-wide CPUs, respectively). At lower sparsity, adding 2-OoO threads improves performance over a 1-OoO thread. However, as ILP diminishes with increasing sparsity, six InO threads surpass two OoO threads. FIFOShelf achieves performance comparable to a 2-OoO-thread

Table 4: Hardware overhead of SHADOW

HW Structure	Components	Quantity / Size
Thread Control State Registers	General-purpose registers, Link register, Program counter and Current Program Status Register	68 Registers
Thread Control Block	Thread State Information, Thread Local Storage (TLS) Pointer, Stack Pointer, Thread ID, Priority Information, Pointers to resources	1664 Bits
Circular FIFO Queues	20 Entries per thread	5120 Bits
Multiplexers and Demultiplexers	Added in Fetch, Decode, Rename and Issue stages	4
Fetch Queues	Queue to store 8 fetched instructions	2048 Bits
Scoreboard	Scoreboard with six multi-ported tables, one dedicated per thread	2688 Bits

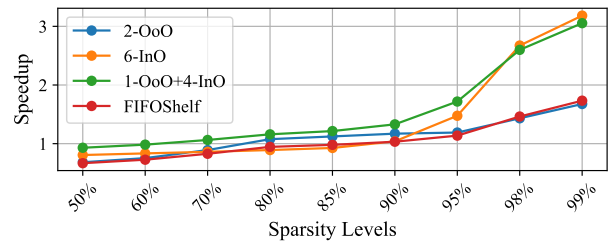

Figure 14: Performance of SHADOW with varying degrees of sparsity for a 4 wide CPU over 1 OoO thread.

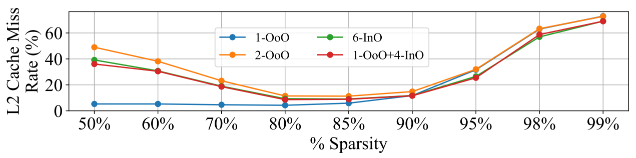

Figure 15: Change in miss rate for different thread configurations.

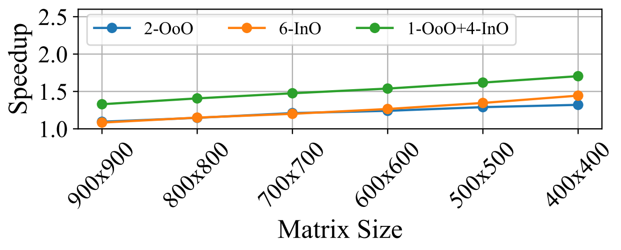

Figure 16: Performance of SHADOW with varying matrix size for 95% sparsity for an 8 wide CPU over 1 OoO thread.

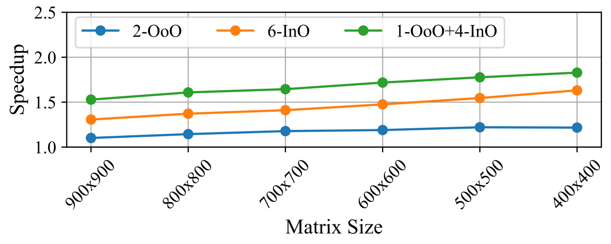

Figure 17: Performance of SHADOW with varying matrix size for 90% sparsity for a 4 wide CPU over 1 OoO thread.

configuration. Its instruction streams still contend for shared structures—most critically the register file (for renaming) and the load/store queues, which limits its overall gain. SHADOW's 1 OoO + 4 InO configuration achieves the highest performance up to 95%, effectively balancing ILP and TLP.

On the 8-wide core, 1 OoO + 4 InO peaks at 3.1 × speed-up (99% sparse) and averages 1.33 × over the single-OoO baseline. Relative to 2 OoO it delivers 1.72 × (avg. 1.20 ×); to 6 InO, 1.43 × (avg. 1.20 ×); and over FIFOShelf, 1.78 × (avg. 1.40 ×). Its lead over pure OoO widens as sparsity—and thus TLP—rises, while its edge over InO-only designs is greatest at lower sparsity, where the OoO thread can still harvest ILP. TThe 4-wide core follows suit, also peaking at 3.1 × and averaging 1.56 ×.

