OUTRIDER: Efficient Memory Latency Tolerance with Decoupled Strands

Neal C. Crago and Sanjay J. Patel
University of Illinois at Urbana-Champaign
crago, sjp@illinois.edu

ABSTRACT

We present OUTRIDER, an architecture for throughput-oriented processors that provides memory latency tolerance to improve performance on highly threaded workloads. OUTRIDER enables a single thread of execution to be presented to the architecture as multiple decoupled instruction streams that separate memory-accessing and memory-consuming instructions. The key insight is that by decoupling the instruction streams, the processor pipeline can tolerate memory latency in a way similar to out-of-order designs while relying on a low-complexity in-order micro-architecture. Moreover, instead of adding more threads as is done in modern GPUs, OUTRIDER can tolerate memory latency with fewer threads and reduced contention for resources shared amongst threads.

We demonstrate that OUTRIDER can outperform single threaded cores by 23-131% and a 4-way simultaneous multithreaded core by up to 87% on data parallel applications in a 1024-core system. Moreover, OUTRIDER achieves these performance gains without incurring the overhead of additional hardware thread contexts, which results in improved area efficiency compared to a multithreaded core.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*

General Terms

Design, Performance

Keywords

Accelerator, Computer Architecture, Memory Latency

ISCA'11, June 4–8, 2011, San Jose, California, USA.

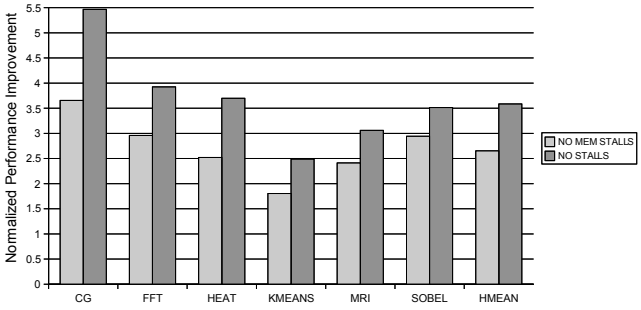

Figure 1: Potential performance improvement of highly parallel benchmarks on a baseline in-order 1024-core design when memory stalls and all resource stalls are removed.

1. INTRODUCTION

Execution stalls due to memory latency and bandwidth constraints are the limiting factor for performance in highly parallel workloads. Figure 1 presents the impact of execution stalls on performance for several visual computing benchmarks on a 1024-core accelerator system by comparing a baseline single-threaded, two-wide issue, in-order processor with perfect branch prediction and instruction cache against two scenarios: the baseline augmented with zero-latency memory accesses (NO MEMSTALLS), and the baseline idealized with both zero-latency memory accesses and zero-latency functional units (NO STALLS). In these scenarios, the results from the function units and memory subsystem are available immediately. Additional information on the baseline used can be found in Table 1.

We find that most of the performance lost in our 1024-core system is attributable to the memory system, rather than fetch, branch prediction, or functional unit latencies. Removing all stalls due to memory latency more than doubles performance (2.7x), whereas idealizing the entire core increases performance by 3.6x. This result demonstrates the impact of the memory system on performance, and indicates the importance of efficient mechanisms for tolerating memory latency. There have been many proposed solutions for tolerating these stalls, including data prefetching, more complex cache hierarchies, multithreading, and more complex core pipelines. In this paper, we focus our attention on core pipelines. We consider current techniques such as out-of-order and multithreaded architectures and how they tolerate memory latency, and what limits their suitability for deployment in throughput-oriented processors.

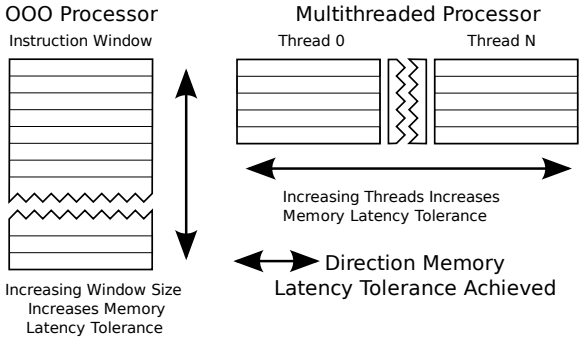

Figure 2: Memory Latency Tolerance Mechanisms of Out-of-order and Multithreaded Processors.

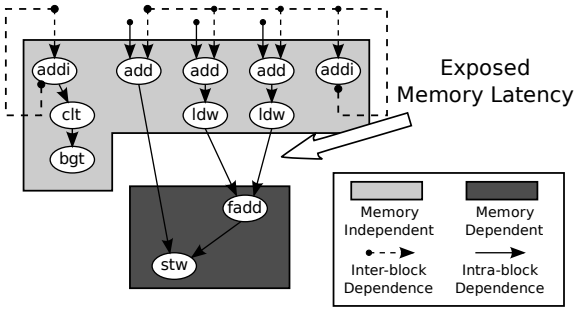

Figure 3: Dependency graph for the inner loop of code adding two vectors. The instructions either depend on memory or do not, and misses in the cache can lead to significant stalls in in-order processors.

In this paper, we present OUTRIDER, a novel implementation of a decoupled architecture that outperforms multithreading on highly parallel benchmarks in terms of performance while maintaining a low level of complexity. Decoupled architectures leverage the compiler to separate a single thread of execution into multiple instruction streams that can be executed concurrently. The thread is split into *memory-accessing* and *memory-consuming* instruction streams, which we call `strands`. The strands execute in separate hardware contexts while following the same control path through the program. Memory-accessing strands communicate data values from memory to the memory-consuming strands and have the ability to non-speculatively execute substantially ahead of the memory-consuming strands, thus tolerating memory latency.

By leveraging compilation to provide some degree of instruction parallelism, decoupled architectures can have less complex hardware than out-of-order processors. Each strand executes in-order with respect to itself, but can execute out-of-order with respect to other strands. Communication between strands also happens in-order. Thus, large associative hardware structures are avoided. Additionally, we identify and propose solutions to avoid performance cliffs and area efficiency mechanisms which improve resource utilization.

2. MEMORY LATENCY TOLERANCE APPROACHES

2.1 Out-of-Order Processors

Out-of-order processors (OOO) enable applications to execute instructions out-of-order with respect to one another by using an associative instruction window. Instructions that depend on a previous memory access are kept in the instruction window until the access completes. In the meantime, instructions not dependent on that memory access can be issued. The ability of the OOO processor to tolerate memory latency and execute independent instructions is largely dependent on the number of instructions that can be stored in the instruction window. Figure 2 shows how increasing the instruction window in OOO processors enables memory latency tolerance.

In addition to instruction windows, contemporary out-of-order processors such as Intel's i7 [22] and IBM's POWER7 [29] also utilize hardware structures such as reorder buffers, physical register files, load-store queues, and register renaming to increase the number of instantaneous instructions available for execution. The associative structures found in OOO processors do not scale well, as increasing the memory latency tolerance usually requires increasing the number of entries which not only increases the area, but also significantly increases the energy consumed per access. As a result, a significant tradeoff of energy efficiency for performance must be made. Other structures such as large physical register files and register renaming can require additional pipeline stages which introduces additional branch mispredict penalties.

