ViReC: The Virtual Register Context Architecture for Efficient Near-Memory Multithreading

Matthew Barondeau
mebarondeau@utexas.edu
The University of Texas at Austin
Austin, Texas, USA

Sophia Jiang
sopjiang@utexas.edu
The University of Texas at Austin
Austin, Texas, USA

Jonathan Beard
jonathan.c.beard@gmail.com
Google
Austin, Texas, USA

Andreas Gerstlauer
gerstl@utexas.edu
The University of Texas at Austin
Austin, Texas, USA

Abstract

Near-memory processing can reduce latency and increase available bandwidth but requires extracting memory-level parallelism under strict operating constraints. Many memory-intensive workloads have low arithmetic intensity with often small register working sets and memory access patterns that cause frequent stalls and poor processor utilization. Latency hiding using instruction-level parallelism is limited for such workloads and can incur significant overheads. Instead, multithreading is widely used to hide stalls, but existing methods use large statically banked context storage that can remain underutilized while limiting the number of threads.

This paper presents the **Vi**rtual **Re**gister **C**ontext (ViReC) system architecture as a novel performance-area optimized approach for efficient, hardware-assisted dynamic context management and multithreading of memory-intensive workloads on near-data processors. ViReC virtualizes the register file and uses it as a cache for active partial register contexts. To manage the register cache state, ViReC employs a novel Least Recently Committed (LRC) replacement policy tailored to register access patterns of memory-intensive workloads. With ViReC, multiple contexts can be stored within a smaller physical register file that does put a static limit on thread counts where performance degrades gracefully as additional threads are scheduled. We evaluate ViReC on a range of near-memory benchmarks. Results demonstrate that ViReC achieves 95% of the performance of a banked processor while reducing area by up to 40% and scaling to higher per core thread counts.

CCS Concepts

• **Computer systems organization → Parallel architectures**.

Keywords

Near-Memory Processing, Multithreading, Caches

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '25, San Diego, CA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Matthew Barondeau, Sophia Jiang, Jonathan Beard, and Andreas Gerstlauer. 2025. ViReC: The **Vi**rtual **Re**gister **C**ontext Architecture for Efficient Near-Memory Multithreading. In *Proceedings of International Conference on Parallel Processing (ICPP '25).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In modern systems, in- and near-memory processors are employed to mitigate data movement costs, particularly for memory-intensive workloads [10]. These processors are commonly placed near the memory controller [8, 11], at each rank [55] or each bank [59] using function- or task-level offloading to leverage increased bandwidth and remove latency. However, only 20-30% of overall latency [54] is removed in these cases and the remaining latency must be hidden for high performance. This requires exploiting additional memory level parallelism (MLP) within strict resource constraints [23].

Conventional approaches to exploit MLP, such as out-of-order execution or prefetching, hide memory latency behind the execution of other instructions or by predicting future behavior. Such mechanisms can introduce large overheads [22] with limited performance gains [38]. As such, near-memory solutions typically focus on exploiting Thread Level Parallelism (TLP) instead [40], hiding memory latency behind other thread executions [10]. Latency hiding using TLP requires minimizing context-switching overheads, which is accomplished by storing complete thread contexts in on-chip register banks [6, 40]. However, banked register files can take up a significant portion of the processor area (up to 50%, see Section 6.2) where banks can remain unused while at the same time imposing a cap on the number of supported threads. Furthermore, given the small register working sets of memory-intensive workloads (see Section 2), registers within each bank are often underutilized. Overall, existing statically provisioned solutions suffer from resource inefficiencies and limited thread scalability.

In this paper, we present a novel approach to efficiently exploit TLP in near-memory processors by dynamically allocating and sharing physical registers between threads. We propose the **Vi**rtual **Re**gister **C**ontext (ViReC) system architecture as an area-performance optimized design for hardware-assisted thread context management. ViReC virtualizes the register file, dynamically managing partial register contexts from multiple offloaded threads concurrently. To manage the register state, ViReC employs a novel **L**east

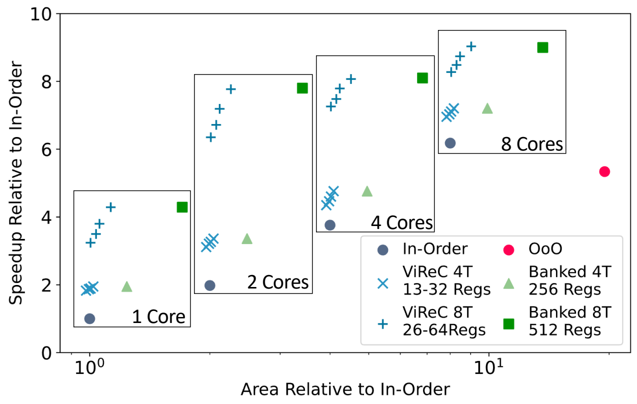


Figure 1: Performance-area tradeoff for a memory-intensive kernel with streaming indirect accesses (gather from [36]).

Recently **C**omitted (LRC) replacement policy tailored to register access patterns of memory-intensive workloads. ViReC optimizes area by caching active subsets from each thread in the register file and using the L1 data cache (dcache) and memory to store inactive state. This approach reduces the area overhead of multithreading and enables supporting high thread counts in a small register file.

ViReC differs from existing register caching schemes by tailoring optimizations to near-data constraints and workload properties. Existing register caches, such as in GPUs [3, 25, 45] use a small per-thread register file as a cache for a larger banked on-chip structure, or they rely on prefetching from an on-chip backing store. By contrast, ViReC realizes a hardware-managed register file shared across multiple threads that stores inactive registers off-chip. This approach requires managing more registers across multiple threads, targeting different thread scheduling and workload access patterns, and mitigating larger off-chip miss penalties. Other work [41] tackled a subset of these issues by designing a cached register file for general-purpose processors. However, their work did not target near-memory workloads and did not examine the impact of frequent thread interleaving on the replacement policy. Additionally, the tradeoffs associated with prefetching or other multithreading approaches are not compared against, nor is the area impact evaluated. ViReC addresses these points and presents a full system architecture while incorporating a novel replacement policy and miss penalty optimizations that exploit memory-intensive access patterns for high performance in a near-memory environment.
In summary, this paper makes the following contributions:

- We characterize register usage and access patterns within memory-intensive workloads running on a coarse-grain multithreaded processor.
- We propose the ViReC near-memory system architecture with a dynamically managed register file, controlled by a novel LRC replacement policy.
- We evaluate the ViReC architecture using the gem5 [13] AArch64 in-order core and demonstrate performance within 95% of a banked processor with only 40% of the area. We also show that LRC improves performance over existing policies by 21% and 7% at low and high register file contention.

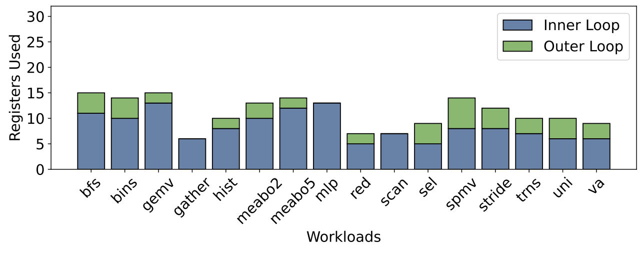


Figure 2: Memory-intensive workload register utilization.

2 Motivation

Figure 1 shows performance-area tradeoffs between different near-memory processing alternatives running a memory-intensive workload [36] that performs indirect array accesses. Details of the experimental setup and processor configurations can be found in Table 1 and Section 6. A single In-Order (InO) processor (based on the CVA6 [57]) (**gray**) incurs a small area overhead but has a limited ability to hide latency and frequently stalls, leading to low performance. An OoO processor (based on the Arm N1 [29]) can improve performance (by 5.3x in our evaluation) at significant area costs (19.1x higher) [43] (red). While the performance gains over a single InO are substantial, since instructions are routinely dependent on data from memory, there is limited ILP and a performance ceiling.

