Hybrid Latency Tolerance for Robust Energy-Efficiency on 1000-Core Data Parallel Processors

Neal C. Crago
Intel Corporation
neal.c.crago@intel.com

Omid Azizi
HiCAMP Systems, Inc
oazizi@hicampsystems.com

Steven S. Lumetta and Sanjay J. Patel
University of Illinois
{lumetta, sjp}@illinois.edu

Abstract

Currently, GPUs and data parallel processors leverage latency tolerance techniques such as multithreading and prefetching to maximize performance per Watt. However, choosing a technique that provides energy-efficiency on a wide variety of workloads is difficult, as the type of latency to tolerate, required hardware complexity, and energy consumption is directly related to application behavior. After qualitatively evaluating five commonly used latency tolerance techniques, we develop a hybrid technique utilizing multithreading and decoupled execution to maximize performance while minimizing hardware complexity and energy consumption across a wide variety of workloads.

We compare our hybrid technique with the five commonly used techniques on a 1024-core data parallel processor by performing a comprehensive design space exploration, leveraging detailed performance and physical design models. By intelligently leveraging both decoupled execution and multithreading, our hybrid latency tolerance technique is able to improve energy-efficiency by 28% to 89% over any single technique on data parallel benchmarks. Compared to other combinations of latency tolerance techniques, we find that our hybrid latency tolerance technique provides the highest energy-efficiency by over 26%.

1 Introduction

Energy efficiency is a growing concern in the field of computer architecture. While the number of transistors is expected to continue to scale with Moore's law for at least the next five years, the "Power wall" has brought the end of single-core processor performance scaling. As a result, improving throughput performance has become a main focus of the community and current GPU and data parallel processors utilize tens to hundreds of simple cores on a single chip to maximize performance per Watt [14].

To maximize the throughput performance of GPU and data parallel processors, latency tolerance techniques are implemented to keep functional units busy. However, while memory and functional unit latency tolerance has been an active area of research, application behavior drastically affects the performance and energy consumption of a given technique, making it less clear which technique should be selected. Current data-parallel processors such as GPUs typically focus on multithreading, which has proven to be an effective way to improve throughput performance. However, GPUs and data parallel processors are now adding caches to improve programmability and can suffer the effects of cache contention, a significant performance and energy pitfall. Overall, to reach the energy efficiency goals of future 1000-core data-parallel processors, the choice of latency tolerance technique needs to be revisited.

In this work, we evaluate the energy efficiency and suitability of hardware prefetching, out-of-order execution, multithreading, hardware scout prefetching, and decoupled execution for 1000-core data parallel processors. After qualitatively discussing the functionality and tradeoffs of each technique, we propose a hybrid technique that combines multithreading and decoupled execution. While multithreading and decoupled execution in isolation have performance pitfalls on different common code patterns, these pitfalls can be avoided when the techniques are combined to provide robust energy efficiency across a wide variety of workloads while minimizing hardware complexity. Our hybrid technique focuses primarily on decoupled execution to provide latency tolerance, and falls back to multithreaded execution when decoupling is not useful.

For the purpose of our evaluation, we developed a comprehensive design space exploration framework with high-fidelity performance and physical design models that allow fair comparison of our proposed hybrid technique with the five commonly used techniques. To further support the fidelity and fairness of the design space exploration, energy-efficient and complexity-effective versions of each latency tolerance technique are implemented. The development of the detailed models for each latency tolerance technique is a significant contribution of this work, and leverages a cycle-accurate 1000-core simulator and RTL synthesis and ana-

978-1-4673-5587-2/13/$31.00 ©2013 IEEE

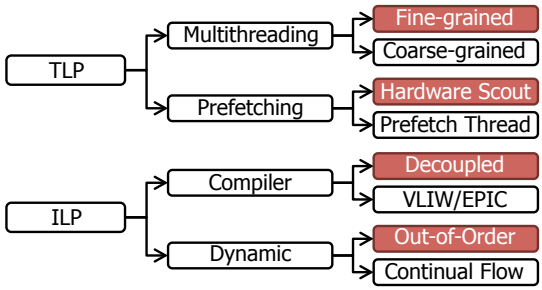


Figure 1: Categorization of latency tolerance techniques. Highlighted techniques are considered in this work.

lytical SRAM models for a 45nm CMOS manufacturing process. Additionally, microbenchmarks are developed to illustrate important code patterns for 1000-core data parallel processors and evaluate each latency tolerance technique on isolated application behavior.

We find our hybrid latency tolerance technique avoids the pitfalls of other techniques and can provide 28% to 89% better energy efficiency than a single technique alone. Out-of-order, multithreading and decoupled techniques provide similar levels of energy efficiency on average across data parallel benchmarks. While decoupled execution and multithreading have significantly lower hardware complexity than out-of-order, energy pitfalls due to application behavior limit their energy efficiency on average. Even when considering other possible combinations of hardware prefetching, out-of-order execution, multithreading, hardware scout, and decoupled execution, we find that our hybrid latency tolerance technique provide the highest energy efficiency by over 26%.

2 Goals and Latency Tolerance Techniques

The peak throughput performance of 1000-core data parallel processors is fundamentally limited by chip area and power budgets. An appropriate latency tolerance technique maximizes performance improvement while minimizing additional cost. Specifically, added hardware complexity, energy consumption, and applicability to a variety of workloads must be considered.

We compare five latency tolerance techniques for future 1000-core data-parallel processors while advocating for the hybrid technique we propose. As the baseline, we evaluate the ability of hardware prefetching to tolerate memory latency. Figure 1 shows that we compare hardware prefetching to a wide range of techniques including instruction-level parallelism (ILP) techniques that are dynamic and leverage the compiler, and thread-level parallelism (TLP) techniques that provide dynamic prefetching or execute another thread of execution. For each technique, we discuss the high-level operation, applicability for latency tolerance, and concerns for implementation on 1000-core data parallel processors.

Hardware Prefetching: Prefetching data into the caches can help tolerate memory latency, which can otherwise dominate memory-intensive data parallel applications. To minimize hardware complexity, we consider next-line and stride-based prefetching at the core level, which prior research has shown to perform well on GPUs and manycore [5, 13]. When the prefetch tables are sized appropriately for data-parallel workloads, both of these approaches require a relatively low amount of additional hardware. The largest concern with prefetching is inaccurately predicting the memory access stream, and prefetching the data either too early or too late. If an application's memory-stream is irregular or unpredictable, cache pollution can occur and increase the number of cache accesses and energy consumption. Additionally, prefetching can only provide memory latency tolerance, and cannot tolerate functional unit latency.