2.2 Multithreaded Processors

Multithreaded processors exploit thread-level parallelism and maintain multiple contexts per core. Multithreaded processors tolerate latency by executing instructions from non-stalled threads while other threads are stalled. In a typical multithreading scheme, when a cache miss occurs the current thread is deactivated from scheduling and another thread available for scheduling replaces it in the scheduler. GPUs such as NVIDIA's Tesla [13] and CPUs such as Sun's Rainbow Falls [23] utilize high degrees of hardware multithreading to increase pipeline utilization, especially during long-latency memory accesses. Figure 2 shows how increasing the number of threads in multithreaded processors enables memory latency tolerance.

A large number of threads may be required to tolerate long memory stalls, with each additional thread requiring significant resources to store its state. The state required for multithreading includes the hardware architectural space such as the register file, and cache and memory space which holds the instruction and data working sets of the thread. As such, the scalability of multithreading is limited when area is a concern. An additional register file is required for each thread, which takes up a significant part of the processor's area. Even if the area for the scratch space per thread is justified, increased cache resources may be necessary in order to obtain performance gains. If cache resources are insufficient, contention between the threads can significantly degrade performance.

2.3 Decoupled Processors

Decoupled architectures separate the *memory-access* and *memory-consuming* instructions into separate instruction streams, called `strands`, that are executed on logically separate hardware contexts. These strands execute a part of the original

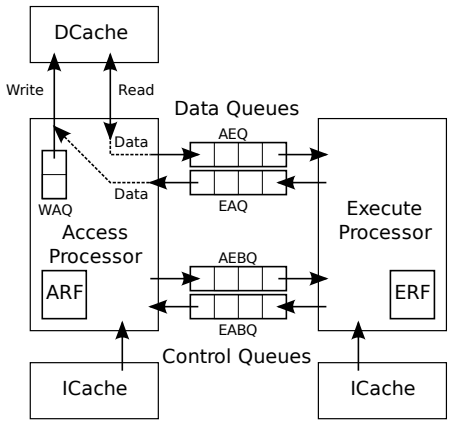

Figure 4: Organization of a Decoupled Access/Execute processor.

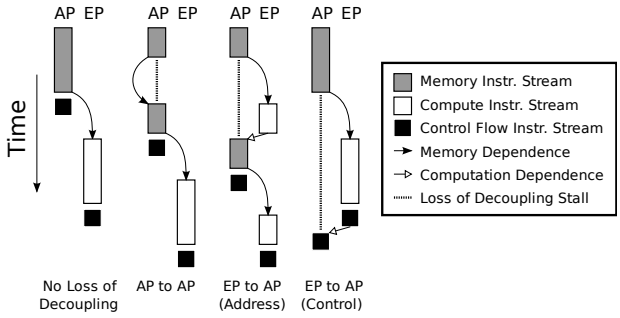

Figure 5: Loss of decoupling (LOD) events for access and execute Processors (AP, EP) in DAE architectures.

instruction stream and follow the same control flow path through the program. Strands communicate data and control flow decisions with one another and must execute together in order to perform the same program as the original sequential thread. Strands are responsible for either accessing memory or consuming memory values, with partitioning occurring along memory dependence lines. In the simplest case, there are two strands, one accessing memory and one consuming memory values. Typically, address generation instructions and memory operations are found in memory-accessing strands, while floating-point and integer arithmetic are found in memory-consuming strands. Store instructions are split across both types of strands, with the memory-accessing strand providing the address and memory-consuming strand providing the data.

The main advantage of decoupling a sequential thread into strands is the ability to tolerate memory latency. Figure 3 shows the dependency graph of the inner loop of code that adds two vectors together. In this example, only the floating-point addition (`fadd`) and store (`stw`) instructions are dependent on memory. Consider the situation when the load (`ldw`) instructions require a long latency to fill the request from memory. In-order processors stall when a primary data cache miss occurs and a dependent operation is waiting to be issued. Decoupling into separate strands enables the memory-accessing stream to continue to issue instructions while the memory-consuming strand waits on the data to return from memory. Essentially, decoupled architectures execute instructions out-of-order, but this parallelism is extracted by the compiler from the original program, rather than dynamically in hardware which leads to significantly simpler hardware similar to in-order designs. However, the complexity increase in software is on the order of other compiler transformations and utilizes much of the knowledge the compiler has already. Finally, OUTRIDER has the ability of multithreading through multiple instruction streams, but without additional increase of contention for cache resources.

Strands have their own context of program counter, register space, and mechanisms to communicate with other strands. However, because an individual strand executes only a portion of an original sequential thread, the context requirements such as register working set are significantly smaller. Considering all the strands together, the aggregate register working set requirements is on the order of the original sequential thread. Working set requirements similar to the original sequential thread can result in significantly less complexity to enable memory latency tolerance than out-of-order and in some cases multithreaded designs. On the other hand, the compiler must be designed to extract strands from the original program.

Figure 4 depicts a classical implementation of the Decoupled Access/Execute architecture [24] which was the first published decoupled architecture. The access processor (AP) and execute processor (EP) are physically separate entities that are connected only through FIFO data queues for communicating data values loaded from memory (AEQ), data values to be stored into memory (EAQ), and control flow decisions (AEBQ and EABQ). DAE achieves memory latency tolerance by executing the memory instruction stream on the AP and the computation program on the EP. The nonblocking property of the AP requires that the AP calculate control flow decisions, which it then forwards well in advance to the EP's control queue, which is later used by the EP's instruction fetch hardware.

3. TRADITIONAL LIMITATIONS

Although decoupled architectures enable memory latency tolerance, potential performance improvement is limited when the memory-accessing instruction stream cannot achieve the nonblocking property with respect to the rest of the program. These situations are known as as *loss-of-decoupling* (LOD) events [2]. Figure 5 presents the loss of decoupling events on traditional DAE processors, which represent a dependence between the processors that must be resolved before the AP is allowed to continue execution. In the optimal case there is no LOD event, and the memory-accessing stream is not blocked. **AP to AP** LOD events are caused by cache misses during indirect memory accesses, such as sparse matrices and multi-dimensional arrays, where the latency to access memory is exposed and the AP must stall. When the AP depends on data provided by the EP, LOD can also occur. This can be due to the AP needing an address generated by the EP (**EP to AP Address**) or the AP waiting on a control flow decision to be determined by the EP (**EP to AP Control**). The AP must wait on the EP to proceed, which removes the ability of the AP to execute ahead and tolerate memory latency. LOD events significantly reduce the usefulness of DAE for programs that exhibit memory indirection and compute-dependent behavior.