A near-memory solution can instantiate multiple InO processors to exploit TLP and, with 8 cores, achieve higher performance at better area efficiency than the OoO . However, each InO processor will be poorly utilized due to memory stalls. Multithreading can improve utilization within an InO processor by interleaving threads to hide stalls. Typical hardware-assisted multithreaded implementations use a banked register file (RF) (green) that stores the complete register contexts of multiple threads. However, this structure incurs large area overheads and can be poorly utilized when the number of threads needed to hide memory latency is smaller, or few registers are needed per thread. Conversely, if the workload needs more threads and thus more storage than is available, then performance will suffer. This behavior is shown in the gap between the banked approaches with 256/512 registers supporting up to 4/8 threads.

ViReC (blue) offers equivalent performance to banked approaches but within a smaller area by caching a subset of the registers. We compare configurations that store between 40 – 100% of the active context for 4/8 threads (13-32/26-64 registers). A configuration with 100% context has identical performance to a banked approach while reducing area by 40% and 42.4% for 4 and 8 threads, respectively. ViReC also enables a fine-grain trade-off between performance and area. For example, by storing only 80% of the active contexts, ViReC incurs a 3.3%/11% performance loss on a single core while saving an additional 1%/6% area for 4/8 threads. Further, a 40% context configuration only degrades performance by 7%/25%. Unlike banking, ViReC thread counts are not limited by available registers. Since performance degrades gracefully with increasing register pressure, ViReC allows exploiting thread scaling to enable performance gains with the same number of registers. For example, a configuration with 32 registers that supports 4 threads at 100% context can run 8 threads at 40% context with a 65% speedup.

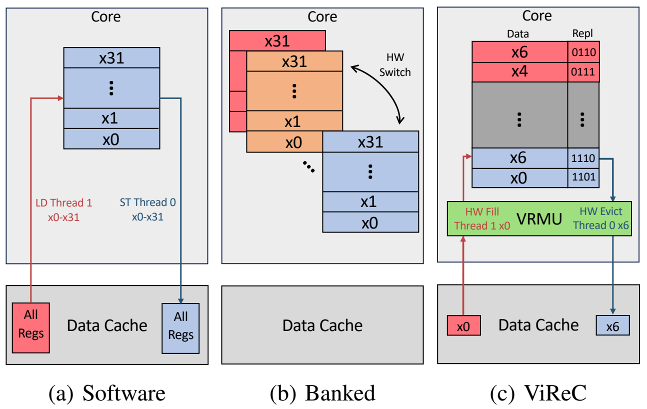


Figure 3: Software multithreading (a) loads and stores contexts from memory. Banked approaches (b) stores multiple contexts on-chip. ViReC (c) caches partial contexts and uses the VRMU to fill missing registers.

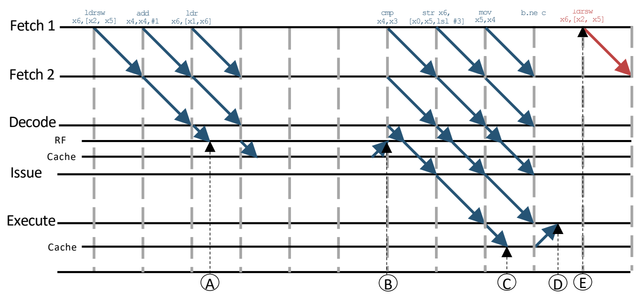


Figure 4: Instruction misses in the RF at Ⓐ result in a front end stall until the operand returns from the dcache at Ⓑ. Instructions that miss the cache at Ⓒ will result in a miss response at Ⓓ and a context switch Ⓔ.

ViReC leverages that active contexts of memory-intensive workloads are typically small. We examine several memory benchmark suites [1, 7, 28, 36] used in prior near-data processor evaluations [28, 30] and plot their register use in Figure 2. Many of these workloads use less than 30% of their register context within the innermost loop, where they spend most of their runtime. Additionally, a subset of the registers are used infrequently within the outermost loops, and storing them on-chip continuously gives little benefit. Thus, approaches that store only the active context on-chip can save significant area compared to storing multiple complete contexts. However, this hinges on determining the active registers and intelligently managing the on-chip storage.

3 ViReC Overview

ViReC is a Pareto-optimized coarse-grain multithreading (CGMT) architecture implemented on top of an InO processor that targets memory-intensive workloads. Similar to conventional CGMT, ViReC avoids long pipeline stalls by detecting misses in the dcache from memory instructions, flushing the pipeline, and scheduling a new thread using a round-robin approach. Unlike these approaches, however, ViReC saves area by only storing partial register contexts.

Existing register storage mechanisms have inherent inefficiencies in the area-performance space. Software context switches [2] (Figure 3(a)) require minimal area to only store the current context but introduce delays to save and restore register contexts on thread switches, which can exceed memory latency [19]. By contrast, a banked RF can hide switching delays by introducing additional hardware storage to store complete states for multiple threads (Figure 3(b)). These large register files are banked to decrease the access latency [18] and have the number of ports, supported threads, and the total number of registers fixed at design time. ViReC (Figure 3(c)), by contrast, caches actively used registers in a smaller RF, with the inactive registers being spilled to cacheable memory by the hardware. To accomplish this, ViReC adds a **V**irtual **R**egister **M**anagement **U**nit (VRMU) to map per-context architectural registers into physical registers or the backing store, along with logic to fill and spill registers to the backing store as needed.

Figure 4 shows an example ViReC operation. As instructions from the teal thread progress through the pipeline, they access the RF in the decode stage. This access can miss (Ⓐ), resulting in a request to the backing store (the dcache in our case) that stalls the pipeline until the register is filled at Ⓑ. If a load or store instruction needs to access memory, the request can result in a dcache miss (Ⓒ). The dcache will notify the processor (Ⓓ) to flush the pipeline and switch to a new (red) thread (Ⓔ), which will, in turn, trigger register replacements and fills to establish its context. Overall, ViReC performance depends on minimizing the impact of register misses and context switching time through a replacement policy that evicts the registers with the least performance impact, as well as through microarchitecture improvements to shorten the data miss detection and register fill cost.

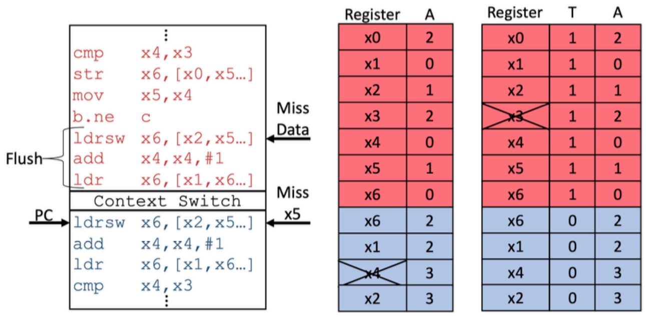
4 Register File Replacement Policy

In the following, we study access patterns of near-memory workloads and their implications on the design of ViReC and its register replacement policy. We examine inter- and intra-thread access patterns in memory-intensive workloads on a CGMT processor to determine reuse distances using the *gather* benchmark [36] as an example. We explain the shortcomings of existing policies and propose a **L**east **R**ecently **C**omitted (LRC) policy aimed at evicting the registers used furthest in the future, similar to Belady's min [12].