Figure 18 shows IPC trends as sparsity increases. At 70% sparsity, 1 OoO thread achieves 4.0 IPC; adding additional threads reduces IPC due to higher L2 conflict misses. At 85% sparsity, OoO-only IPC falls further, yet the 1 OoO + 4 InO asymmetric configuration maintains an edge at 3.4 IPC, nearly matching the single-OoO's 3.7 IPC. Beyond 95% sparsity, 1 OoO + 4 InO best balances ILP and TLP, delivering the highest IPC of 1.4 among all configurations.

5.3.2 L2 Miss Rate Across Various Thread Configurations. Figure 15 shows the L2 cache miss rate for different configurations: a single OoO thread, two OoO threads, six InO threads, and SHADOW (1 OoO + 4 InO). At 50% sparsity, the L2 miss rate is low (5%) but increases with additional threads. At 99% sparsity, it peaks at 73% for a single OoO thread due to the irregular access pattern. Despite a 99% sparse 600×600 matrix containing only 7200 elements, high L2 misses occur as the result matrix must be loaded from DRAM, the extreme sparsity prevents effective prefetching, limiting cache efficiency. Overall, while more threads lead to a small rise in L2 misses, the impact remains limited and does not significantly degrade cache performance across evaluated configurations.

When modeling a more aggressive L2 cache Access-Map Pattern Matching prefetcher [30], the relative SMT performance remains unchanged. On a 98%-sparse 600×600 SpMM, L2 misses drop from 63% to 32% with this prefetcher, yet an 8-wide CPU still achieves a 1.19× speedup with 2-OoO threads, 1.94× with 6-InO threads, and 1.93× with the 1-OoO + 4-InO asymmetric SMT.

Adding a 256 MB L3 cache did not alter the L2 miss rate for a 95%-sparse 600×600 SpMM, but it did increase the average DRAM access latency. As a result, relative to a single OoO core, two OoO threads now achieve a 1.35× speedup, six InO threads reach 1.76×, and the 1 OoO + 4 InO asymmetric configuration remains the highest performing at 1.95×.

5.3.3 Varying Matrix Sizes. Figures 16 (8-wide CPU) and 17 (4-wide CPU) show SHADOW's performance on 95% sparse SpMM across different matrix sizes. Across both CPU widths, SHADOW (1 OoO + 4 InO) outperforms 2 OoO and 6 InO configurations, demonstrating

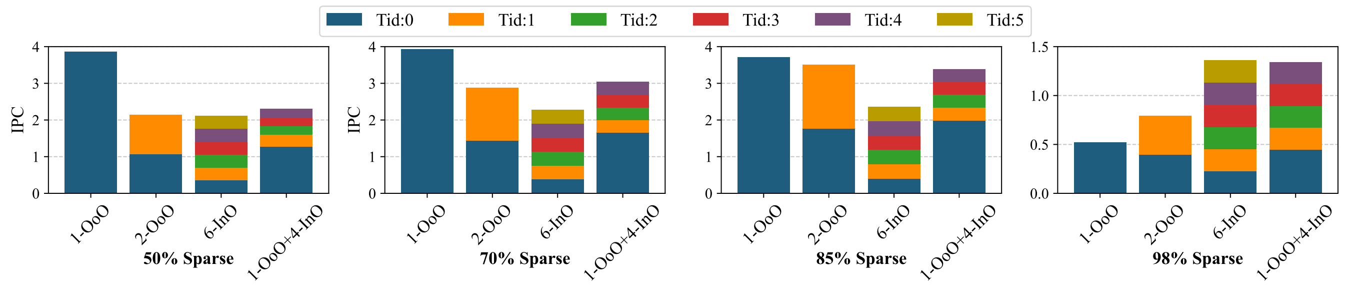

Figure 18: Breakdown of IPC contribution from each thread with varying degrees of sparsity over 1 OoO thread.

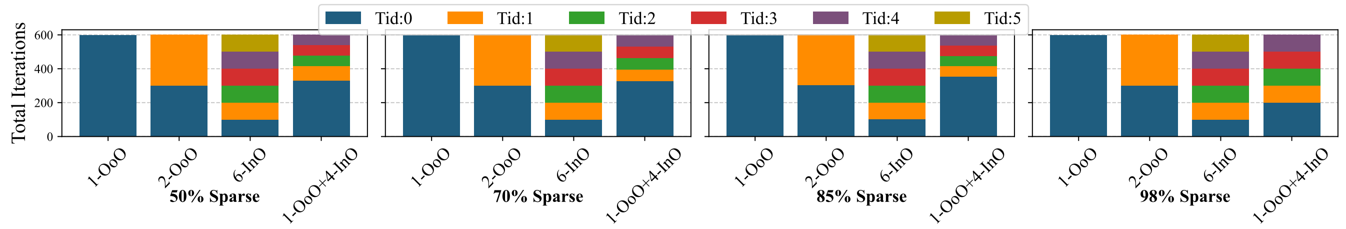

Figure 19: Distribution of work across threads with dynamic work stealing for various degrees of sparsity.

the effectiveness of its asymmetric multithreading. On an 8-wide CPU, it achieves up to 1.7× speedup over a single OoO thread, while on a 4-wide CPU, it reaches 1.82×, efficiently utilizing resources across varying matrix sizes.