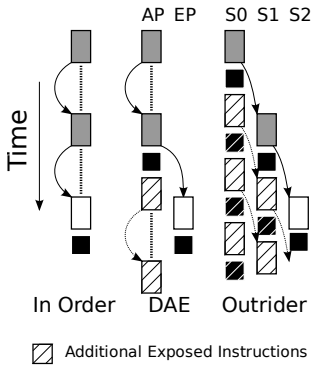

Figure 6: Handling AP to AP LOD Event in OUTRIDER**.**

Additionally, the under-utilization of resources in traditional decoupled designs that have separate processors is also an issue. These designs require separate fetch, decode, and execute resources, which can instead be shared, reducing area overheads. For example, memory-consuming strands may execute integer or floating-point arithmetic, causing replication of hardware such as multipliers and shift units between decoupled processors. Traditional fetch resources such as instruction caches also represent duplicated hardware that can be shared.

3.1 Addressing Memory Indirection

Figure 6 shows our approach to addressing the memory indirection LOD. Performance loss due to memory indirection can be alleviated by adding additional memory-accessing strands. The original memory-accessing stream can be split into multiple strands, with the goal of having at least one instruction stream without LOD. By adding strands, the amount of decoupling is increased and more parallelism is exposed. In order to handle compute-generated memory accesses, we allow floating-point instructions to exist within a memory-accessing strand, unlike DAE. Using these techniques, we can remove the memory indirection LOD in many cases and improve performance.

Increasing the number of strands increases the amount of hardware resources required for OUTRIDER. We find that many programs have only one or two levels of memory indirection, which led us to choose four strands in our design. In addition to register file and fetch resources required to support a strand, the number of data queues for communication increases quadratically with the number of strands in the system. A certain strand may wish to communicate with any of the other strands. While this might seem to represent poor hardware scalability of our technique, we find in practice that many strands do not communicate with one or more of the other strands. Additionally, the number of strands extracted is not fixed, as the compiler can generate between two and four strands. Communication pattern information is available when strands are extracted, so we utilize a dynamically-partitioned buffer and allocate a portion of the space for individual strands' data queues. The strand's FIFO queue is virtualized onto a part of the larger space. Virtualization enables area efficiency in the case that space is not allocated to facilitate communication between two strands when such communication does not exist.

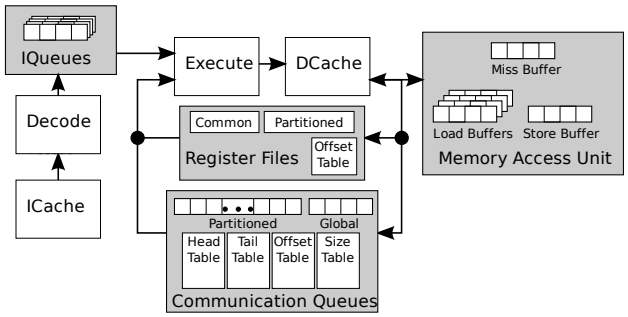

Figure 7: Implementation diagram of an OUTRIDER **architecture.**

3.2 Addressing Resource Utilization

Scaling DAE to more than two strands results in many physical processor entities that have hardware resources such as instruction fetch that could be potentially shared. Resource utilization can be improved by the use of multithreading on a single processor. We propose having a general processor with all the functional units and functionality to execute memory accessing and memory consuming code, but with four contexts which each execute a single strand. Because some software may exhibit memory indirection and enable multiple strands and some may not, we also propose dynamically partitioned data queues. When strands are extracted, a portion of the data queues is allocated to each strand. These techniques are critical for enabling area efficiency in OUTRIDER and ensure that hardware will not be left unutilized.

4. OUTRIDER ARCHITECTURE

Figure 7 shows a block diagram of the OUTRIDER architecture which supports the decomposition of a thread into a maximum of four strands working concurrently. The dynamically partitioned shared communication queue, common and partitioned thread register files, and the memory access unit are the main additions to the baseline in-order core. All structures necessary for achieving high performance with OUTRIDER consist of a low number of entries. In this section we also discuss building stronger memory consistency models. We propose a simple technique for detecting deadlock induced by software faults and a way to define the precision of exceptions on a strand-based architecture.

4.1 Communication Queues

Figure 7 presents our implementation for the queues used for communication between strands. In OUTRIDER, strands use hardware data queues for general communication or broadcast of any value, unlike the special-purpose queues found in DAE. The partitioned queues utilize pointer tables to virtualize a single buffer into multiple FIFO queues used for inter-strand communication that can extend across multiple iterations of a loop. The virtualization allows communication between all pairs of strands. A fixed-size global data queue is used for broadcasting common values shared among strands and is primarily used for communicating control flow decisions between the strands. Strands waiting for data from the queue are blocked, while other strands continue execution. The queues can achieve good performance

with a small number of entries because when data is available on the queue it is likely it will be quickly consumed by a waiting strand. Additionally, since the strands are mostly independent, the frequency with which communication occurs is relatively low.

The partitioned communication queues are mapped to architectural registers 1, 2, and 3 in the register file enabling each of the strands to communicate with one another. When register reads and writes are performed using these registers, the queue hardware tables are indexed and physical locations found in the queue. When an instruction whose result is destined for a queue is scheduled for execution, the issued instruction is given the current tail pointer value for that entry as the storage location. By giving the instructions indices as they are issued in-order, we allow instructions to be executed and written out-of-order into the data queues, but read in-order. Out-of-order writes into the data queues is particularly important for allowing instructions with both variable and static latencies such as loads and ALU operations to write to the data queues.

Figure 7 shows the communication queues and the hardware tables each strand uses to access their partition, which is configured by the compiler. For the initial study of Outrider we count the number of communication occurrences between each strand and size proportionally to the total number of communication occurrences. For strands that consume but rarely produce data, larger receive queues and smaller send queues are allocated using special purpose registers to write the hardware tables.

4.2 Register Files

Figure 7 shows the register file system used in Outrider. The compiler allocates a portion of the register file to each strand sized relative to the working register set size by using a special purpose register to set the starting offset. The strand then uses architectural register 8-31 to access its portion of the register file. Dynamic allocation enables high utilization of the register file, while allowing flexibility for varying the number of registers between the strands. This provides a benefit compared with separate and statically sized register files for each strand.

Additionally there is a small portion of the register file not privately owned by a single strand which allows constants to be shared among the strands, such as the stack pointer. These shared registers are only safe to be set at barrier synchronizations between the strands and remain unchanged throughout Outrider execution phases. Strands use architectural registers 4-7 to directly access the shared portion of the register file.

4.3 Memory Access Unit and Memory Ordering

Figure 7 shows the memory access unit (MAU) which enables multiple memory operations to be in-flight simultaneously. Each strand has its own load buffer, while the store buffer is a shared associative buffer that is used to enforce memory ordering. The low number of store entries that must be looked up associatively is effective in keeping the design of the MAU compact. The MAU is shared across all strands to enable correct memory ordering.