4.1 Inter-Thread Reuse

The order in which threads are run affects the order in which registers are accessed, and an informed policy must account for thread scheduling. With a round-robin scheduler, the next thread being switched to will always be the one that ran furthest in the past. A pseudo-LRU (PLRU) policy similar to those used in prior approaches [41] will wrongly evict registers from the oldest previous thread, which will be the thread currently executing or executing in the near future. Note that a LRU or FIFO policy, such as those used in prior approaches [25, 27], also suffers from the same problem.

Figure 5 shows a 2-thread example execution of the *gather* benchmark, where an PLRU policy thrashes the register context by relying



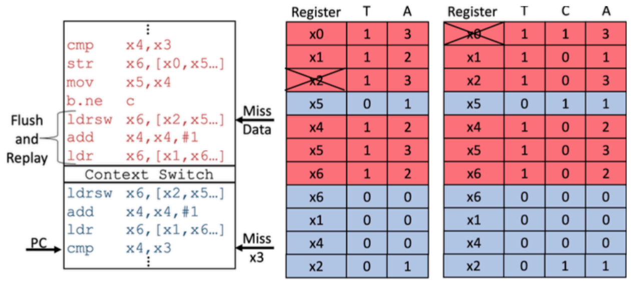
```
        ⋮
cmp     x4,x3
str     x6,[x0,x5…]
mov     x5,x4
b.ne    c
ldrsw   x6,[x2,x5…]
add     x4,x4,#1
ldr     x6,[x1,x6…]
   Context Switch
ldrsw   x6,[x2,x5…]
add     x4,x4,#1
ldr     x6,[x1,x6…]
cmp     x4,x3
        ⋮
```

Register	A
x0	2
x1	0
x2	1
x3	2
x4	0
x5	1
x6	0
x6	2
x1	2
~~x4~~	3
x2	3

Register	T	A
x0	1	2
x1	1	0
x2	1	1
~~x3~~	1	2
x4	1	0
x5	1	1
x6	1	0
x6	0	2
x1	0	2
x4	0	3
x2	0	3

(a) Instruction Trace (b) PLRU (c) MRT-PLRU

Figure 5: When missing in the register file (a), PLRU evicts registers from the upcoming thread (b), whereas MRT-PLRU evicts registers from the most recently run threads first (c).



```
        ⋮
cmp     x4,x3
str     x6,[x0,x5…]
mov     x5,x4
b.ne    c
ldrsw   x6,[x2,x5…]
add     x4,x4,#1
ldr     x6,[x1,x6…]
   Context Switch
ldrsw   x6,[x2,x5…]
add     x4,x4,#1
ldr     x6,[x1,x6…]
cmp     x4,x3
        ⋮
```

Register	T	A
x0	1	3
x1	1	2
~~x2~~	1	3
x5	0	1
x4	1	2
x5	1	3
x6	1	2
x6	0	0
x1	0	0
x4	0	0
x2	0	1

Register	T	C	A
~~x0~~	1	1	3
x1	1	0	1
x2	1	0	3
x5	0	1	1
x4	1	0	2
x5	1	0	3
x6	1	0	2
x6	0	0	0
x1	0	0	0
x4	0	0	0
x2	0	1	1

(a) Instruction Trace (b) MRT-PLRU (c) LRC

Figure 6: Within the thread (a), MRT-PLRU results in many saturated age values (b) while a LRC policy uses a commit bit (c) to differentiate register reuse distances.

only on age (A) bits for each register. When the red thread misses in the dcache when executing *ldrsw x6, [x2, x5..]*, a context switch occurs. Immediately after the switch, the blue thread will begin executing *ldrsw x6, [x2, x5..]*, which will attempt to access *x5* in the RF. This access will miss, and a PLRU policy would evict *x4* or *x2* (Figure 5(b)) as they have been used furthest in the past when the thread last executed. The PLRU policy is not optimal as the evicted registers will be used again shortly. Instead, the policy should evict from the red thread that will run furthest in the future.

To identify and target the most recently suspended thread, the replacement policy can add thread (T) bits that track the recency of the thread associated with each register. On a context switch, this field is set to the maximum for the stalled thread and decremented for every other thread. Then, when making eviction decisions, the policy can first evict registers from a recently suspended thread before those of an upcoming or current thread. We deem such a policy **M**ost **R**ecent **T**hread-**PLRU** (MRT-PLRU) and show that it correctly targets the furthest registers in Figure 5(c). Implementing MRT-PLRU requires adding a counter that tracks the distance of each suspended thread from the executing thread. MRT-PLRU concatenates the age and thread fields, with the thread field being the most significant, and selects the register with the highest value to evict. While ViReC using a round-robin scheduler only requires a simple counter to select the furthest thread, more complicated thread scheduling schemes can also be accounted for.

4.2 Intra-Thread Reuse

Beyond targeting the right thread, an optimized policy must also determine which registers to evict within the selected thread. In particular, workloads can have nested loops where some registers are accessed exclusively in the outermost loops, resulting in extremely long distances between subsequent uses. Such registers should be prioritized for eviction while the thread executes inner loops. While the replacement policy can identify and evict these infrequently used registers from the RF, compiler support can also remove them. A compiler can artificially reduce the registers available for register allocation to only those required in the innermost loops [24]. This register reduction will generate code that will spill outer loop values to memory using regular load/store instructions. As the outer loops run infrequently, the additional instructions constitute a negligible overhead (less than 0.1% in our experiments). Note, however, that this approach requires analysis of inner loop register usage to determine the proper compiler settings. In this work, we utilize compiler register reduction for outer-loop registers.

Among inner loop registers, the reuse distance of a register depends on whether it was part of an in-flight instruction when the context switch occurred. Since the processor flushes in-flight instructions on context switches, the thread will replay them when it resumes. This results in the flushed instructions providing the last and next accesses to the RF in that thread. Therefore, the replacement policy should retain registers of flushed instructions over those from committed instructions within the same thread.

A perfect LRU scheme for each thread within an overall MRT-LRU policy will approximate the eviction of committed registers. However, since the RF has much higher associativity than typical caches or prior work [25, 27], tracking perfect LRU ages across the entries would be prohibitively expensive. By contrast, using a PLRU policy with limited age ranges, all registers within the same thread that were accessed long ago will have the same maximum age values. This fuzzing of reuse distances, especially around the instruction that caused the context switch, can result in the eviction of flushed registers before committed ones. Alternatively, we can implement a simple commit (C) bit to differentiate flushed from committed registers. The C bit is initially set to zero and reset when a context switch flushes the register from the pipeline. It is set to one whenever an instruction using the register is committed. We deem the policy that augments MRT-PLRU a commit (C) bit as the **L**east **R**ecently **C**omitted (LRC) policy.

Figure 6 shows an updated example that includes each register's C bit and a two-bit age field implementing a PLRU scheme. In the example, the blue thread misses when accessing *x3* in its *cmp x4,x3* instruction. In a MRT-PLRU policy (Figure 6(b)), *x2*, *x5* and *x0* in the red thread would all have the same saturated age. However, some of these registers will be immediately needed once the red thread is scheduled again. By contrast, LRC in Figure 6(c) determines that *x0* is committed and will evict it, leaving the in-flight registers.

5 ViReC System Architecture

In this section, we describe the overall ViReC system architecture, shown in Figure 7. The system architecture consists of three main components: the VRMU placed within the decode stage, the context switching logic (CSL) in the fetch stage, and the backing store modifications in the dcache. ViReC also requires a small commit detection logic (CDL) in the commit stage to inform the other stages when an instruction has been completed. The CDL is implemented as a simple buffer and is not discussed in further detail here.