Out-of-Order Execution: Out-of-order (OOO) cores improve performance by leveraging hardware structures to dynamically execute instructions in a single thread out-of-order. These additional hardware structures include large physical register files, reorder buffers, and register renaming. When sized appropriately, functional unit latency and even memory latency can be tolerated by dynamically executing non-blocked instructions in the instruction window. The extra hardware complexity required for out-of-order execution can be significant, both in terms of chip area and energy consumption. If the application experiences long memory access latencies, the additional hardware complexity needed to increase the number of inflight instructions to tolerate the latency can be prohibitive.

Multithreading: In contemporary data parallel processors, multithreading is often used to interleave multiple in-order threads of execution and provide both memory and functional unit latency tolerance [14, 21]. The hardware complexity for multithreading can be relatively low, and generally requires additional register files and scratch space to be added for each thread. A significant concern with multithreading is cache contention due to application behavior and shared caches. If the active data set for each thread is too large or the memory addresses do not map into the cache well, contention between threads can occur, causing performance degradation and increased energy consumption.

Hardware Scout Prefetching: Hardware scout, also known as runahead execution, dynamically generates prefetches to tolerate memory latency [4, 7]. The hardware scout thread operates as a separate thread of execution which speculatively preexecutes the main thread to generate prefetches while the core is otherwise stalled during cache misses. The potential benefit of the hardware scout style of prefetching is improved prefetch accuracy and timeliness.

However, preexecuting the instruction stream requires significant extra energy consumption, as each instruction preexecuted requires additional energy to be spent in instruction fetch, the register file, and accessing the caches. Similarly to other prefetching techniques, hardware scout cannot be used to tolerate functional unit latency.

Decoupled Execution: Recently, there has been renewed interest in decoupled execution for improving performance in GPUs and manycore [6, 1]. Decoupled execution leverages the compiler to partition a single thread of execution into separate *memory-accessing* and *memory-consuming* instruction streams called `strands`, which communicate data and control flow decisions with one another through FIFO data queues [22]. Decoupled execution provides significant memory latency tolerance, as instead of stalling when a cache miss occurs, the memory-accessing strand can execute ahead, executing independent instructions in a similar manner to OOO execution. However, decoupled architectures have significantly less complex hardware due to placing more complexity in the compiler.

While recent work has presented solutions to performance limitations, such as memory indirection and functional unit latency tolerance [6], the lack of ability to tolerate *data-dependent control flow* remains a concern. Data-dependent control flow occurs when the memory-consuming strand is responsible for determining the next basic block, which causes the memory-accessing strand to not be able to execute ahead and stall. If an application has frequent occurrences of data-dependent control flow, these stalls can severely limit the potential performance benefit of decoupled execution. Additionally, the instruction overhead for maintaining separate fetching and executing instruction streams leads to additional energy consumption.

3 Hybrid Latency Tolerance

An ideal latency tolerance technique has low hardware cost and provides robust energy efficiency across a wide variety of workloads. However, each of the commonly used techniques either has high complexity or has a performance or energy pitfall under certain application behavior. To move closer to a robust technique, different techniques could be combined together. However, combination must be performed in a thoughtful manner, as naively combining techniques together can result in significant additional hardware complexity and little benefit.

To minimize additional hardware overhead and energy consumption, we propose combining multithreading and decoupled execution. The major differences in applicability across workloads enables such a combination to provide robust performance on large variety of workloads. Additionally, both techniques provide both memory and functional unit latency tolerance and require similarly low hardware requirements. As a result, we can avoid the significant hardware complexity of out-of-order execution, as well as the speculation waste and lack of functional unit latency tolerance of both hardware prefetching and hardware scout.

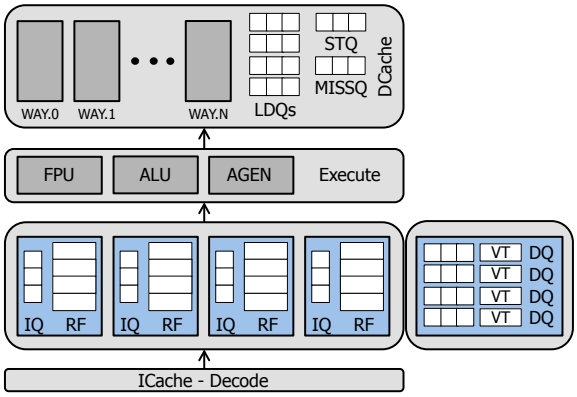


Figure 2: Proposed core architecture. Each context is configurable as a decoupled strand or full thread of execution

3.1 Intelligently Combining Techniques

The straightforward approach in combining multithreading and decoupled execution would be to architect the hardware to support both simultaneously. To achieve the full combination of a four-strand decoupled and four-way multithreaded core, support for sixteen strand contexts including register files, instruction queues, and data queues must be added, resulting in a large and possibly prohibitive increase in hardware complexity.

Instead of naively combining these two techniques, we consider the use case for such a hybrid with the goal of intelligently combining multithreading and decoupled execution. In general, each technique can operate efficiently on its own, with the exception of workloads with data dependent control flow and those that experience cache contention. Another interesting observation is that these two different techniques use similar hardware in the form of strand and thread contexts. These observations lead us to propose a hybrid technique with the ability to switch between modes of execution. In doing so, hardware can be used efficiently while enabling robustness on a larger variety of workloads.

3.2 Architecture and Execution Modes

Figure 2 presents our low-complexity core architecture used to implement hybrid latency tolerance. A total of four contexts can be supported, each of which can either be a strand context from decoupled execution or a hardware thread context. Unlike prior decoupled architecture proposals [22], all strands interleave their execution on a single

pipeline. To facilitate strand communication during decoupled execution, data queues are provided using the virtualization technique described in [6]. Finally, each context has support for issuing multiple non-blocking load instructions.

The architecture supports three execution modes:

- **Decoupled Mode** (1 thread of up to 4 strands)
- **Multithreaded Mode** (4 threads of 1 strand each)
- **MT-Decoupled Mode** (2 threads with 2 strands each)

3.3 Selection of Execution Mode

The execution mode for a data parallel task is selected based upon application behavior to improve performance and to avoid energy waste. This paper focuses on automatic static selection, which we find performs within 8% on average of oracular static selection. While dynamic selection with runtime and operating system support is left for future work, we believe the potential benefit of implementing it to be relatively low. Our selection scheme primarily focuses on leveraging decoupled execution, and then falling back to multithreading when data-dependent control flow exists and decoupling is not useful.

The compiler uses the following selection algorithm:

- If data-dependent control flow exists in the inner loop of a task, multithreaded execution mode is chosen.
- Otherwise, if the number of decoupled strands extracted from the task is only two, the multithreaded decoupled execution mode is chosen.
- Otherwise, decoupled execution mode is chosen.