The store buffer is used for memory disambiguation. For each store instruction found in the original code, each strand will have either a `st_addr` or `st_data` instruction. The `st_data` instruction is found in the strand providing the data for the store, while all other strands have the `st_addr` instructions which provide the address of the original store. For each store instruction, a single entry in the store buffer is used. A single store may be completed once each strand issues its corresponding instruction. By requiring each strand to have either the address or the data, loads can perform associative lookups into the store buffer to ensure proper memory ordering

4.4 Handling Stronger Memory Consistency Models

Outrider provides correct load-store ordering within a single processor. However, for applications that need to enforce strict ordering between strands, Outrider utilizes a pair of synchronizing instructions with full memory fence semantics. The two instructions `mem_proceed` and `mem_wait` are used to signal a particular strand in a single direction through the communication queues. A strand uses `mem_wait` before a memory access to wait for the memory fence, while another strand executes a `mem_proceed` instruction to signal that the fence has been reached and that it is safe to execute. Utilizing the memory fence can build a stronger consistency model on top of the relaxed consistency model that Outrider naturally supports.

4.5 Handling Software Deadlocks and Exceptions

Outrider is a software threading technique, and deadlock in Outrider is similar to a software deadlock. Given correct program semantics and communication between strands, deadlock will not occur in Outrider. However, it can occur in improper code if all strands are waiting on the queue for data while the queues are empty, or if all strands are waiting to insert data into the communication queues, but all the queues are full. Hardware detection of deadlock for software debugging purposes is straightforward, and requires checking to see if all strands are blocked in the aforementioned case. When deadlock is detected, the pipeline, instruction, and data queues are flushed and an exception is raised to allow the runtime system to recover.

Software that targets Outrider is composed of multiple strands extracted from a single thread of execution. The concurrent execution of strands requires that one program counter (PC) be kept for each strand. To enable precise semantics for faulting memory instructions, we define the point of the exception in the memory-accessing strand to occur immediately before the memory access triggering the fault. The fault is initially stalled and the strand issuing the faulting instruction is blocked. The memory-consuming strands are allowed to continue executing until they reach the instruction dependent on the faulting instruction. At this point, the fault is delivered precisely across strands comprising the original thread: at the PC of the faulting instruction in the memory-accessing strand and at the PC of the first dependent instruction in the memory-consuming strand. Recovery from an exception requires addressing the fault in the memory-accessing strand and restarting it. Doing so causes the memory-consuming strand to unblock and execution to proceed as normal.

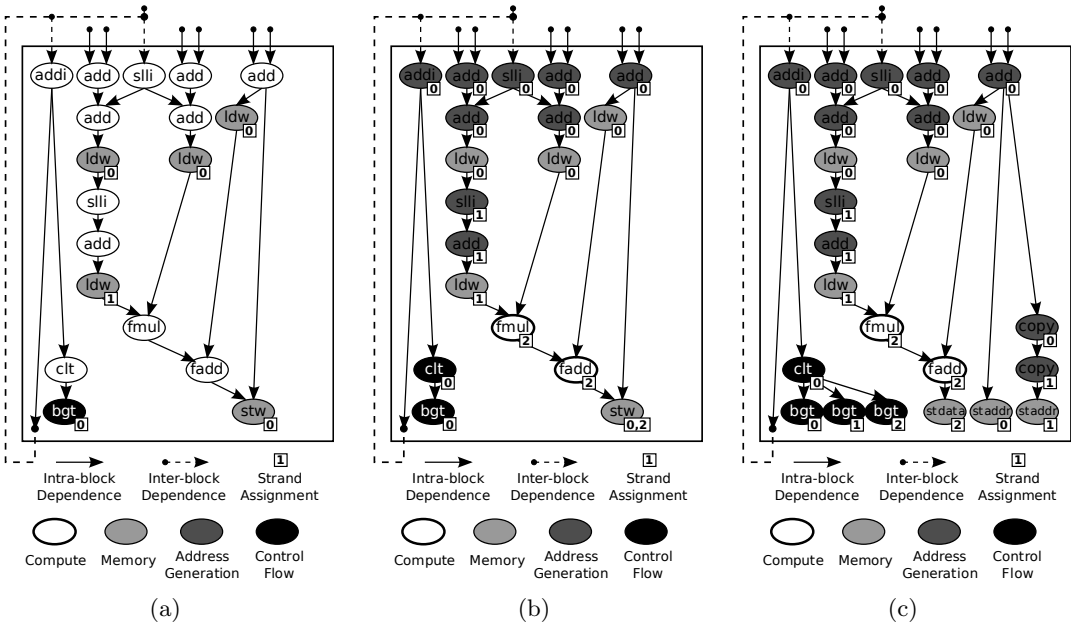

Figure 8: Diagram of the example code's dependency graph during phases of strand extraction process.

5. OUTRIDER STRAND EXTRACTION

OUTRIDER depends on compiler extraction of strands from the original thread by examining the dependency graph and partitioning the program into memory accessing and memory consuming instruction streams. For the purposes of this paper, we perform the partitioning process on leaf node functions which we assume make up the majority of execution time of data parallel applications. This simplies handling of parameter and return values, and enables different transformations on a function-by-function basis which allows additional flexibility. For simplicity during partitioning, we also assume that functions we transform output data directly to the memory system rather than returning a value.

Past research has demonstrated code partitioning and optimization [21, 26], and the approach OUTRIDER adopts is similar. In short, memory dependence chains are identified and strands are created along memory-access and memory-consumption lines. The process to extract strands consists of four phases of strand assignment: loads, stores, and control flow, unassigned instructions, final partitioning, and hardware mapping.

For the purposes of this paper, manual construction of strands is performed using the partitioning process presented. The partitioning process has successfully generated the benchmarks used in the evaluation section, which includes programs with memory-indirection and compute-generated addresses. Using prior research, we have made substantial progress on automated code generation and find that performance is within 6% on the `sobel` benchmark. Our automated code generation results are preliminary, but serve to provide evidence that automated code generation can provide performance comparable to manual construction. Complete details on automated code generation for OUTRIDER are outside the scope of this paper.

We provide an example of the strand extraction process using the inner loop of the sparse-matrix vector multiply of the `cg` benchmark. Figure 4.4 presents the partitioning process on the dependency graph of original inner loop in which the `addi` instruction acts as a loop counter which provides data for itself and the `slli` shift instruction used for calculating memory addresses.

5.1 Phase 1: Partition Loads, Stores, and Control Flow

Phase 1 partitions the load, store, and control flow instructions into their proper strands in order to achieve decoupling. Loads are partitioned into strands according to how many levels of loads are required to generate its address. Control flow instructions should ideally be serviced in strand 0, then communicated to all other strands. If the decision cannot be determined by strand 0, loss of decoupling will occur. Stores that may alias with loads must have their addresses calculated in strand 0 in order to prevent loss of decoupling caused during the distribution of `st_addr` instructions. Similarly to partitioning the load instructions, both the store and control flow instructions are partitioned by determining how many levels of loads must be traversed. At this phase, the maximum number of strands to be extracted is determined, which can potentially be more than hardware supports. In this case, Phase 4 reduces the number of strands during hardware mapping.

Figure 8(b) presents the result of the partitioning of Phase 1 on the dependency graph of the example loop from `cg`. As both the address of the `stw` and inputs of the `bgt` instructions do not depend on memory, they can be placed in the lowest level strand, strand 0. As these instructions exist in all strands, they are partitioned as such. There is a single level of indirection found in the loop, which results in the dependent load being assigned to strand 1.