5.3.4 Benefit of Dynamic Work Stealing Across Threads. SHADOW dynamically balances workload distribution through work stealing (Algorithm 1). Threads acquire more work upon completing assigned chunks, with the OoO thread taking a larger share when ILP is high, and distribution equalizing as ILP decreases. Figure 19 shows how SHADOW adjusts workload allocation with a CHUNK_SIZE of 1 based on IPC. At 50%, 70% sparsity and 85%, the OoO thread's high IPC allows it to process more iterations. At 98% sparsity, declining IPC shifts more work to in-order threads, leading to an even distribution. By adapting to workload characteristics, SHADOW effectively balances ILP and TLP, optimizing parallelism while simplifying workload management for programmers.

Varying CHUNK_SIZE across 1, 5, 10, 25, 50 for a 95% sparse matrix shows that OoO-only and InO-only configurations are insensitive to chunk size, while the 1 OoO + 4 InO setup is modestly affected: its speedup stays at 1.53× for CHUNK_SIZE = 1, falls slightly to 1.51× at 25, and drops to 1.25× at 50.

5.4 Area and Power Overhead of SHADOW

Table 4 summarizes SHADOW's hardware overhead. Despite supporting multiple threads, the PRF remains unchanged, with each thread adding minimal state via Thread Control State Registers and a Thread Control Block (TCB). Pipeline bandwidth is unaffected, as SHADOW employs per-thread fetch queues with simple multiplexers/demultiplexers. Unlike OoO threads, InO threads bypass renaming and ROB allocation, relying instead on lightweight FIFO queues. The reservation station size, CAM complexity, and ROB remain unchanged. Using a modified McPAT [38], we estimate SHADOW's area and power overhead to be just 1% over a high-performance core.

6 Related Work

Prior work has explored various techniques to improve ILP and TLP on CPUs. This section summarizes key approaches and contrasts them with SHADOW's design.

SMT enhances TLP by allowing multiple threads to share execution resources [19, 28, 59, 60, 64]. Traditional SMT treats all threads as microarchitecturally identical, limiting its ability to adapt to workload characteristics. Prior efforts introduced thread prioritization and resource partitioning [18, 43, 54], but these still rely on symmetric thread execution. SHADOW breaks this limitation by integrating OoO threads with multiple lightweight InO threads, enabling dynamic ILP-TLP balancing.

Prior heterogeneous cores (TRIPS EDGE [52], Tilera TILE-Gx [49], Shasta/VISC [61]) rely on a new ISA, many replicated tiles, or speculative core fusion. SHADOW instead keeps the legacy ISA and runs OoO and lightweight InO threads side-by-side in one conventional pipeline, blending ILP + TLP with only 1% extra logic. SSMT [11] improves single-threaded performance by spawning auxiliary microthreads to optimize branch prediction and cache behavior. While these threads assist execution, they remain symmetric OoO threads. SHADOW differs by using microarchitecturally asymmetric threads, leveraging both deep ILP in OoO execution and wider TLP through InO threads to improve multi-threaded performance. Other approaches have reconfigured multi-core architectures to improve TLP, combining simple cores for higher ILP when needed [23, 29, 32, 63]. Unlike these inter-core techniques, SHADOW enhances TLP within a single core, retaining the full performance of an OoO core with only 1% area overhead. Similarly, prior work has proposed migrating applications across heterogeneous cores to optimize utilization[2, 3, 35, 44], while SHADOW improves intra-core parallelism without requiring core migration.

SHADOW achieves these benefits without increasing CPU area overhead, modifying the ISA, or altering the programming model. This preserves general-purpose flexibility while improving TLP and maintaining single-threaded performance.