At the end of the selection process, one version of task code is generated. Although we utilize our knowledge of the application to statically pick the execution mode, our scheme can also can be implemented into a compiler by detecting data-dependent control flow.

3.4 Context Scheduling Algorithm

An architecture supporting both decoupled execution and multithreading together must decide how the strands from different threads should interact and share resources, such as the execution pipeline. The instruction scheduling algorithm must intelligently decide which context should get scheduling priority among a pool of strands from different threads, in a way that promotes fairness and thereby improves throughput and utilization of the execution pipeline.

When our proposed hybrid technique is in multithreading-only mode, round-robin is used to provide fairness by evenly distributing priority among the threads. However, when either of the two decoupled modes is active, deciding which strand of which thread should be given priority is less clear. For example, either the leading strand (executing furthest ahead) or the following strand (executing furthest behind) of a thread could be prioritized. Prioritizing the leading strand of a thread would enable that strand to execute far ahead and support a greater amount of latency tolerance. However, prioritizing the leading strand can cause the data queues to fill up, and artificially starve that strand from issuing instructions until there is free space in the data queue. On the other hand, prioritizing the following strand of a thread enables the data queues to be kept empty, with the downside being the potential reduction in latency tolerance ability.

In practice, the fullness of a data queue is a dynamic property that depends on runtime behavior, while many scheduling algorithms are fixed and do not change over time. Therefore, we combine static scheduling algorithms to create a novel dynamic algorithm. The algorithm prioritizes strands with full data queues first, then leading strands from each thread, followed by subsequent strands.

The dynamic instruction scheduling algorithm:

- Strands with full data queues get the highest priority.
- Otherwise, round robin among the threads considering only the leading strand.
- Continue to round robin among the threads by considering only the next strand.
- Finish when following strands have been considered.

4 Modeling Methodology

To more fairly compare and evaluate the performance and energy efficiency of our hybrid technique and the five commonly used techniques, we perform a comprehensive design space exploration varying chip and core configurations. We build detailed performance models of each latency tolerance technique and a detailed physical design model generated using synthesis and analytical SRAM models for a 45nm CMOS manufacturing process to determine benchmark runtime, dynamic energy consumption, and leakage energy consumption. Each latency tolerance technique is individually implemented with complexity-effective and energy-efficient hardware to avoid unfair comparisons. The runtime and energy values for a single technique are then analyzed to determine the Pareto-optimal configurations with respect to energy-efficiency.

4.1 Modeling Infrastructure

Figure 3 presents our flow for evaluating the performance, area, and energy consumption of the techniques. Each architectural configuration's performance is measured using RigelSim, a cycle-accurate performance model, while the associated area and energy costs of the design are based on models derived through a combination of RTL synthesis and CACTI 6.0 [16].

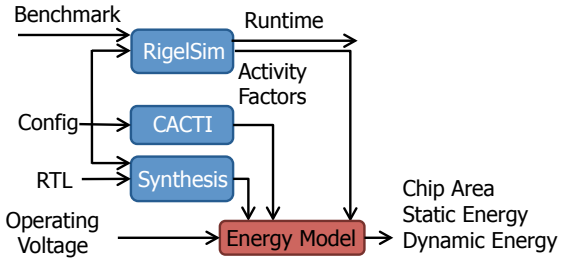


Figure 3: Energy, area, and leakage generation workflow.

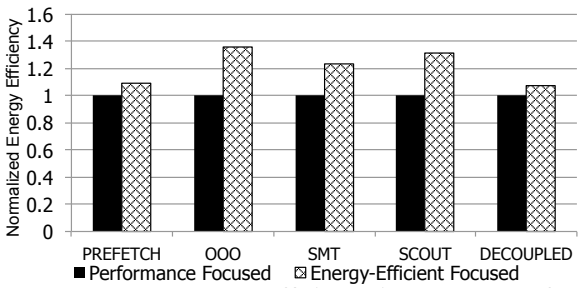


Figure 4: Average energy-efficiency improvement of each technique using tuned implementations.

Performance Modeling

We evaluate the latency tolerance techniques on Rigel, a 1024-core data parallel processor, with cores and caches organized into a 3-level cache hierarchy [10]. Each core has private L1 instruction and data caches, a single-precision floating point unit, and a MIPS-like RISC instruction set. Eight cores form a *cluster* and share a unified L2 cache, with lower access latency than the L3 cache. All 128 clusters share a banked L3 cache via an on-chip interconnection network, which is connected via GDDR5 memory controllers to off-chip DRAM.

The latency tolerance techniques compared in the design space exploration have been implemented in RigelSim, an execution-driven simulator that models the cores, caches, memory controllers, and DRAM. The core models the structure of the pipeline, including pipeline stages, functional units, and storage components such as branch prediction tables, register files, reorder buffers, load-store units and instruction windows. Additionally, each memory structure is modeled with a specific number of read and write ports in the simulator.

Physical Design Modeling

For modeling the energy and area of each design, a CMOS 45 nm manufacturing process is used. CACTI is used to model the register files, the caches (L1, L2 and L3), and other storage components (BTB, instruction queues, etc.); synthesized Verilog is used for all other major core structures such as functional units, pipeline latches, and bypass logic. As area and energy of a module depends on how aggressive the implementation is, each chip component was generated for a range of delays. Specifically, CACTI was modified to output the energy-area-delay points of all configurations considered during its internal design space exploration, and our synthesis toolflow was used to generate different circuit implementations at different target clock frequencies. We chose the most efficient implementation of a circuit based on the target frequency chosen during the design space exploration.

To model the effect of the operating voltage, we used voltage scaling equations extrapolated from SPICE simulations of 45 nm circuits. These equations model the impacts of scaling voltage on circuit delay, energy consumption, and leakage power. We then tied these voltage scaling equations together to the Pareto-optimal energy, area, and delay values for each of our modules. Linking these values into the core pipeline model defined by the simulator, we were able to approximate the operating frequency of each core configuration under different voltage operating points.

The final step in creating the energy model requires linking the energy costs at the module level with architectural activity factors to generate dynamic energy consumption. The simulation infrastructure outputs the activity for each component in the core and memory system, measured by counting the number of accesses to the structure. Reads, writes, and associative lookups are counted for storage components, while the number of active cycles are counted for all other structures. These activity factors are then used with the module configurations and the selected operating voltage to determine area, dynamic energy and leakage.

4.2 Efficient Technique Implementation

For each latency tolerance technique, we implement energy-efficient extensions to improve these techniques over performance-focused implementations. Figure 4 presents the impact on energy efficiency when deploying these techniques on a 1024-core data parallel processor with a two-wide issue core and the data parallel benchmarks found in Section 5. The extensions improve hardware prefetching by 8%, out-of-order execution by 36%, multithreading by 23%, hardware scout by 32%, and decoupled by 8% on average across the benchmarks. Additional details on methodology is found in Section 5.