5.2 Phase 2: Partition Unassigned Instructions

Phase 2 uses the identified loads, control flow, and stores and their assigned strands to identify and partition the address generation instructions. The back slice of instructions from a particular load, branch, or store in the dependency graph are considered. Only the instructions in the back slice found before reaching a load operation are considered for inclusion in the same strand as the initial instruction. As the backslice of some loads, branches, and stores inspected may

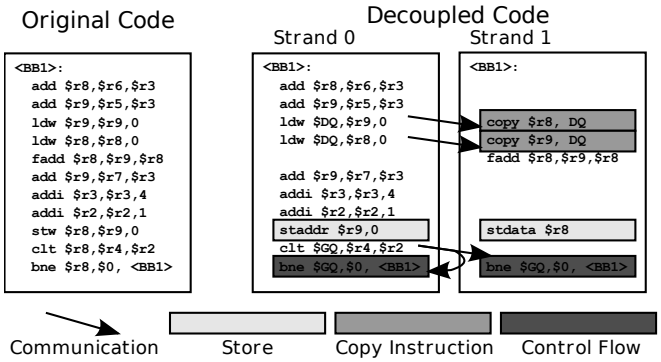

Figure 9: Code example from a scaled vector addition loop illustrating strand extraction.

share instructions, we give priority to the lower level strand. In the case that there are address generation instructions that are shared between strands, the value is communicated to the other strands in order to reduce circular dependences and promote decoupling. As a special case, the backslice of instructions providing data to store instructions are placed in the highest level strand.

Starting at strand 0, for each load, store, or control flow at the current strand level:

1. *Look backwards in the dependency graph, marking all unassigned instructions as belonging to this strand.*
2. *Terminate when an instruction has already been assigned.*
3. *Mark terminating instruction as also assigned to this strand*

5.3 Phase 3: Final Partitioning

With all instructions marked, strands are created. Figure 8(c) shows the final partitioning of instructions. Loads and their address generating instructions are included in their assigned strand, using the data queues to communicate their resulting values to the dependent strand utilizing copy instructions. Stores are split, with the address providing instruction `st_addr` assigned to strands 0 and 1 and the data providing instruction `st_data` assigned to strand 2, enabling the compute to pass the data directly to the MAU. Control flow instructions are assigned to their respective strand, with the result being broadcast to all the other strands. Branch instructions which source the globally-communicated decision are copied to all strands. Instructions that are shared among strands are placed in the lowest level strand, and communicated to other strands using copy instructions.

5.4 Phase 4: Mapping Strands to Hardware

During partitioning, more strands can be created than hardware has resources for. For example, a function with many levels of memory indirection may generate five strands, more than the four strands that OUTRIDER supports. In the case that too many strands are generated for the hardware to handle, we reduce the number of strands to the maximum size permitted by hardware by combining some of the strands.

In general, strands numbers adjacent to one another are considered for merging together. Specifically, we perform several passes using different priorities. During each pass, we reduce the number of strands by one. First we identify loss of decoupling events and merge those strands, as those strands have the least amount of performance improvement. Next, strands that contain few instructions incur extra overhead for communication and control flow, which can hurt performance efficiency. Finally, we choose the lower level strands as a last resort to reducing the amount of strands to the maximum allowed by the architecture.

After the code has been partitioned into strands, the compiler is responsible for generating setup code. The number of strands extracted and resource allocation information is written to a special purpose register and a command is issued which spawns the threads, at which point they begin fetching instructions from the initiating thread's PC. A jump table is used to direct the strands to the relevant code section they are to execute. Strands execute until the function is finished, at which point decoupled execution is turned off.

Core Base	8 stage, 2-wide in-order, 32 entry RF BTFN branch prediction, 8 entry BTB
OUTRIDER	4 strands, 32 entry shared RF, 32 entry data queue
Multithreading	8 entry instr. queue, 32 entry RF per thread
L1 ICache	2kB 2-way, 1 cycle, 2 Misses, Next-line Pref.
L1 DCache	1kB 4-way, 1 cycle 8 Misses, 8 Loads per strand/thread, 4 Stores
L2 Cache	64kB Shared. 4 cycle, 8-way
Interconnect	Two-level tree and crossbar, 16+ cycle latency
L3 Cache	4MB Shared, 32-Bank, 4 cycle, 8-way, Next-line Pref.
DRAM	8 Channels & GDDR5

Table 1: Simulation parameters for our 1024-core architecture.

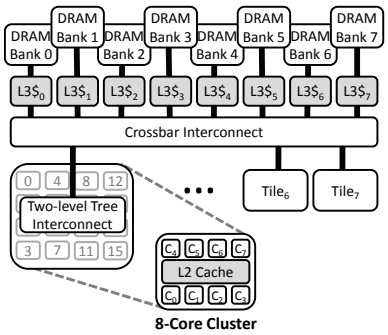

Figure 10: Evaluation architecture

5.5 Code Example

Figure 9 presents the inner loop of vector addition example code and its partitioning into strands. This code reads in two vectors, adds them, and stores the result. The original code is presented alongside the partitioned code, with corresponding instruction even between the two. During Phase 1, all the loads and stores are identified and heights recorded. The loads in this example are not memory dependent, and so they are assigned to strand 0. The store instruction is split across the two strands, with the address being provided by strand 0 and the data by strand 1. The control flow is placed in strand 0, where the result is communicated to the global queue. The result of this partitioning is that strand 0 can continue to execute and generate memory requests while strand 1 is still waiting on data from memory to perform the floating-point addition. As such, memory latency tolerance is exhibited.

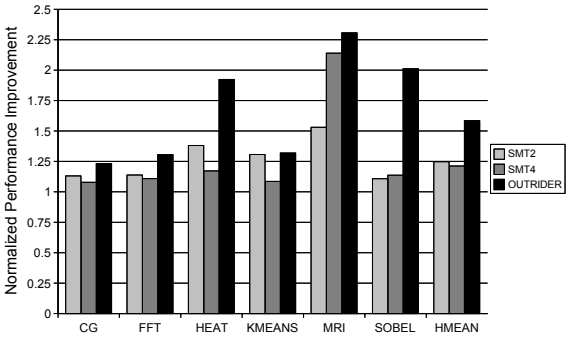

Figure 11: Overall performance of two-wide in-order baseline, two-way and four-way SMT, and OUTRIDER architecture relative to baseline.

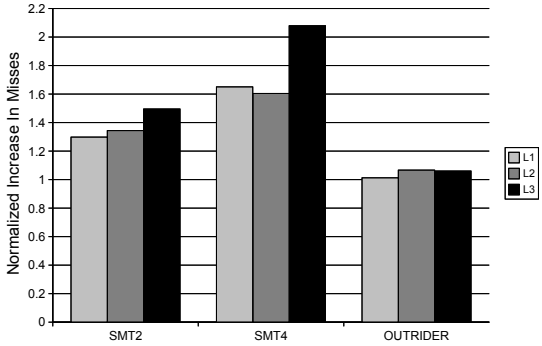

Figure 12: Increase in cache misses in two-way and four-way SMT, and OUTRIDER architecture relative to two-wide in-order. Harmonic mean across all benchmarks is presented.