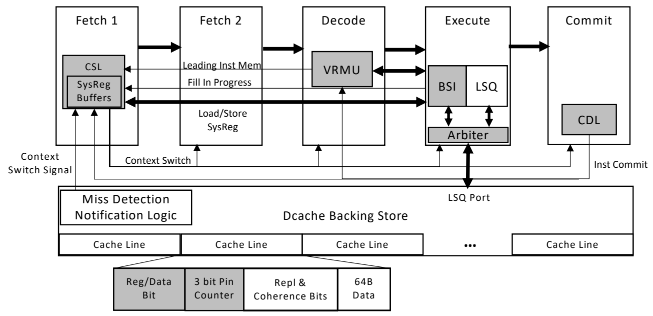


Figure 7: ViReC System Architecture with changes in gray.

5.1 Virtual Register Management Unit

Figure 8 shows the **V**irtual **R**egister **M**anagement **U**nit (VRMU) within a ViReC processor. The VRMU contains a tag store, rollback queue, and a slightly modified interface to the register file. In addition, the VRMU connects to the backing store interface (BSI) that is within the execute stage to load and store registers from memory.

Tag Store: The tag store consists of content-addressable memory that manages the mapping of register numbers to physical RF indices. When an instruction enters decode, the register portion is sent to the tag store and concatenated with the current thread ID to look up registers.

The tag store relies on internal Thread (T), Commit (C), and Age (A) replacement policy bits (3/1/3 bits, respectively) to determine which register to evict in case of a miss. The A bits are concatenated with the other fields to form a retention priority for all registers, with T being the most significant, followed by C and finally A. The registers with the highest value are evicted first.

Updating the T bits requires determining the registers from the currently executing thread. This is performed by comparing the thread IDs of each register, stored as an entry in the tag store, to a register containing the thread ID of the currently running thread. The T bits are set to zero for any register where these values match. On a context switch, the T values of any registers belonging to the previous thread will be set to the maximum while all others will be decremented and saturate at zero.

The C bit is set based on the commit status of the last instruction to access the register. Determining commit status requires tracking the registers that each instruction accesses and the instruction status. Rather than updating the C bit after every access and instruction commit, the VRMU speculatively initializes the C bit of each accessed register to 1, and when a context switch occurs, the rollback queue will reset the C bit of any in-flight registers to 0.

Finally, the A bits in the replacement policy are for pseudo-LRU, which uses three bits to determine the age of each register.

Rollback Queue: To update the C bit, the processor must determine which registers were used by in-flight instructions and were flushed from the pipeline on a context switch. These in-flight registers are identified and stored in a FIFO queue with a depth equivalent to the maximum number of instructions in the processor backend. After an instruction hits in the tag store, the register indices and a flag indicating whether the instruction is a memory operation are stored in the rollback queue. When an instruction is committed, a signal is sent to the queue to delete the oldest entry. On a context switch, the rollback queue compacts all the indices in the queue to a 1-hot vector that resets the C bits of the corresponding entries. In addition, the rollback queue forwards the memory instruction status bit of its oldest entry to the context switch logic (CSL), where it can prevent context switches.

Register File Interface: When all registers from an instruction are present (hit), accesses to the RF are identical to normal accesses. When registers are missing, accesses to the RF are divided into two parts: evicting selected register entries and inserting the missing registers. To evict registers, indices of the selected victims are sent to the RF, where the registers are read and forwarded to the BSI. If there are not enough RF read ports, this process is broken into multiple cycles. To insert registers, the BSI uses the RF's write ports to overwrite evicted entries, which can also be split into multiple cycles due to limited write ports. Rather than adding more ports, ViReC performs evictions and insertions in multiple cycles, as full evictions and insertions are rare.

5.2 Context Switching Logic (CSL)

ViReC incorporates a thread-switching mechanism within the fetch stage that consists of trigger logic with four inputs and a buffer for storing and prefetching system registers. The first input is from the dcache, indicating that a data miss has occurred and that the processor should switch to another thread. The other three signals, one from the VRMU, one from the BSI, and one from the commit stage, are used to mask context switching. The VRMU signal from the rollback queue determines if the oldest instruction is not a memory operation, which is used to ensure that long-running instructions issued before the memory instruction, and thus are not blocked by memory latency, are finished before executing a context switch. The signal from the BSI prevents context switching during an ongoing fill request to simplify fill logic. The final signal from the commit stage indicates whether at least one instruction has been committed since the last context switch to prevent ViReC from cycling through threads when unable to cover memory latency.

ViReC stores system registers of each thread in the backing store alongside the general purpose registers. Unlike the general-purpose registers, these system registers must be loaded every time a thread executes. Rather than fetching on demand, ViReC prefetches and stores the system registers for the current and next threads in a ping-pong buffer. The buffer is switched when a context switch occurs. In parallel to fetching instructions from the new thread, the buffer contents of the previous thread are written to the backing

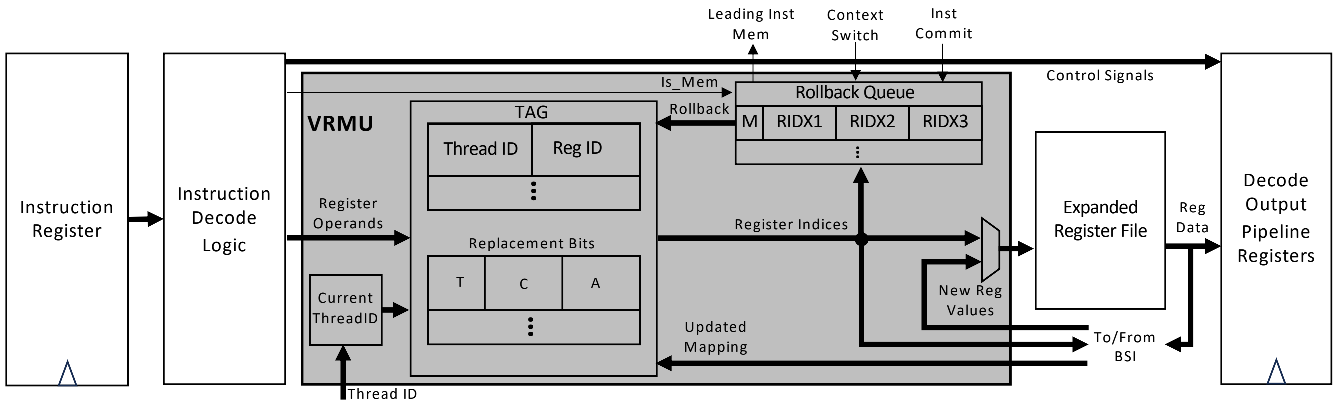


Figure 8: Virtual Register Management Unit (VRMU) integrated into the decode stage.

store and system registers for the next thread are prefetched. These accesses overlap the pipeline warmup and limit contention for the dcache port between system registers and program loads.

5.3 Backing Store and Backing Store Interface

In this section, we describe the changes to the dcache backing store and the backing store interface (BSI) within the pipeline.

Backing Store: The dcache is a backing store containing program data and register values. To retrieve registers from the dcache, the BSI issues a request just like a normal load through the LSQ port. Access to the LSQ port is controlled through a simple arbiter that always prioritizes LSQ requests over register requests.

Registers are stored as regular data values with eight registers per 64B line. Cache lines are augmented with additional meta-data to determine if the line contains registers or data and for pinning of register lines. Furthermore, the miss detection logic in the dcache is expanded to send an additional signal to the CSL whenever a load request for data through the LSQ port misses in the tag array. Since the LSQ and BSI share a port, an address computation is performed to check if the missing data is within the reserved register region. If within the region, no context switch signal is sent and the processor waits for registers from memory.