Hardware Prefetching: In our design space exploration, we consider both next-line and stride-based prefetching [5]. Table-based stride prefetching is used to improve energy-efficiency by better predicting the access stream and improving timeliness over next-line prefetching. Software can statically switch off prefetching if it is not useful.

Out-of-Order: We include complexity-effective hardware to improve energy efficiency. To simplify the instruction window, we include dependence-based instruction

steering which uses simple hardware FIFOs [18]. We also enable aggressive partitioning of the core into floating point and integer pipes to reduce physical register file complexity, and partition the reorder buffer and register renaming free list to reduce the number of read and write ports [24].

Multithreading: We implement simultaneous multithreading to better utilize issue slots as issue width increases. Cache set hashing is used on the L1 and L2 cache to more uniformly distribute memory addresses and reduce the effects of cache contention between threads [8, 11]. A bitwise XOR hashing function on the memory address is used with low additional hardware complexity. Software can statically configure the number of active threads on each core to further reduce the effects of cache contention.

Hardware Scout: The extra energy spent preexecuting the instruction stream can be improved by avoiding preexecution of useless instructions [17]. For example, floating point instructions do not generally contribute to generating prefetches and can not be executed to reduce functional unit energy consumption. We also eliminate short scout sessions and avoid overlapping scout sessions.

Decoupled Execution: The instruction overhead found in decoupled architectures can be partially alleviated by reducing the complexity for each strand context. We therefore reduce the number of instructions that each strand can issue per cycle, leading to a lower number of read and write ports in the register files, instruction and data queues.

5 Experimental Setup

We present the parameters, microbenchmarks, and data parallel benchmarks for the design space exploration. The design space is listed in Table 1. We vary all major components in the 1024-core Rigel chip including issue width, number of functional units, and cache sizing. We also vary specific parameters for each latency tolerance technique.

5.1 Microbenchmarks

We utilize microbenchmarks to isolate five key code patterns found in data-parallel applications: compute-intensive, data-dependent control flow, data-sharing intensive, pointer-chasing, and memory-streaming intensive. These patterns stress the ability to achieve energy-efficiency, such tolerating functional unit and memory latency and exploiting locality in the cache hierarchy. We note that this collection is not a comprehensive taxonomy; rather these are the patterns that we find most significantly affect the techniques we evaluate. Additionally, these patterns are not mutually-exclusive, and we find in practice multiple patterns can and are found in applications.

Shared Parameters	
Target Frequency	500 MHz - 2.5 GHz
Operating Voltage	0.7V - 1.4V
Base Core Pipeline	8 stage - Fetch, Decode, Schedule/RF, Execute (4), Writeback
Issue Width	1-4
BTB	8-64 entry
GShare Table	32-256 entry
LDQ, STQ, MissQueue	LDQ/STQ 8-32 entry; MQ 4-16 entry
L1I (2-way), L1D (4-way)	L1I 2kB, 1 cycle; L1D 1-4kB, 1 cycle
L2 (8-way)	32-128kB, 4 cycle, shared by 8 cores
L3 (32-bank, 4-way per bank)	4MB total, 128kB per bank, 32 cycle access through on-chip network
DRAM	8 Channels GDDR 5
Hardware Prefetch Parameters	
Nextline Prefetching	1-4 cache lines
Stride Prefetching Table	8-32 Entries
Out-of-Order Parameters	
Reorder Buffer	32-128 entry unified or partitioned
Instruction Window	16-32 entry Assoc. or (16) 16-entry FIFOs
Physical Registers	128-256 Unified or Split FP/INT
Multithreaded Parameters	2-4 SMT threads
Decoupled Parameters	16-128 entry partitioned data queues

Table 1: Design space exploration parameters.

Compute-intensive: The *compute-intensive* pattern isolates the ability to tolerate functional unit latency. Code with this pattern usually consists of dependent chains of arithmetic and floating-point instructions as opposed to memory instructions. The microbenchmark consists of a loop that performs iterations of floating point computation on values in the register file. Each iteration of the loop executes a collection of floating point instructions connected in a tree-like structure where each stage fans into the next, much like a reduction operation. While there is ILP that can be exploited at each stage in the tree, there is latency that must be tolerated to keep the core from stalling.

Data-dependent Control Flow: The *data-dependent* control flow pattern occurs when the next basic block to execute cannot be known ahead of time. To continue execution under long instruction stalls, either another instruction stream must execute or speculative execution performed. The microbenchmark consists of a loop where each iteration calculates a value, which is then compared to a threshold. If the calculated value is above the threshold, a running sum is updated. Whether the running sum will be updated or not corresponds to different control paths. The threshold conditional branch is statically biased to 75% taken.

Data-sharing intensive: Memory-intensive code with *high reuse* exhibits a large number of memory accesses favorable to a cache hierarchy. Peak performance is achieved by fitting the dataset of a thread into the cache. *Sharing intensity*, the fraction of data shared between threads, impacts the amount of cache space required to limit cache

contention. The data-sharing microbenchmark consists of a loop dominated by memory accesses. Each thread of execution has a vector dataset which is iterated over continually in a loop. The dataset size is set such that data for 8 baseline threads in a cluster can fit in the L2 cache without aliasing.

Pointer-Chasing: The *pointer-chasing* pattern occurs when an application is dominated by memory accesses used to traverse nodes in linked-lists or graphs. Each node is a dynamic element whose address is not known ahead of time and can be highly unpredictable, requiring the inspection of pointers in the parent node's data structure. The pointer-chasing microbenchmark is a linked-list traversal, with a small number of floating point instructions used to update a value in each node traversed. Each task operates on a separate linked-list and each node aligns to a cache line irregularly skewed across the address space, simulating the effects of dynamic memory allocation and linked list manipulation.

Memory-streaming intensive: The memory-streaming pattern isolates the ability to tolerate large amounts of memory latency. The pattern exhibits a large number of memory accesses of data that exhibit low reuse and often miss in the cache hierarchy, exposing long memory access latencies. The microbenchmark uses a simple vector addition code to model the memory-streaming code pattern. Using this code base, we model the situation where the dataset is resident in the L3 cache, enabling access latencies in the tens of cycles and a high amount of available bandwidth.

5.2 Data Parallel Benchmarks

We use a set of nine optimized parallel kernels from scientific and visual computing applications implemented for Rigel. The benchmarks exhibit a high degree of parallelism and are written using a task-based, barrier-synchronized work queue model implemented fully in software. The benchmarks include black scholes (`blackscholes`), conjugate gradient linear solver (`cg`), coloumbic potential with cutoff (`cutcp`), dense matrix multiply (`dmm`), 2D fast fourier transform (`fft`), 2D stencil computation (`heat`), k-means clustering (`kmeans`), medical image reconstruction (`mri`), and image edge detection (`sobel`). Each benchmark is executed for at least one billion instructions.