6. EVALUATION

We evaluate OUTRIDER by comparing against traditional fine-grained simultaneous multithreading (SMT) [27]. We use the Rigel 1024-core throughput architecture shown in Figure 10 [10]. The cache hierarchy comprises three levels. Each core is a two-wide issue in-order with private L1 instruction and data caches and a RISC-like instruction set. Eight cores, an interconnect to a shared unified L2 cache, and an on-chip network interface form a *cluster*. The clusters connect to a multi-banked shared last-level L3 cache through a two-level interconnect network. Groups of four L3 cache banks share an independent GDDR memory channel. Table 1 lists the chip and core design parameters.

We evaluate our design on the Rigel simulator infrastructure, which is execution-driven and models cores, caches, interconnects, memory controllers, and DRAM. Each benchmark is executed for at least one billion instructions. For evaluation, we use a set of six optimized parallel kernels from scientific and visual computing applications. The benchmarks exhibit a high degree of parallelism and are written using a task-based, barrier-synchronized work queue model similar to Carbon [11], but implemented fully in software [10]. The benchmarks include conjugate gradient linear solver (`cg`), 2D fast Fourier transform (`fft`), 2D stencil (`heat`), k-means clustering (`kmeans`), medical image reconstruction (`mri`), and edge detection (`sobel`). Benchmarks are decomposed into strands manually as shown in Section 5.

6.1 Overall Performance

Figure 11 compares the performance of OUTRIDER to the baseline single-threaded core architecture, two-way and four-way SMT. Figure 12 shows the harmonic mean of the increase in misses observed at the L1, L2, and L3 caches. Two-way SMT improves performance by 25% over the baseline, while four-way SMT has mixed results due to a significant amount of contention at the L3 cache which counteracts potential performance gains. Four-way SMT performs best when contention is kept to a minimum, as in `mri`. The superior memory latency tolerance enables OUTRIDER to outperform two-way and four-way SMT significantly despite being a single thread. SMT can tolerate only a small amount of memory latency tolerance, while increasing contention for shared resources. OUTRIDER does not significantly increase contention for shared resources over the baseline and SMT cores despite executing the same memory stream on the same amount of cache resources.

The performance gains in OUTRIDER come from both the SMT interleaving of strands and the memory latency tolerance of strands. The interleaving effect is especially clear in `kmeans`, where performance is substantially improved using SMT. Memory-intensive benchmarks such as `cg` and `fft` do not see as much benefit from OUTRIDER. The lack of much performance improvement is due to extreme and irregular memory accesses that result in reduced utilization of cache resources, a performance limiter also found in the baseline. On memory-intensive benchmarks such as `heat` and `sobel`, which exhibit locality favorable to our cache hierarchy, OUTRIDER outperforms SMT by up to 87%.

6.2 Communication Queue and MAU Sizing

Figure 13 show the mean sensitivity of OUTRIDER to the size of the communication queues for all benchmarks. We design OUTRIDER with a 32-entry partitioned communication queue, and evaluate the sensitivity to reducing or increasing the number of communication queue entries. We observe that when data is written into the data queues, it is usually quickly consumed by a waiting strand. OUTRIDER requires a modest 32 entries to achieve nearly all of the performance benefit on the data parallel benchmarks. Our results for OUTRIDER demonstrate that our proposal can achieve good performance with 8 misses, 8 loads per strand, and 4 stores in the MAU, allowing for area- and power-efficient implementation. When scaled down to 4 misses, 4 loads per strand, and 2 stores the performance loss is only 4%, while increasing the size of MAU does not generally improve performance for our workloads.

6.3 Cache Latency and Size Sensitivity

Figure 14 shows the mean sensitivity to L2 cache latency across all benchmarks. We evaluate how increasing the access latency from a baseline 4 cycles affects the performance of inorder, SMT, and OUTRIDER designs. We find that OUTRIDER is not as sensitive to memory latency as the baseline and SMT cores; a two-way SMT processor with a 16-cycle L2 cache latency performs as well as OUTRIDER with 64-cycle L2 Cache latency. SMT cores can only tolerate a small amount of memory latency per thread, while OUTRIDER's

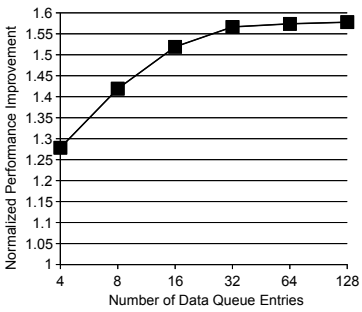

Figure 13: OUTRIDER **sensitivity study for data queue sizing.**

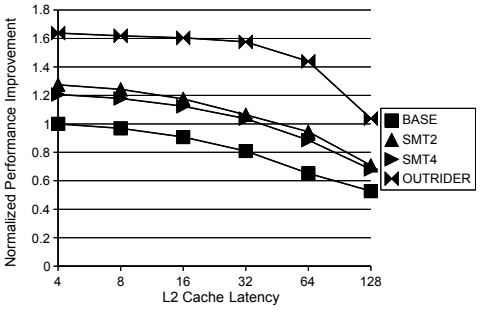

Figure 14: L2 cache latency sensitivity

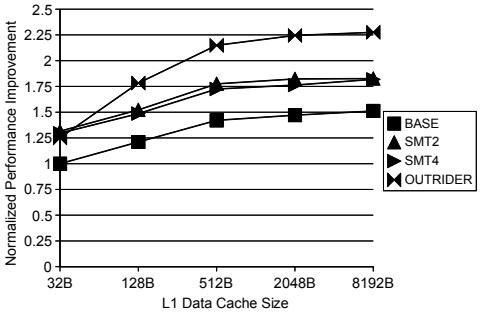

Figure 15: L1 data cache size sensitivity study.

explicit decomposition into non-blocking memory stands enables significantly more memory latency tolerance. In some cases four-way SMT outperforms two-way SMT as L2 cache latencies increase, as the additional memory latency tolerance provides benefit greater than the degradation due to L3 cache contention. In some benchmarks, such as `mri` and `sobel`, insensitivity exists up to 64 cycles without significant performance degradation. OUTRIDER begins to exhibit sensitivity to L2 latency at 32 cycles due to reaching MAU and communication queue capacity limits. Additional memory latency tolerance exists in the memory-accessing strands, and increasing the MAU and communication queue size allows additional insensitivity to latency.

Figure 15 shows the mean sensitivity of OUTRIDER and SMT to L1 data cache sizing for all benchmarks. We evaluate how reducing or increasing the cache size affects the performance of inorder, SMT, and OUTRIDERdesigns. As the size of the L1 data cache is increased, all processors see improvement. The increase in cache size decreases the amount of cache contention which improves SMT, but the addition storage space also benefits OUTRIDER and the baseline. OUTRIDER experiences a larger benefit to increased cache sizing, as the MAU and L1 data cache can be utilized more efficiently.