Unlike program data, register lines are pinned while registers within the line are alive in the RF. This pinning accelerates register fills and spills and relies on spatial locality to keep evicted registers within the cache, but requires that a portion of the dcache be used for storing registers (Section 6.1). Pinning is implemented using a 3-bit counter per line alongside a register/data bit that determines if the line should use or ignore the counter. The counter is incremented when a read to a register line occurs, indicating that a register is now alive in the RF. The counter is decremented on a write when a register is evicted from the RF.

Backing Store Interface: On a RF miss, the BSI reads registers from and writes evicted registers to the dcache. To fill registers, their indices and the ID of the current thread are concatenated together, added to an offset to the start of a reserved memory region, and the BSI issues a load to this location. For evicted registers, the BSI generates an address by storing a local copy of the thread ID and register indices for all entries currently in the RF and indexing into this table using the RF indices. The BSI prioritizes loads for register fills over stores for evictions as loads directly affect program performance. Furthermore, when a register load or store is outstanding, the BSI signals the CSL to block context switches to prevent the eviction of registers that are being retrieved.

The BSI implements an optimization for any registers that are destination operands of an instruction as these registers do not need their old values for correct execution. In these cases, i.e. for any destination registers to be filled that are not also source operands, the BSI can write a dummy value to the RF. The BSI will still issue a transaction to the backing store for meta-data bookkeeping, but the backing store latency is removed from the critical path.

The BSI can be implemented in a blocking or non-blocking manner. A blocking BSI will issue one request to the dcache at a time and wait for that request to return before issuing a second. This is area-efficient but slow for evicting multiple registers. A non-blocking BSI will allow issuing multiple pipelined requests to the cache to increase throughput and partially hide the backing store latency. We implement a non-blocking BSI in our evaluation.

6 Evaluation

We examine ViReC area and delay tradeoffs alongside performance implications. We use a combination of CACTI [39] and synthesis targeting 45nm [51] for area and delay evaluation, and gem5 [13] for full-system performance simulation with processor configurations shown in Table 1. The multithreaded processors implement a modified AArch64 InO architecture that we attach to the system crossbar near the memory controller, similar to the configuration in [8, 11]. Each processor has a small icache and dcache but no L2.

We run several suites of memory-intensive workloads that contain different access patterns and arithmetic intensities from prior work to evaluate near-data processors [1, 7, 28, 36]. These workloads originate on an OoO processor and are dispatched to one or more near-data processors using a task-level offload mechanism [8], where workload contexts are shipped through the crossbar and written to a reserved region of memory per processor. The near-memory processor is then notified and will begin fetching the register contexts when the thread is scheduled.

Table 1: Performance simulation parameters.

Processor	Out-of-Order [29]	In-Order [57]	ViReC	Banked Core
Core	2GHz 8-wide issue (2-LD, 2-FP/VEC, 4-ALU) 384/384 Phys Int/FP Regs, 224 ROB Entries 113 LQ 120 SQ Entries	1GHz single-issue 32/32 Int/FP Regs 5 SQ 2 outstanding LD	1GHz single-issue 24-120 Regs 5 SQ 1 outstanding LD	1GHz single-issue 8 Banks 32/32 Int/FP Regs 5 SQ 1 outstanding LD
Caches	64kB 4-way I-cache (2-cycle access) 1R, 1W port 32kB 4-way D-cache (4 cycle access) 2R, 2W ports, 32 MSHRs 1MB 8-way L2 cache (12 cycle access), 64 MSHRs, Stride Prefetcher, degree 8	32kB 4-way I-cache (2-cycle access) 1R, 1W port 8kB 4-way D-cache (2 cycle access), 1R, 1W port, 24 MSHRs		
Memory	DDR5_6400 1 rank, 2 channels, tRP-tCL-tRCD: 14-14-14			

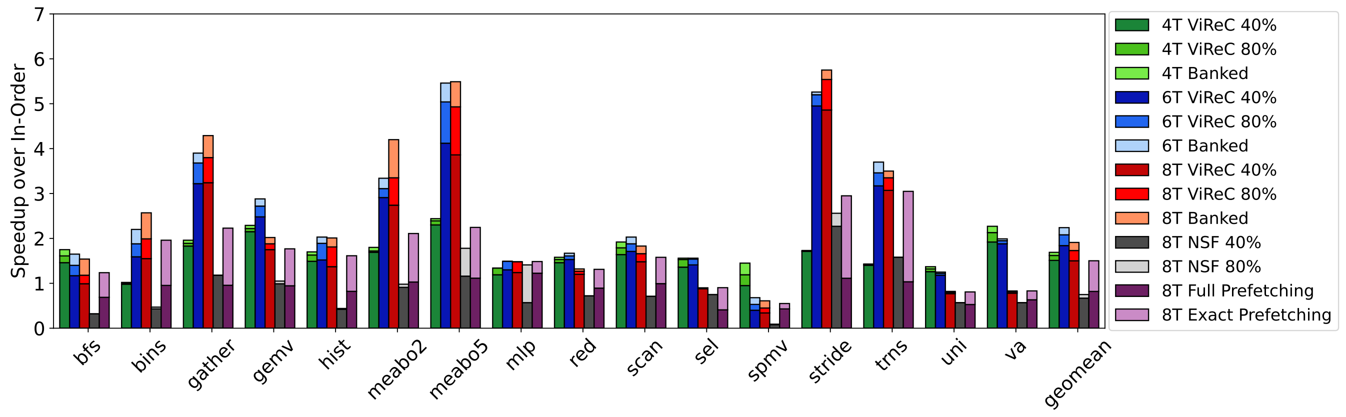


Figure 9: Performance comparison of a ViReC processor, a banked processor, and a banked processor with RF prefetching.

6.1 Performance Evaluation

We evaluate performance for a single processor running 4, 6, or 8 threads of a given workload. To model RF contention in ViReC, we sweep the RF size from storing 80% of the active workload contexts (low contention) down to 40% of the register contexts (high contention). We also compare ViReC to RF prefetching and evaluate ViReC in systems with increased activity.

Performance comparison: As shown in Figure 9, ViReC's performance degrades gracefully compared to a similarly-threaded banked solution under increasing RF contention. ViReC shows a mean performance drop of 4.4%, 7.1%, and 10% for 4, 6, and 8 threads when storing 80% context, and of 10.7%, 17.6%, and 22.1% when storing 40% of the context within the RF. In these cases, ViReC trades off area savings for reduced performance.

We compare ViReC's performance to the NSF from [41] and a prefetching approach. Overall, ViReC improves performance by 133% and 125% over the NSF at 80% and 40% context, respectively. This is due to reduced RF misses from the LRC policy (see Figure 12, 14%/6% higher hit rate vs. PLRU resulting in 21%/7% better performance) and lower register miss penalties from improvements like the BSI and register pinning.

Prefetching contexts into just two banks used as double buffers presents an alternative area-efficient approach to either banking or caching [45]. We evaluate two strategies in Figure 9, where we prefetch either the full context or the exact needed context (assuming an oracle prediction) of an upcoming thread while the prior thread is executing. Due to the short runtime between context switches (as short as 15 cycles) for many of these workloads, loading the full context between executions is expensive. All registers from the previous thread must be stored in memory, and all upcoming registers must be loaded into the RF. As a result, prefetching the full context is almost always worse than a caching approach, regardless of the size of ViReC. Prefetching the exact set of needed registers reduces the performance penalty compared to ViReC . Under high contention (ViReC 40% context), exact prefetching results in a mean performance improvement of 3.3% over ViReC . However, when ViReC can store 60% or 80% contexts, exact prefetching results in 4.7% and 10.6% lower performance, respectively. This is due to exact prefetching still needing to load and store all used registers each time the thread runs, while ViReC can maintain a subset of the registers in the RF between context switches. In workloads with frequent switching (*maebo*, *gather*, *stride*), loading registers each context switch is significantly slower. Furthermore, exact prefetching requires predicting the next-used registers and introduces metadata storage per thread to determine these registers at runtime, which places an upper limit on the number of threads. Thus, a caching approach allows better performance and enables thread scaling.