Each benchmark is tuned at the source level using loop unrolling and software pipelining techniques, and then compiled using LLVM with optimizations turned on, providing significant latency tolerance over naive code. For decoupled execution and the hybrid technique proposed in this work, the kernels are decomposed into strands as found in [6], with support for extracting more memory-consuming strands for functional unit latency tolerance. The partitioning scheme is implemented into a binary rewriter, which automatically generates decoupled code by decompiling the binary and applying the partitioning scheme.

6 Hybrid Technique Architecture Evaluation

6.1 Scheduling Algorithm Evaluation

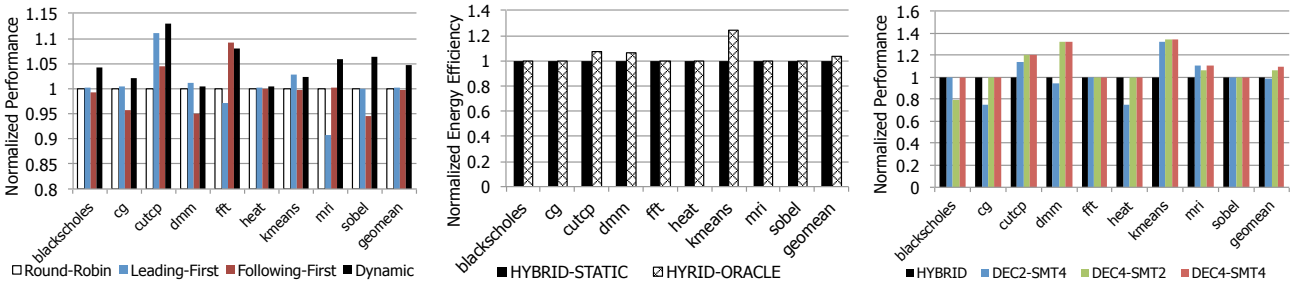
Figure 5a presents the evaluation of the instruction scheduling algorithms for our hybrid technique. Round-robin scheduling across all strands on a single core is considered as the baseline for comparison. Each benchmark is run on a fixed configuration 1024-core system with a two-wide issue core and the smallest cache sizes. As expected, the baseline round-robin scheme does not perform the best on any of the benchmarks. However, the impact of choosing one scheduling algorithm over another is quite significant and can affect performance more than 15%. Choosing a static scheduling policy gives mixed results as prioritizing the leading strand causes significant starvation on `fft` and `mri` due to full data queues, while prioritizing the following strand significantly reduces the ability to execute ahead and tolerate memory latency on `cg`,`cutup`, `dmm`, and `sobel`. Our dynamic scheduling algorithm improves performance by nearly 5% on average and in some cases improves upon the static algorithms, by avoiding the performance pitfall case when the data queues are full while enabling the leading strands to execute as far ahead as possible.

6.2 Static Mode Selection Evaluation

We evaluate the ability of our static scheme to select the best execution mode for our hybrid technique by comparing the Pareto optimal design points from our full design space exploration. Figure 5b compares our hybrid latency tolerance technique with an oracle statically choosing the best hybrid execution mode. Overall, we find that our static algorithm performs within 8% of the oracle. Our scheme is able to statically choose the best mode on many of the benchmarks, with the exception of `cutup`, `dmm`, and `kmeans`. In these benchmarks, choosing multithreading over decoupled (`dmm`), two-way multithreaded decoupled over three-strand decoupled (`kmeans`) and two-way multithreaded decoupled over multithreading (`cutup`) can improve energy-efficiency. While dynamically choosing the execution mode is left for future work, we find that the impact on energy efficiency is likely to be relatively low.

6.3 Limited Execution Mode Evaluation

We evaluate our decision to restrict our hybrid technique to three fixed execution modes by comparing with full combinations of decoupled execution and multithreading. Figure 5c presents the performance of our hybrid technique and



(a) Dynamic Scheduling Algorithm Evaluation (b) Static Execution Mode Evaluation (c) Comparison to Combination Alternatives

Figure 5: Relative performance and energy-efficiency of our dynamic scheduling (a) and static mode selection algorithms (b) against alternatives and an oracle. Relative performance of our hybrid technique and naive fixed configurations (c).

fixed configurations supporting two and four-strand decoupled execution (DEC2, DEC4) and two and four-way multithreading (SMT2, SMT4). The experiment is performed on a fixed configuration 1024-core system with a two-wide issue core and the smallest caches sizes. For each benchmark, the best configuration of decoupling and multithreading on each alternative design is presented. On average, our hybrid technique performs within 10% of the higher-complexity designs that must support significantly more hardware contexts. In the case of `cutup`, `dmm`, and `kmeans` the ability to execute in multithreading mode with more than two strands for each thread provides a benefit. However, increasing the number of contexts in our hybrid technique from four to six would enable two-way multithreading with three strands apiece and reduce much of the advantage of the alternative designs while requiring less complexity.

7 Design Space Evaluation

This section presents the results of the comprehensive design space exploration for the latency tolerance techniques on the 1024-core data parallel processor, comparing performance, energy consumption, and chip area using the design space parameters in Table 1. Energy efficiency of a configuration is defined as the ratio of normalized performance improvement divided by the normalized energy consumption. Energy consumption and performance improvement are normalized to a one-wide issue in-order baseline with the smallest configuration. Pareto-optimal curves of the design space exploration are presented when appropriate. As performance increases and runtime decreases, component activity and power consumption increase, as seen in the curves representing a fixed power budget of 150 Watts.

7.1 Microbenchmarks

To understand the impact of choosing one technique over another, microbenchmarks are utilized to isolate application behavior. Figures 6a, 6b, 6c, 6d, and 6e present the Pareto-optimal curves of the latency tolerance techniques. As performance improves, total energy can actually be lower than the baseline as leakage energy is avoided.

Figure 6a presents the compute-intensive microbenchmark. Due to the lack of memory accesses, prefetching techniques such as hardware prefetching and hardware scout cannot provide any benefit. Both decoupled and out-of-order extract ILP from a single thread, with the decoupled providing better energy-efficiency due to lower hardware complexity. Multithreading has low hardware complexity and is able to tolerate even more floating point latency with four threads, providing the best energy-efficiency. Our proposed hybrid technique operates in decoupled mode, extracting three strands to tolerate floating point unit latency. Overall, energy-efficiency is improved through out-of-order by 17%, multithreading 36%, decoupled 30%, and hybrid 30% over hardware prefetching at the 150 Watt power budget.