6.4 OUTRIDER Overheads

Table 2 presents the amount of added instruction overhead among the benchmarks. Instructions that copy shared data between strands, and control flow instructions needed to direct dependent strands make up the overhead. The number of memory operations and computation instructions remains the same. The harmonic mean of the instruction overhead is 14%, which compared with the 62% mean performance improvement is a performance-overhead advantage. Further reduction in the branch overhead can be achieved by traditional techniques such as loop unrolling. While the overhead can reach 38%, these instructions consume less energy as compared with memory and floating-point operations, which dominate the application's instruction stream.

We use Cacti 6.5 [16] to estimate the area cost for implementing OUTRIDER in a 45nm process. The added structures are the communication queues, offset tables, instruction queues, and the MAU. We also model the cost of adding threads for the SMT processor. The resulting areas are found in Table 3. For register files and communication queues we assume SRAM, while for instruction queues, MAU, and hardware tables we assume latches with size 8 μm^2 per bit including overhead. From Table 3, we see that the majority of area is taken up by the communication and instruction queues. Past work such as Rigel [10] is able to fit 1024 single-threaded cores in 320 mm^2 in a 45nm technology. The total extra area for OUTRIDER compared to such a system is 29.0 mm^2 or about 9.1%. SMT2 systems require 18.7 mm^2 (5.8%) and SMT4 systems require 51.7 mm^2 (16.2%). As such, we believe that from a performance per area perspective, OUTRIDER is a significant improvement over SMT.

7. RELATED WORK

In this section, we discuss other reduced-complexity processor architectures that leverage either hardware or software approaches to provide some level of memory latency tolerance. OUTRIDER relies on the compiler to extract parallelism instead of costly hardware structures and as a result does not require duplicated execution of memory access instructions or compute instructions as found in other designs. OUTRIDER is a proactive and non-speculative mechanism that provides both memory and functional unit latency tolerance through extracting up to four semi-independent strands of execution. Additionally, OUTRIDER leverages hardware and software techniques to minimize hardware and instruction execution overheads. While other designs require register files per thread or even processor fetch, decode and functional units to be replicated, OUTRIDER enables an area efficient design without this requirement.

7.1 Compiler-Enabled Techniques

VLIW and EPIC processors leverage the compiler to schedule instructions to avoid both functional and memory latency using loop unrolling and speculative code motion [6]. While these designs can remove the need for associative instruction windows, these designs require both large register files to hold in-flight values and memory disambiguation hardware. Additionally, loop unrolling and code motion techniques grow the code footprint considerably, which causes greater impact on accelerator architectures with small instruction caches per core. Even with speculative code motion, VLIW and EPIC designs can still be sensitive to memory latency and can stall if not enough software pipelining is done. OUTRIDER is a non-speculative technique that tolerates all levels of memory latency using completely separate streams of execution and hardware data queues which store only in-flight values from long-latency operations as opposed to every value.

	%Total	%Copy	%Branch
cg	14.59%	8.39%	6.19%
fft	37.82%	14.74%	23.08%
heat	14.10%	9.87%	4.22%
kmeans	24.37%	11.30%	13.07%
mri	6.05%	4.03%	2.02%
sobel	14.29%	13.17%	1.12%
hmean	13.57%	8.56%	3.15%

Table 2: Total instruction overhead for OUTRIDER compared to the baseline in-order design. OUTRIDER copy instructions and replicated branch instructions are also shown.

Item	Addition	ORA Area	SMT2 Area	SMT4 Area
MAU	8L(per extra strand/thread)x32	6,144 μm^2	2,048 μm^2	6,144 μm^2
Reg. Files	32x32 bit	None	12,071 μm^2	36,214 μm^2
Comm.Queues	32x32 bit	12,071 μm^2	None	None
Instr. Queues	8x32 bit	8,192 μm^2	4,096 μm^2	8,192 μm^2
CQ tables	(12) 4x5 bit	1,920 μm^2	None	None
RF tables	(4) 5 bit	20 μm^2	None	None
Total		28,347 μm^2	18,215 μm^2	50,550 μm^2

Table 3: Area overhead of OUTRIDER, SMT2 and SMT4 in regards to additional storage required.

Decoupled software pipelining (DSWP) is a compiler technique that creates parallel tasks from loops in sequential programs [18]. These pipeline-parallel tasks are mapped onto physical thread contexts in a CMP system, made up of either high-performance wide-issue out-of-order or VLIW/EPIC processors that already have some degree of memory latency tolerance. This is a different motivation than OUTRIDER, which targets highly parallel systems and applications on simple in-order processors. DSWP partitions based upon strongly-connected components in the dependency graph, while estimating the latency per instruction to combine these SCCs into the threads run on the processor. Following this partitioning scheme can result in memory dependences existing in a single thread which can lead to exposed latency on simple in-order processors. OUTRIDER assumes that variable-latency memory instructions are the most costly, and specifically partitions between memory access instructions and their consuming instructions to avoid exposed latency. DSWP is complementary to memory-latency tolerant techniques such as those found in OUTRIDER and can improve performance [20].

7.2 Pre-execution Techniques

Hardware scout threads were proposed to enable memory latency tolerance for in-order [3, 5] and out-of-order designs [17]. This is done be pre-executing the memory access stream. Hardware scout is not as effective for programs which have memory indirection. Flea-flicker two-pass pipelining [1] improves on hardware scout by adding a large instruction buffer to handle dependent memory operations and adds a result store buffer to enable the reuse of pre-executed instructions to combat data dependences. These schemes duplicate execution of the memory access instruction stream and only extract two ways of parallelism, and are sensitive to traditional prefetching concerns such as timeliness and accuracy of speculation. OUTRIDER does not require duplicate execution of the memory access stream and is not speculative, while having up to four strands of concurrent execution.

In-order continual flow pipelines (iCFP) [8], and Simultaneous Speculative Threading (SST) [4] allow execution to continue normally under a cache miss by deferring dependent instructions and their operands to a hardware queue. The deferred instructions are executed once the cache miss returns. This is an improvement over previous pre-execution work as no duplicate instruction execution is required except under a misspeculated branch dependent on a cache miss. However, memory disambiguation hardware is required in order to detect violations. OUTRIDER does not rely on adding large structures, such as large deferred instruction queues, or multiple checkpoints to provide memory latency tolerance. Another difference is that iCFP and SST spend overhead cycles fetching and decoding instructions only to defer them to the deferred queue. This is a reactive mechanism that can potentially waste issue slots that could be used for executing independent instructions. By using the compiler to partition dependent instructions into strands, independent work can potentially be uncovered more quickly. iCFP and SST also are limited to extracting only two stream of execution, while OUTRIDER extracts up to four.

7.3 Helper Thread Techniques

Slice processors [15] implement prefetching by dynamically extracting the memory miss instruction stream and then executing that stream in parallel with the main thread to prefetch data. When a miss occurs, the backslice of instructions is identified that caused the miss. The extracted stream can then be used to actively prefetch into the data cache. Like other prefetching techniques, accuracy and timeliness are not guaranteed and executing the prefetching instruction stream creates duplication of executed instructions. OUTRIDER is not speculative and does not prefetch data, nor does it require duplicate execution of the memory accessing stream. Slice processors require several large additional data structures, including a slice cache, an instruction stream slicer, and the candidate selector predictor table. OUTRIDER requires much more meager hardware overheads, only enough to buffer instructions and the data communicated between strands.