ViReC scaling: Figure 10 examines the performance tradeoff when scaling the number of registers and threads running on a ViReC processor compared to a banked approach. We plot the performance per register for a single processor running *gather* and sweep the number of scheduled threads. Each thread configuration has four points corresponding to the 40%, 60%, 80%, and 100% context storage for ViReC , along with a banked solution.When memory latency

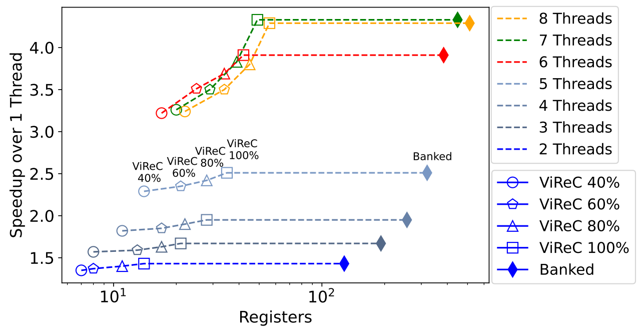


Figure 10: Performance-per-register tradeoff for Gather [36].

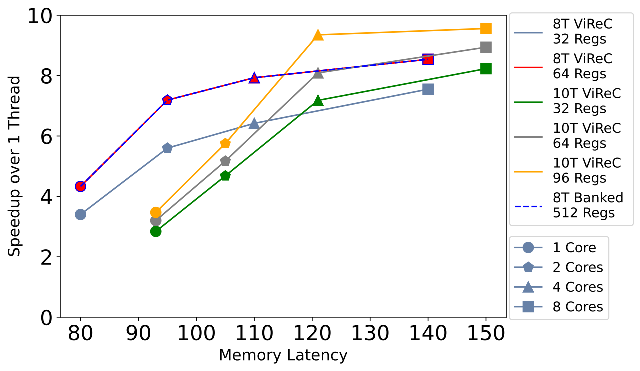


Figure 11: Performance scaling with increased system load.

is not fully hidden (few threads are scheduled), register misses from ViReC have little performance impact since they overlap with memory latency. In these cases, ViReC offers comparable performance to a banked processor even with less than 100% context storage. Once memory latency is hidden with more threads, there is a steeper performance improvement from storing additional per-thread context. Thus, it is better to rapidly increase the number of scheduled threads with a small per-thread context until memory latency is hidden. Once latency is hidden, RF capacity is better spent reducing register misses than scheduling more threads.

ViReC can offer area and performance benefits over banked solutions when few threads are needed or when faced with a high system load. In workloads with high cache line locality or arithmetic intensity, memory latency is hidden with fewer threads, and ViReC can store full contexts with equivalent performance while saving area. When running workloads that have high cache miss rates and frequent switching (*gather*, *maebo*, *stride*), ViReC can schedule additional threads by reducing per-thread register storage. Banked solutions would require additional banks of registers or hierarchical scheduling to cover this latency. By contrast, ViReC can reduce per-thread registers to enable scheduling additional threads and achieve higher performance than a banked solution.

Figure 11 examines this tradeoff between scheduling additional threads and reducing per-thread register context. We instantiate systems with 1, 2, 4, or 8 processors executing the *gather* benchmark and measure the observed latency for systems running 8 or 10 threads. With only 1 or 2 active processors in the system, memory contention is relatively low, and only 8 threads are needed per processor to hide memory latency. However, as observed latency increases due to system activity, additional threads are required to hide latency. In these cases, 10 threads perform best for the 4 and 8 processor configurations. A statically provisioned banked processor would require two-level scheduling or waiting until threads are completed before launching additional threads. By contrast, ViReC can schedule additional threads and improve performance given a sufficiently sized but much smaller RF. Thus, depending on system and workload behavior, ViReC can be configured to mimic or exceed the performance of a statically provisioned banked processor.

Replacement policy evaluation: Figure 12 compares different replacement policies for the register cache on a single ViReC processor with eight threads and different context sizes. This includes LRC (Section 4), MRT-PLRU, and PLRU as used in prior work [3, 41] and their perfect (non-pseudo) variants.

Policies that account for thread scheduling perform significantly better than the scheduling-oblivious LRU variants. Compared to PLRU, MRT-PLRU improves the hit rate on average by 14.4% and 4.6% at 80% and 40% context sizes. The largest gap occurs with larger RFs as PLRU will thrash irrespective of the RF size, while MRT-PLRU can preserve partial contexts for each thread. For some workloads, (*maebo*), subsets of each context are accessed each time the thread is run. PLRU benefits from fewer registers being accessed but overall performs worse as these workloads have high temporal register locality between the partial executions. A perfect LRU policy still results in RF thrashing, and similar performance, due to the round-robin scheduling. By contrast, MRT-LRU improves performance over LRU by 16.5% and 4.3%. When comparing among scheduling-aware policies, MRT-LRU improves the RF hit rate over MRT-PLRU by 2.8% and 3.2% at 80% and 40%, respectively, but requires perfect commit information. LRC performs within 0.3% of MRT-LRU and has a 2.6% and 3.2% higher hit rate than MRT-PLRU with average hit rates of 93.9% and 82.9% at 80% and 40% context.

Overall, LRC improves mean speedup by 20.7% and 7.1% over PLRU at 80% and 40%, respectively. Similarly, LRC improves mean speedup over MRT-PLRU by 3.6% and 2.7%. This indicates that the commit bit is a good feature for differentiating reuse within threads.

Backing store sensitivity: We further examine how backing store characteristics affect ViReC. We sweep the dcache latency in a system with a single processor running 8 threads and show the IPC geometric mean across all workloads in Figure 13. In general, all approaches see a performance loss as the dcache latency increases. Compared to a banked approach, ViReC performance degrades more quickly due to additional fill latencies. However, for reasonable dcache latencies, register fills have a limited impact.

Figure 13 also shows the effects of reduced dcache capacity with a single processor running 8 threads. Since our implementation of ViReC uses the dcache as a backing store, each thread uses between 2 and 4 cache lines to store their general and system registers. In ViReC , these lines are pinned (Section 5.3) so they cannot be evicted, which results in increased contention for the dcache capacity that is not present in a banked solution. As cache size decreases, ViReC thrashes the dcache earlier than a banked processor. This results

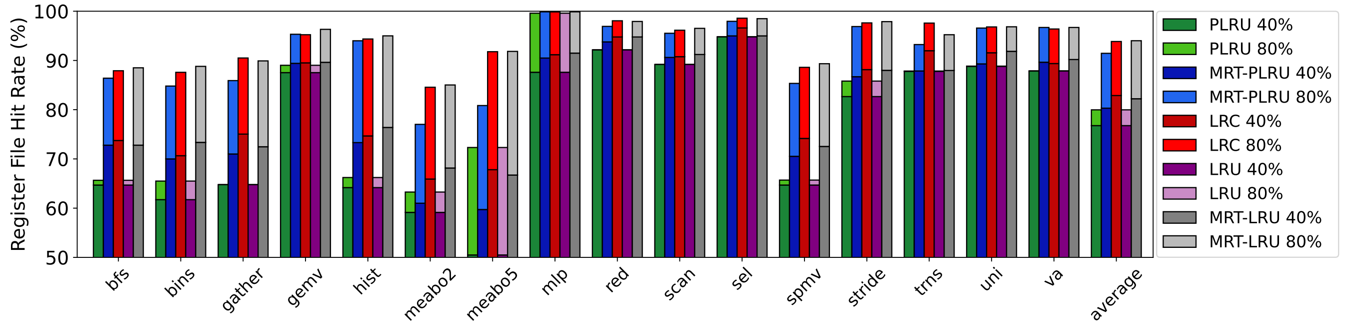


Figure 12: Register replacement policy hit rate.