Figure 6b presents the data-dependent control flow microbenchmark. Similar to the compute-intensive pattern, hardware prefetching and hardware scout cannot improve performance. Decoupled execution experiences loss-of-decoupling when data-dependent control flow is present, limiting performance to in-order execution. Both out-of-order and multithreading have the ability to tolerate data-dependent control flow. However, multithreading is more energy efficient as speculation and the dynamic hardware in out-of-order consumes extra energy. Our proposed hybrid technique detects data-dependent control flow at compile time and enables multithreaded execution, resulting in high energy-efficiency. Overall, energy-efficiency is improved in out-of-order by 13%, multithreading 77%, and hybrid 77% over hardware prefetching at the 150 Watt power budget.

Figure 6c presents the data-sharing intensive microbenchmark. With a large amount of predictable L1 misses, hardware prefetching provides substantial performance. While hardware scout provides better prefetching capability, the added extra performance is offset by the extra energy required to dynamically preexecute the instruction stream. Multithreading experiences high levels of cache contention which increases cache activity and energy con-

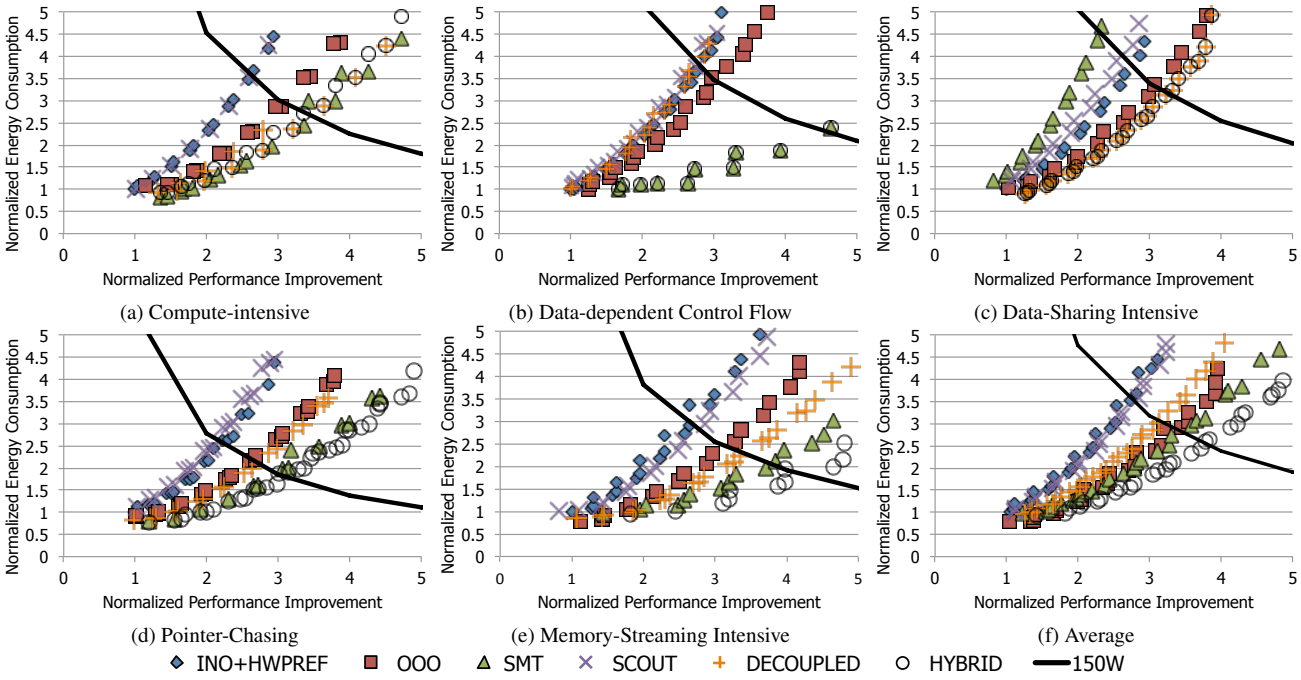


Figure 6: Pareto-optimal energy efficiency for techniques on microbenchmarks normalized to hardware prefetching.

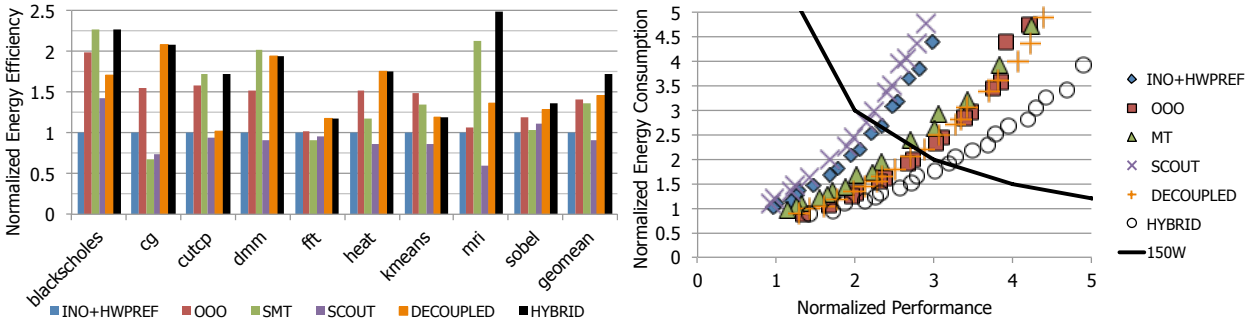
sumption, and prefers larger caches to reduce contention at a higher cost in energy consumption. Out-of-order and decoupled execution offer the best energy-efficiency, with only low complexity implementations of out-of-order required to tolerate the relatively small L2 access latency. Our hybrid proposal executes in decoupled mode, yielding high energy-efficiency. Overall, energy-efficiency is improved in out-of-order by 12%, decoupled, 21%, and hybrid 21% over hardware prefetching at the 150 Watt power budget.

Figure 6d presents the pointer-chasing microbenchmark. Both hardware prefetching and hardware scout cannot easily predict the access stream, leading to lower performance and significant energy waste. Out-of-order tolerates memory latency by executing the update portion of the microbenchmark independently, while only needing a small instruction window to do so. Decoupled execution exhibits similar behavior to out-of-order, with significantly lower complexity. The independent nature of multithreading provides more outstanding memory accesses, which enables substantial performance improvement over out-of-order or decoupled execution. The hybrid technique enables two threads of decoupled execution on the same core, enabling better performance than four-way multithreading. Overall, energy-efficiency is improved in out-of-order by 23%, multithreading 33%, decoupled, 26%, and hybrid 42% over hardware prefetching at the 150 Watt power budget.