7 Conclusion

SHADOW introduces an asymmetric SMT architecture that balances ILP and TLP by running OoO and InO threads concurrently. Unlike traditional SMT, it dynamically adapts to workload demands, leveraging deep ILP in the OoO thread and high TLP in lightweight InO threads. SHADOW is runtime-configurable, optimizing execution based on workload characteristics. Across nine diverse benchmarks SHADOW achieves up to 3.1× speedup and 1.33× average improvement over an OoO core, with just 1% area and power overhead.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (1996), 66–76. https://doi.org/10.1109/2.546611

[2] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. 1995. The MIT Alewife machine: Architecture and performance. *ACM SIGARCH Computer Architecture News* 23, 2 (1995), 2–13.

[3] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. 1993. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE micro* 13, 3 (1993), 48–61.

[4] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. 2013. ReconOS: An operating system approach for reconfigurable computing. *IEEE Micro* 34, 1 (2013), 60–71.

[5] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 44–55.

[6] Mehdi Alipour, Rakesh Kumar, Stefanos Kaxiras, and David Black-Schaffer. 2019. Fiforder microarchitecture: Ready-aware instruction scheduling for ooo processors. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 716–721.

[7] Arm Ltd. 2010. *Cortex-A9 Technical Reference Manual: Register Renaming*. https://developer.arm.com/documentation/ddi0388/h/Functional-Description/About-the-functions/Register-renaming Accessed: 2025-02-20.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 207–216. https://doi.org/10.1145/209937.209958

[10] Christopher Celio, David A. Patterson, and Krste Asanović. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Technical Report UCB/EECS-2015-167. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[11] Robert S Chappell, Jared Stark, Sangwook P Kim, Steven K Reinhardt, and Yale N Patt. 1999. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th annual international symposium on Computer architecture*. 186–195.

[12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[13] Wikipedia contributors. 2025. ARM Neoverse. https://en.wikipedia.org/wiki/ARM_Neoverse Accessed: 2025-02-21.

[14] Brett W. Coon, John Erik Lindholm, Gary Tarolli, Svetoslav D. Tzvetkov, John R. Nickolls, and Ming Y. Siu. 2009. Register File Allocation. https://patents.google.com/patent/US7634621B1/en

[15] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 1–20.

[16] darchr. 2025. Grace Out-of-Order CPU Implementation. https://github.com/darchr/novoverse/blob/main/components/processors/grace/gracecore/grace_o3_cpu.py Accessed: 2025-02-21.

[17] Kaushik Datta and et al. 2008. Stencil Computation Optimization and Auto-tuning on Modern Microprocessors. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (2008).

[18] Gautham K Dorai and Donald Yeung. 2002. Transparent threads: Resource sharing in SMT processors for high single-thread performance. In *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 30–41.

[19] Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. 1997. Simultaneous multithreading: A platform for next-generation processors. *IEEE micro* 17, 5 (1997), 12–19.

[20] Adi Fuchs and David Wentzlaff. 2019. The accelerator wall: Limits of chip specialization. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–14.

[21] Nigel Griffiths. 2019. POWER CPU Memory Affinity 3 - Scheduling Processes to SMT and Virtual Processors. https://www.ibm.com/support/pages/power-cpu-memory-affinity-3-scheduling-processes-smt-and-virtual-processors. Accessed: 2025-02-19.

[22] Dheevatsa Gupta and et al. 2020. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *arXiv preprint arXiv:2008.07678* (2020).

[23] Shantanu Gupta, Shuguang Feng, Amin Ansari, and Scott Mahlke. 2010. Erasing core boundaries for robust and configurable performance. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 325–336.

[24] Fred G Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (1978), 250–269.

[25] Juan Gómez-Luna, Kaan Sari, Animesh Das, Saher Abulila, Sayantan Ghose, Rachata Ausavarungnirun, and Onur Mutlu. 2022. Evaluating Machine Learning Workloads on Memory-Centric Computing Systems. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–12. https://doi.org/10.1109/IISWC55726.2022.00012

[26] Sébastien Hily and André Seznec. 1999. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. IEEE, 64–67.

[27] Glenn Hinton. 2001. The microarchitecture of the Pentium 4 processor. *Intel technology journal* 5 (2001), 1–13.

[28] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. 1992. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th annual international symposium on Computer architecture*. 136–145.

[29] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. 2007. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*. 186–197.

[30] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2011. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism* 13, 2011 (2011), 1–24.

[31] Xin Jin and Ningmei Yu. 2021. A defense mechanism against transient execution attacks on SMT processors. *IEICE Electronics Express* (2021), 18–20210041.