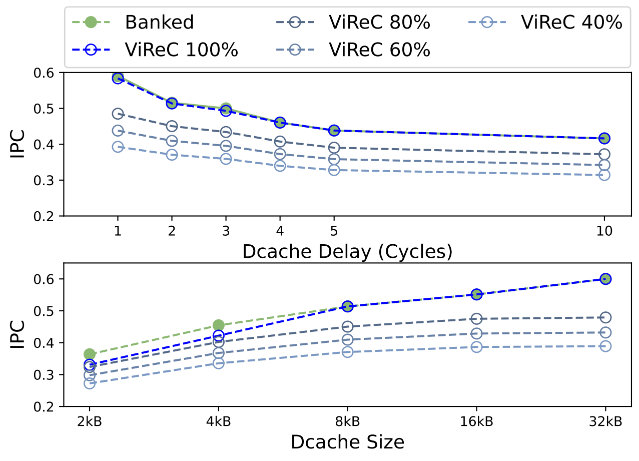


Figure 13: Data-cache latency and capacity sweep.

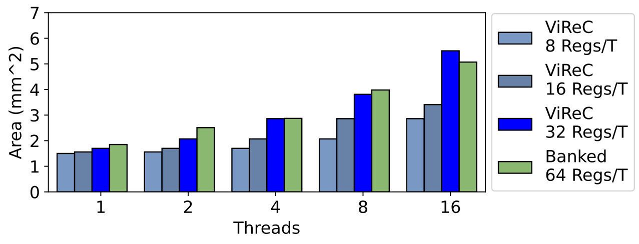


Figure 14: ViReC and banked processor area comparison.

in additional context switches due to data misses, and, for some workloads with multiple loads within the innermost loop, ViReC begins executing fewer instructions each time the thread runs. At the same time, we observe that many workloads have limited temporal locality, and for workloads with a single load per loop, there is no noticeable performance overhead for a smaller cache compared to a larger one.

6.2 Area and Delay Evaluation

We examine ViReC area for a baseline implementation in a RISC-V InO core from [57] scaled to 45nm [50]. To determine ViReC overhead, we synthesized a RF, rollback queue, and VRMU logic in 45nm technology [51]. Since our libraries did not include an optimized CAM cell, we used CACTI to determine the area and delay for the tag store and an RF identical to our synthesized configuration. We then compared the synthesized RF against the CACTI estimates and scaled the CACTI results into our 45nm synthesis.

Our analysis found that most of the area overhead is from the VRMU tag store and RF. The rollback queue and other VRMU logic constitute less than 10% of the RF size and scales more slowly. Figure 14 shows the processor area versus the number of threads for a ViReC approach with variable register context sizes per thread alongside a banked approach with 64 registers per bank. As the number of registers increases, the ViReC area grows much more rapidly than a banked approach due to the poor scaling of the tag store. Thus, storing large or complete contexts in a fully associative cache will require more area than banked RFs. However, as memory-intensive workloads require fewer registers per thread (on the order of 5-10 registers at 100% context, see Figure 2), ViReC can achieve higher performance per area. For example, with 8 or 16 threads, a banked core will require an area of 2.8-3.9 mm^2, while a ViReC core with 8 registers (80-100% context) per thread requires only 1.7 mm^2. In this case, ViReC incurs an overhead of 20% over the baseline core and offers up to 40% area savings over a banked design.

Similar to area, RF access delays in ViReC start lower but grow more quickly with additional registers than in a banked approach. However, a ViReC configuration with 80 registers only adds around 10% delay overhead (0.24 ns vs. 0.22 ns) compared to a baseline core. This is equivalent to the delay of a similarly threaded banked core.

7 Prior Work

Several near-data solutions have been proposed by academia and industry. Industry implementations of near-data processors vary from embedded logic blocks [37] for computing SIMD operations on DRAM rows or simple cores with hardware multithreading [28]. Academic work has begun to investigate methods to increase utilization [30]. However while a few works have looked at the design of the near memory processor [20], most work has focused on application mapping [9, 16, 55] or communication with other hosts [15].

When looking at RF caching, the closest prior work to ViReC is the Named State Register File (NSF) [41]. The NSF was designed without considering the system architecture and it did not target memory-intensive workloads. The design used a PLRU policy to manage registers, but this policy trashes the RF for a multithreaded environment. Other RF caching approaches aim to reduce access costs in both CPUs [34, 48, 56, 58] and GPUs [21, 26, 27]. These approaches focused primarily on caching in a small RF to hide latency of large RFs. As a result, their miss penalties and area tradeoffs differ as the entire context is still stored in a second-level RF. Other approaches share multiple contexts within the RF without caching by shifting offsets between threads [53] or virtualizing registers to memory [42], but these works do not target maintaining a partial context.

There are many existing cache replacement policies [32]. We evaluate against policies that can be implemented within the processor pipeline and do not require table lookups to predict reuse distance [31, 47] or have large overheads to generate predictions [46]. Other policies [33, 44] sample cache sets to determine whether cache items are recency-friendly or averse based on prior access, which does not work for registers as the reuse distance depends on the instruction and context switch behavior.

Some works [45] have examined GPU RF prefetching, but these approaches do not face the same latency requirements or threading model as ViReC. However, we do compare against an oracle prefetching mechanism and show that, for RF latencies in a near-memory environment, register prefetching performs worse than a well-informed replacement policy. Other work prefetches data into unused registers in the GPU [35]. This requires that inactive register state be available on-chip, whereas ViReC stores unused registers off-chip and can reduce the overall RF area. Still other work [49] prefetches data from the data caches into physical registers to minimize OOO backend latencies and accelerate critical chains of instructions. Similar to [35], this requires that there be available physical registers and is targeted at a different class of latencies than our prefetching evaluation.

There are many multithreading solutions [4–6, 14, 17, 52] that target many application domains. Similar to our approach, these solutions aim to improve pipeline resource utilization with reduced area overhead compared to duplicating discrete processors. However, to the best of our knowledge, no paper has examined multithreading tradeoffs for near-data processing and addressed area-performance tradeoffs in the presence of memory-intensive workload behavior.

8 Summary and Conclusions

In this paper, we presented the ViReC system architecture as a Pareto-optimized solution for multithreading of memory-intensive workloads in near-data processors. ViReC virtualizes the register file and dynamically manages partial register contexts through a VRMU, which employs a novel LRC replacement policy. We evaluate the LRC policy against prior PLRU and LRU policies and demonstrate up to a 20.7% performance improvement. Additionally, we find that an LRC can improve performance over an oracle register prefetching mechanism by up to 10.6%. In a single processor configuration, ViReC performs within 95% of a banked processor when facing RF contention while using up to 40% less area. Furthermore, ViReC can adapt to varying system and workload behavior by supporting thread scalability by reducing per-thread register storage for higher thread counts or by reducing thread counts to improve per-thread performance. This flexibility allows ViReC to improve performance over a banked solution when higher thread counts are required and improve register utilization at lower thread counts. T Finally, we show the area impact of ViReC and the system impact of storing partial contexts alongside program data in the dcache.

Overall, VIREC provides area-efficient storage of thread contexts and a Pareto-optimized solution for accelerating memory-intensive workloads in near-data processors. In future work, we plan to explore improved replacement policies for group evictions and combinations of prefetching with ViReC caching.

References

[1] The coral-2 benchmark suite.