Figure 6e presents the memory-streaming microbenchmark. Both hardware prefetching and hardware scout can tolerate a large amount of memory latency. However, prefetches are reactively generated on cache misses, resulting in the core spending significant time stalled. Out-of-order needs a large and complex instruction window to provide enough independent work tolerate long memory latencies. Decoupling operates similarly to out-of-order, with substantially reduced hardware complexity and energy consumption. Multithreading scales the number of threads to uncover more memory misses and tolerate memory latency at the cost of more context hardware and cache space. The hybrid technique enables two threads of decoupled execution on the same core resulting in better performance than four-way multithreading. Overall, energy-efficiency is improved in hardware scout by 4%, out-of-order by 17%, multithreading 39%, decoupled, 27%, and hybrid 47% over hardware prefetching at the 150 Watt power budget.

7.2 Average Energy Efficiency

Figure 6f presents the mean result of the design space exploration, with the performance improvement and energy consumption of each architectural configuration averaged across the microbenchmarks. Our proposed hybrid technique is the most energy-efficient on average, with multithreading being preferred over out-of-order and decoupled execution. While out-of-order is not the most energy efficient on any single microbenchmark, it provides robust energy-efficiency across the benchmarks as compared with decoupled or multithreading, which can experience pitfalls from data-dependent control flow and cache contention. Prefetching can tolerate some memory latency, but cannot tolerate floating-point unit latency or much memory latency



(a) (b)

Figure 7: Pareto-optimal energy efficiency for techniques for each benchmark and averaged across benchmarks.

from pointer-based memory accesses. Our hybrid proposal is robust across all the microbenchmarks, and does not suffer either a performance or energy pitfall. Across the microbenchmarks, energy-efficiency is improved on average in hardware scout by 2%, out-of-order by 20%, multithreading 24%, decoupled 15%, and our hybrid technique 31% over hardware prefetching at the 150 Watt power budget.

Averaging energy-efficiency across the microbenchmarks weights the importance of each code pattern equally. We also varied the weighting and relative importance of the five microbenchmarks from 0 to 100% in increments of 10%, and determined the most energy-efficient technique for all possible weighting combinations. The proposed hybrid technique is the most energy-efficient across 78% of the combinations, followed by out-of-order at 22%.

7.3 Visual Computing Benchmarks

The microbenchmarks illustrate the profound effect of application behavior on the energy-efficiency of latency tolerance techniques. With this in mind, we perform the design space exploration on the benchmarks from visual computing applications. Figure 7a presents the most energy-efficient configurations for each benchmark normalized to the hardware prefetching baseline. Figure 7b presents the average of a single configuration across all the benchmarks.

On average, out-of-order, multithreading and decoupled techniques provide similar levels of energy-efficiency, with decoupled being preferred due to the majority of benchmarks having memory-intensive patterns. These techniques provide 66%-143% better energy efficiency than hardware prefetching and hardware scout. However upon closer inspection, decoupled and multithreading techniques perform quite differently depending on the code pattern. Benchmarks dominated by data-dependent control flow such as `blackscholes`, `cutcp`, and `kmeans` favor multithreading over decoupled execution by 14% to 76% due to the ability to execute another thread instead of stalling and waiting to determine the next basic block. Memory-intensive benchmarks such as `fft`, `heat`, and `sobel` favor decoupled execution over multithreading by 26% to 140%, as the cache footprint is kept low by maintaining fewer threads per core. Out-of-order provides general performance improvement across all the benchmarks similar to the microbenchmarks, but this benefit is negated by the energy overhead required for dynamic scheduling. Hardware scout is not competitive with hardware prefetching in terms of energy-efficiency, being dominated by the extra energy to preexecute the instruction stream to generate prefetches.

Our hybrid technique enables a larger degree of energy efficiency. By detecting data-dependent control flow in the compiler, `blackscholes` and `cutcp` operate in multithreading mode, while other benchmarks are in decoupled mode. Only two strands are extracted in `dmm`, `kmeans`, and `mri` enabling two-way multithreading. On average the energy-efficiency improvement of the hybrid latency tolerance technique is 28% to 89% over hardware prefetching, out-of-order, multithreading, hardware scout prefetching, and decoupled techniques alone.

7.4 Combining Other Techniques

We perform a design space exploration to compare other combinations of hardware prefetching, out-of-order execution, multithreading, hardware and decoupled execution. Figure 8 presents the Pareto-optimal energy efficiency of these combinations on the benchmarks. Where appropriate, we include the ability of a combination to statically disable hardware prefetching, hardware scout, or multithreading if it does not improve performance. For clarity, we omitted combinations such as decoupled execution with hardware scout and out-of-order due to their similarities.

Adding hardware prefetching improves energy-efficiency on out-of-order 8%, multithreading 6%, and decoupled execution 6%, though additional memory latency tolerance does not benefit every benchmark. Adding hardware scout to out-of-order and multithreading gives mixed results, with only `cg` and `fft` for multithreading seeing a benefit due to better cache contention coverage, and `blackscholes` and `sobel` for out-of-order see-

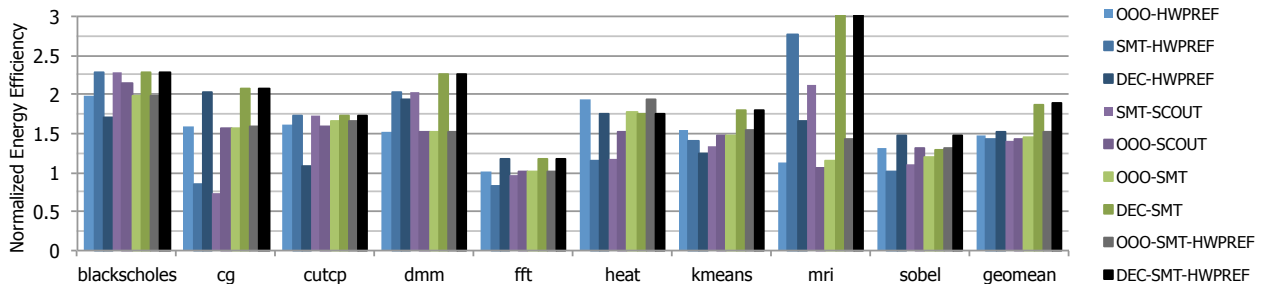


Figure 8: Average improvement of energy efficiency of combinations of hardware prefetching, out-of-order, multithreading, hardware scout, and decoupled execution on data parallel benchmarks. Normalized to hardware prefetching.

ing a small benefit on long cache misses. Combining out-of-order and multithreading yields little benefit, as the fixed cost of implementing dynamic scheduling hardware already provides some amount of latency tolerance, while adding additional threads can also increase cache contention. Combining decoupling with multithreading is the best combination, due to low hardware complexity and robust performance across varying workloads, providing an additional 26% improvement in energy-efficiency over any other combination. Adding hardware prefetching to a combination of decoupled execution and multithreading can further improve energy-efficiency, as shown by the 18% improvement in `sobel`.