[32] Changkyu Kim, Simha Sethumadhavan, Madhu S Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W Keckler. 2007. Composable lightweight processors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 381–394.

[33] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. 2018. Cimple: Instruction and Memory Level Parallelism. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 1–14. https://doi.org/10.1145/3243176.3243185

[34] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: A 32-way multithreaded sparc processor. *IEEE micro* 25, 2 (2005), 21–29.

[35] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. 2003. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 81–92. https://doi.org/10.1109/MICRO.2003.1253185

[36] Hung Q Le, William J Starke, J Stephen Fields, Francis P O'Connell, Dung Q Nguyen, Bruce J Ronchetti, Wolfram M Sauer, Eric M Schwarz, and Michael T Vaden. 2007. Ibm power6 microarchitecture. *IBM Journal of Research and Development* 51, 6 (2007), 639–662.

[37] Jure Leskovec, Anand Rajaraman, and Jeffrey D Ullman. 2014. *Mining of Massive Datasets*. Cambridge University Press.

[38] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*. 469–480.

[39] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. 1997. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems (TOCS)* 15, 3 (1997), 322–354. https://doi.org/10.1145/263326.263382

[40] Jason Loew and Dmitry Ponomarev. 2008. Two-level reorder buffers: accelerating memory-bound applications on SMT architectures. In *2008 37th International Conference on Parallel Processing*. IEEE, 182–189.

[41] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[42] Hadi Makrani, Sai Manoj Pudukotai Dinakarrao, Avesta Sasan, and Houman Homayoun. 2018. A Comprehensive Memory Analysis of Data Intensive Workloads on Server Class Architecture. In *Proceedings of the International Symposium on Memory Systems*. ACM, 125–136. https://doi.org/10.1145/3240302.3240319

[43] Artemiy Margaritov, Siddharth Gupta, Rekai Gonzalez-Alberquilla, and Boris Grot. 2019. Stretch: Balancing qos and throughput for colocated server workloads on smt cores. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 15–27.

[44] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F Wenisch. 2019. Enhancing server efficiency in the face of killer microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 185–198.

[45] Andreas Moshovos. 2003. Checkpointing alternatives for high performance, power-aware processors. In *Proceedings of the 2003 international symposium on Low power electronics and design*. 318–321.

[46] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[47] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.

[48] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems*. McGraw-Hill.

[49] Carl Ramey. 2011. Tile-gx100 manycore processor: Acceleration interfaces and architecture. In *2011 IEEE Hot Chips 23 Symposium (HCS)*. IEEE, 1–21.

[50] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media. https://dl.acm.org/doi/10.5555/1352079.1352134

[51] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics.

[52] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*. 422–433.

[53] Faissal M Sleiman and Thomas F Wenisch. 2016. Efficiently scaling out-of-order cores for simultaneous multithreading. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 431–443.

[54] Allan Snavely and Dean M. Tullsen. 2000. Symbiotic Job Scheduling for a Simultaneous Multithreading Processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, 234–244. https://doi.org/10.1145/378993.379244

[55] Jared Stark, Mary D Brown, and Yale N Patt. 2000. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 57–66.

[56] M Aater Suleman, Milad Hashemi, Chris Wilkerson, Yale N Patt, et al. 2012. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 305–316.

[57] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. 2022. {SecSMT}: Securing {SMT} processors against {Contention-Based} covert channels. In *31st USENIX Security Symposium (USENIX Security 22)*. 3165–3182.

[58] Daniel Townley and Dmitry Ponomarev. 2019. Smt-cop: Defeating side-channel attacks on execution units in smt processors. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 43–54.

[59] Dean M Tullsen, Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, and Rebecca L Stamm. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd annual international symposium on Computer architecture*. 191–202.

[60] Dean M Tullsen, Susan J Eggers, and Henry M Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*. 392–403.

[61] Jim Turley. 2014. VISC Processor Secrets Revealed. EE Journal. https://www.eejournal.com/article/20141203-softmachines2/ Online; accessed 2025-06-13.

[62] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 97–110.

[63] Yasuko Watanabe, John D Davis, and David A Wood. 2010. WiDGET: Wisconsin decoupled grid execution tiles. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 2–13.

[64] Wayne Yamamoto, Mauricio J Serrano, Adam R Talcott, Roger C Wood, and M Nemirosky. 1994. Performance estimation of multistreamed, superscalar processors. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, Vol. 1. IEEE, 195–204.

[65] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2024. Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 931–945.