Helper threads [14] instantiate a partial thread of execution to improve the performance of the main thread. This thread is either programmer or compiler generated, and can either run completely independently or be controlled by the main thread. The main goal of the helper thread is to generate useful prefetches and warm up the data cache for the main thread. Similar to other prefetching techniques, helper threads are sensitive to timeliness and can cause cache contention and thrashing with the main thread if not properly controlled. OUTRIDER is not a prefetching technique and is not sensitive as helper threads are. Also, helper threads duplicate execution of the address generation stream while OUTRIDER does not.

7.4 Decoupled Techniques

Decoupled access/execute (DAE) provides memory latency tolerance by partitioning a program into two strands, one for executing memory instructions and one for executing compute instructions [24]. The programs are run on separate processors and to handle dependences between compute and memory, hardware queues are used for message passing communication. Later related work to DAE included investigating silicon implementations, code partitioning, strand balancing and memory latency tolerance limitations [7, 9, 12, 25]. While DAE enables parallelism and can allow the memory thread to provide memory latency tolerance, it is unable to handle memory indirection or compute dependent memory accesses which degrade performance. OUTRIDER utilizes additional strand parallelism to remove this performance degradation. Additionally, DAE requires in-order completion of memory accesses into the FIFOs, and restricts data to only be from loads or to stores. OUTRIDER enables out-of-order completion of messages and general data communication through the communication queues. Finally, OUTRIDER utilizes SMT to share fetch and execution resources and enable efficiency not found in DAE.

An alternative design, the Multiple Instruction Stream Computer (MISC), attempts to improve over DAE by executing up to four concurrent strands on separate processors [28]. However, this design does not ensure correct load-store ordering between strands and can only have two strands that access memory. MISC requires 24 statically-allocated hardware queues, and the efficiency of the MISC design is degraded if these queues or the separate processors cannot be fully utilized. OUTRIDER enables all four strands to access memory with correct memory ordering, which is enabled by a mixed hardware and software approach to memory aliasing detection. As with DAE, OUTRIDER utilizes SMT to share fetch and execution resources and enable efficiency not found in MISC.

Other contemporary decoupling work involves hardware partitioning and SMT [19]. In this work, the authors propose hardware partitioning of integer and floating-point instructions into separate threads in order to provide memory latency tolerance using large instruction queues to hold dependent floating-point instructions while they wait for the miss to return. These instruction queues needed can be more than an order of magnitude larger that those required for OUTRIDER, and this technique is limited to floating-point applications. Additionally, this technique suffers from memory indirection and compute-dependent memory accesses, which OUTRIDER supports. The technique also only supports SMT of different threads on either the EP or AP, unlike OUTRIDER which uses SMT across strands.

8. CONCLUSION

In this paper we present OUTRIDER, an architecture for efficiently tolerating memory latency in highly parallel workloads. The memory wall is making memory latency tolerance critical to scaling performance on future throughput-oriented processors. OUTRIDER has the goal of increasing the efficiency of throughput processors by decoupling the memory-access streams from the rest of the computation. Doing so allows for increased concurrency in the memory system with minimal additional cost over our in-order baseline micro-architecture and without additional thread contexts found in multithreaded architectures. We find that the key advantage OUTRIDER provides over previous decoupled access-execution architectures is the ability to continue decoupled execution when memory indirection and data-dependent control flow are present in applications. Our results comparing OUTRIDER to a conventional multithreaded architecture show that decoupling an instruction stream into strands can provide performance advantages of 23–131%. Our limit studies demonstrate that the hardware overhead of OUTRIDER structures relative to our in-order baseline can be modest and much lower than the cost of additional register files and increased cache sizing necessary to support more threads. The result is a micro-architecture for throughput processors that can provide memory latency tolerance while relying on a simple in-order pipeline and a lower number of explicit software threads.

9. ACKNOWLEDGEMENTS

The authors acknowledge the support of the Semiconductor Research Corporation (SRC). The authors thanks the Trusted ILLIAC Center at the Information Trust Institute for their contribution of use of the computing cluster.

The authors also wish to thank Steven S. Lumetta, John H. Kelm, Matthew R. Johnson, Daniel R. Johnson, William Tuohy, Wooil Kim, and the anonymous referees for their feedback.

10. REFERENCES

[1] R. D. Barnes, S. Ryoo, and W.-m. W. Hwu. "Flea-flicker" multipass pipelining: An alternative to the high-power out-of-order offense. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, 2005.

[2] P. L. Bird, A. Rawsthorne, and N. P. Topham. The effectiveness of decoupling. In *Proceedings of the 7th International Conference on Supercomputing*, pages 47–56, 1993.

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

[4] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous speculative threading: A novel pipeline architecture implemented in Sun's rock processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 484–495, 2009.

[5] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*, pages 68–75, 1997.

[6] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, 1994.

[7] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young. PIPE: A VLSI decoupled architecture. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 20–27, 1985.

[8] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating all-level cache misses in in-order processors. *IEEE Micro*, 30(1):12–19, 2010.

[9] L. K. John, V. Reddy, P. T. Hulina, and L. D. Coraor. Program balance and its impact on high performance RISC architectures. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, pages 370–379, 1995.

[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 *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 140–151, 2009.

[11] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 162–173, 2007.

[12] L. Kurian, P. T. Hulina, and L. D. Coraor. Memory latency effects in decoupled architectures with a single data memory module. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 236–245, 1992.

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

[14] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, 2001.

[15] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: An implementation of operation-based prediction. In *Proceedings of the 15th International Conference on Supercomputing*, pages 321–334, 2001.

[16] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 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] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, 2005.

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

[20] R. Rangan, N. Vachharajani, G. Ottoni, and D. I. August. Performance scalability of decoupled software pipelining. *ACM Transactions on Architecture and Code Optimization*, 5(2):8:1–8:25, 2008.

[21] K. D. Rich and M. K. Farrens. Code partitioning in decoupled compilers. In *Proceedings of the 6th European Conference of Parallel Processing*, pages 1008–1017, 2000.

[22] 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 International Solid-State Circuits Conference*, pages 56–57, 2009.

[23] 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 International Solid-State Circuits Conference*, pages 98–99, 2010.

[24] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, pages 112–119, 1982.

[25] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, and C. M. Rozewski. The zs-1 central processor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–204, 1987.

[26] N. Topham, A. Rawsthorne, C. McLean, M. Mewissen, and P. Bird. Compiling and optimizing for decoupled architectures. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, page 40, 1995.

[27] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[28] G. Tyson, M. Farrens, and A. R. Pleszkun. MISC: A multiple instruction stream computer. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 193–196, 1992.

[29] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. Rubio, F. Rawson, and J. Carter. Architecting for power management: The IBM POWER7 approach. In *IEEE 16th International Symposium on High-performance Computer Architecture*, pages 1–11, 2010.