7.5 Average Area Requirements

Figure 9 presents the mean performance improvement compared with the absolute chip area for the design space exploration on our 1024-core data-parallel processor. The five microbenchmarks are equally weighted. Voltage scaling provides significant performance improvement over hardware prefetching without cost in area. Out-of-order is significantly more expensive than other techniques. At lower performance improvement levels, hardware prefetching, multithreading, and hardware scout are more area efficient than our proposed hybrid approach. However, these techniques must rely on larger caches to continue to improve performance, and eventually become less area-efficient than our hybrid technique. The memory-latency tolerance ability of decoupling and our hybrid technique is less sensitive to cache-sizing and as a result the area-efficiency scales much more favorably. Overall, our hybrid technique substantially improves performance over hardware prefetching while keeping chip area under 400 mm^2.

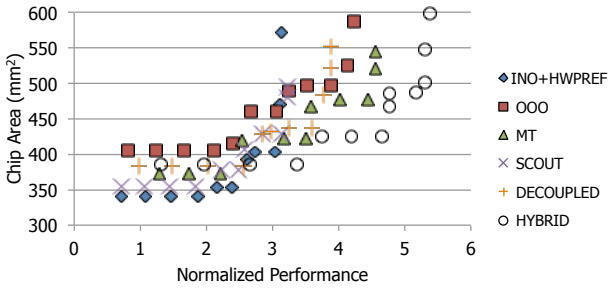


Figure 9: Pareto optimal curves for chip area of each technique averaged across all microbenchmarks.

8 Related Work

Prior work has described methods for investigating energy-performance tradeoffs when considering in-order and out-of-order uniprocessors [2]. That prior research explores the design space of a single processor, varying architectural parameters using CACTI and synthesis flows. We utilize a similar approach, but build upon it considerably by introducing hardware prefetching, multithreading, hardware scout, and decoupled execution as potential latency tolerance techniques and evaluate on a 1000-core data-parallel processor and benchmarks. Other research has investigated the design space for multi- and manycore processors. Mahesri et al. investigated area-efficient throughput-oriented core architectures [15]. Huh et al. presented early work on the design space of multicore CMPs, and enumerate important application characteristics [9]. Bakhoda et al. presented GPU benchmarks and perform sensitivity analyses to chip design parameters [**?**]. Our work improves over past work by focusing on exploring the design space and giving special attention to modeling the physical design, including energy consumption.

Combinations of latency tolerance techniques have been implemented in the context of serial-performance focused processors. Intel's i7 [20] and IBM's POWER7 [23] implement out-of-order and simultaneous multithreading. IBM's POWER6 architecture implements multithreading and a restricted form of hardware scout, called load-lookahead prefetching [12]. Other work proposes dynamic instruction partitioning into separate threads and leverages out-of-order hardware in order to provide memory latency tolerance [19]. Additionally, many designs have implemented hardware prefetching. Our work investigates various combinations of these techniques in the context of 1000-core data-parallel processors and benchmarks and proposes a novel and low-complexity hybrid technique to provide robust energy-efficiency.

9 Conclusion

Modern data parallel processors require energy-efficient latency tolerance to reach the goal of maximizing performance per Watt. However, commonly used techniques fall short of this goal, either by suffering from pitfalls due to application behavior, additional hardware complexity, or excessive energy waste. We propose a novel hybrid latency tolerance technique leveraging both multithreading and decoupling to provide robust performance and energy-efficiency. While multithreading and decoupling in isolation have performance pitfalls on different code patterns commonly found in data parallel workloads, intelligently combining the two techniques can avoid these pitfalls and improve energy-efficiency significantly.

Leveraging the properties of multithreading and decoupled execution, we design static execution mode selection and dynamic instruction scheduling algorithms for our proposed hybrid technique. We then utilize high-fidelity performance and physical design models and perform a comprehensive design space exploration to compare the energy-efficiency of our hybrid technique with commonly used latency tolerance techniques on a 1024-core data parallel processor. Our hybrid latency tolerance technique improves energy-efficiency over other single techniques by 28% to 89%, and over any other combination of techniques by over 26% on data parallel benchmarks.

10 Acknowledgements

The authors acknowledge the support of the Semiconductor Research Corporation (SRC). The authors also thank the Trusted ILLIAC Center at the University of Illinois for use of the computing cluster, Mark Horowitz and Haowei Zhang of Stanford for initial physical design estimates, and the anonymous referees for their feedback.

References

[1] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Boosting mobile GPU performance with a Decoupled Access/Execute fragment processor. In *ISCA'12*, 2012.

[2] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *ISCA'10*, pages 26–36. ACM, 2010.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS'09*, pages 163 –174, 2009.

[4] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25:32–45, 2005.

[5] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609 –623, May 1995.

[6] N. C. Crago and S. J. Patel. Outrider: efficient memory latency tolerance with decoupled strands. In *ISCA'11*, pages 117–128, 2011.

[7] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS'97*, pages 68–75, 1997.

[8] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *ICS'97*, pages 76–83, 1997.

[9] J. Huh, D. Burger, and S. Keckler. Exploring the design space of future CMPs. In *PACT'01*, pages 199 –210, 2001.

[10] 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 *ISCA'09*, pages 140–151, 2009.

[11] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *HPCA'04*, pages 288–299, 2004.

[12] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Res. and Development*, 51(6):639–662, Nov. 2007.

[13] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *MICRO'10*, pages 213–224, 2010.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[15] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel. Tradeoffs in designing accelerator architectures for visual computing. In *MICRO'08*, pages 164–175, 2008.

[16] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO'07*, pages 3–14, 2007.

[17] O. Mutlu, H. Kim, and Y. N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro*, 26:10–20, 2006.

[18] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA'97*, pages 206–218, 1997.

[19] J.-M. Parcerisa and A. Gonzalez. Improving latency tolerance of multithreading through decoupling. *IEEE Transactions on Computers*, 50(10):1084–1094, 2001.

[20] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Kottapalli. A 45nm 8-core enterprise Xeon processor. In *IEEE ISSCC*, pages 56–57, 2009.

[21] J. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A. Leon, and A. Strong. A 40nm 16-core 128-thread CMT SPARC SoC processor. In *IEEE ISSCC*, pages 98–99, 2010.

[22] J. E. Smith. Decoupled access/execute computer architectures. In *ISCA'82*, pages 112–119, 1982.

[23] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. Rubio, F. Rawson, and J. Carter. Architecting for power management: The IBM POWER7 approach. In *Proceedings of the 16th HPCA*, pages 1–11, 2010.

[24] K. Yeager. The Mips R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, April 1996.