[2] J. Aas. Understanding the linux 2.6. 8.1 cpu scheduler. *Retrieved Oct*, 16, 2005.

[3] M. Abaie Shoushtary, J.M. Arnau, J. Tubella Murgadas, and A. Gonzalez. Lightweight register file caching in collector units for gpus. In *Workshop on General Purpose Processing Using GPU*, 2023.

[4] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: a processor architecture for multiprocessing. In *International Symposium on Computer Architecture*, 1990.

[5] H. Akkary and M.A. Driscoll. A dynamic multithreading processor. In *International Symposium on Microarchitecture*, 1998.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, 1990.

[7] Arm. Meabo, 2018.

[8] M. Asri, C. Dunham, R. Rusitoru, A. Gerstlauer, and J. Beard. The non-uniform compute device (nucd) architecture for lightweight accelerator offload. In *Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, 2020.

[9] R. Balasubramonian. Near data processing. In *Innovations in the Memory System*. Springer, 2016.

[10] R. Balasubramonian, J. Chang, T. Manning, J.H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4), 2014.

[11] A. Barredo, A. Armejach, J. Beard, and M. Moreto. Planar: a programmable accelerator for near-memory data rearrangement. In *International Conference on Supercomputing*, 2021.

[12] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 1966.

[13] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[14] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6), 2000.

[15] Benjamin Y Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. Near data acceleration with concurrent host access. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020.

[16] B.Y. Cho, J. Jung, and M. Erez. Accelerating bandwidth-bound deep learning inference with main-memory accelerators. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[17] J. Choquette. Nvidia hopper gpu: Scaling performance. In *Hot Chips Symposium*, 2022.

[18] J. Cruz, A. González, M. Valero, and N.P. Topham. Multiple-banked register file architectures. *SIGARCH Comput. Archit. News*, 28(2), may 2000.

[19] F.M. David, J.C. Carlyle, and R.H. Campbell. Context switch overheads for linux on arm platforms. In *Workshop on Experimental Computer Science*, page 3–10, 2007.

[20] T Dysart, P Kogge, M Deneroff, E Bovell, P Briggs, J Brockman, K Jacobsen, Y Juan, S Kuntz, R Lethin, et al. Highly scalable near memory processing with migrating threads on the emu system architecture. In *Workshop on Irregular Applications: Architecture and Algorithms*, 2016.

[21] T.L. Falch and A.C. Elster. Register caching for stencil computations on gpus. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2014.

[22] E. Fatehi and P. Gratz. Ilp and tlp in shared memory applications: A limit study. In *International Conference on Parallel Architectures and Compilation*, 2014.

[23] M. Gao and C. Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *International Symposium on High Performance Computer Architecture*, 2016.

[24] B. Gatliff. Embedding with gnu: the gnu compiler and linker. *Embedded Systems Programming*, 13(2), 2000.

[25] M. Gebhart, D.R. Johnson, D. Tarjan, S.W. Keckler, W.J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *International Symposium on Computer Architecture*, 2011.

[26] M. Gebhart, S.W. Keckler, B. Khailany, R. Krashinsky, and W.J. Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *International Symposium on Microarchitecture*, 2012.

[27] V. Geraeinejad, Q. Qian, and M. Ebrahimi. Investigating register cache behavior: Implications for cuda and tensor core workloads on gpus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 14(3), 2024.

[28] J. Gómez-Luna, I.E. Hajj, I. Fernandez, C. Giannoula, G.F. Oliveira, and O. Mutlu. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access*, 10, 2022.

[29] ARM Holdings. Arm neoverse n1 core, technical reference manual, 2019.

[30] B. Hyun, T. Kim, D. Lee, and M. Rhu. Pathfinding future pim architectures by demystifying a commercial pim technology. In *International Symposium on High-Performance Computer Architecture*, 2024.

[31] A. Jain and C. Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[32] A. Jain, C. Lin, and E. J. Natalie. *Cache Replacement Policies*. Morgan & Claypool Publishers, 2019.

[33] A. Jaleel, K.B. Theobald, S.C. Steely, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, 38(3), jun 2010.

[34] T.M. Jones, M.F.P. O'Boyle, J. Abella, A. González, and O. Ergin. Energy-efficient register caching with compiler assistance. *ACM Trans. Archit. Code Optim.*, 6(4), oct 2009.

[35] N.B. Lakshminarayana and H. Kim. Spare register aware prefetching for graph algorithms on gpus. In *International Symposium on High Performance Computer Architecture*, 2014.

[36] P. Lavin, J. Young, J. Riedy, R. Vuduc, A. Vose, and D. Ernst. Spatter: A tool for evaluating gather / scatter performance, 2020.

[37] S. Lee, S. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N.S. Kim. Hardware architecture and software stack for pim based on commercial dram technology : Industrial product. In *International Symposium on Computer Architecture*, 2021.

[38] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys*, 49(2), 2016.

[39] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27, 2009.

[40] M. Nemirovsky and D. Tullsen. *Multithreading architecture*. Springer Nature, 2022.

[41] P.R. Nuth and W.J. Dally. The named-state register file: implementation and performance. In *International Symposium on High Performance Computer Architecture*, 1995.

[42] D.W. Oehmke, N.L. Binkert, T. Mudge, and S.K. Reinhardt. How to fake 1000 registers. In *International Symposium on Microarchitecture*, 2005.

[43] A. Pellegrini and C. Abernathy. Arm neoverse n1 cloud-to-edge infrastructure socs. In *Hot Chips Symposium*, 2019.

[44] M.K. Qureshi, A. Jaleel, Y.N. Patt, S.C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *International Symposium on Computer Architecture*, 2007.

[45] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu. Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching. In *Architectural Support for Programming Languages and Operating Systems*, 2018.

[46] S. Sethumurugan, J. Yin, and J. Sartori. Designing a cost-effective cache replacement policy using machine learning. In *International Symposium on High-Performance Computer Architecture*, 2021.

[47] I. Shah, A. Jain, and C. Lin. Effective mimicry of belady's min policy. In *International Symposium on High-Performance Computer Architecture*, 2022.

[48] R. Shioya, K. Horio, M. Goshima, and S. Sakai. Register cache system not for latency reduction purpose. In *International Symposium on Microarchitecture*, 2010.

[49] S. Shukla, S. Bandishte, J. Gaur, and S. Subramoney. Register file prefetching. In *International Symposium on Computer Architecture*, 2022.

[50] A. Stillmaker and B. Baas. Scaling equations for the accurate prediction of cmos device performance from 180nm to 7nm. *Integration*, 58, 2017.

[51] J.E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W.R. Davis, P.D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal. Freepdk: An open-source variation-aware design kit. In *International Conference on Microelectronic Systems Education*, 2007.

[52] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture*, 1995.

[53] C. A. Waldspurger and W. E. Weihl. Register relocation: Flexible contexts for multithreading. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[54] S.L. Xi, A. Augusta, M. Athanassoulis, and S. Idreos. Beyond the wall: Near-data processing for databases. In *International Workshop on Data Management on New Hardware*, 2015.

[55] S. Yun, B. Kim, J. Park, H. Nam, J.H. Ahn, and E. Lee. Grande: Near-data processing architecture with adaptive matrix mapping for graph convolutional networks. *IEEE Computer Architecture Letters*, 21(2), 2022.

[56] R. Yung and N.C. Wilhelm. Caching processor general registers. In *International Conference on Computer Design. VLSI in Computers and Processors*, 1995.

[57] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11), Nov 2019.

[58] H. Zeng and K. Ghose. Register file caching for energy efficiency. In *International Symposium on Low Power Electronics and Design*, 2006.

[59] Y. Zhao, M. Gao, F. Liu, Y. Hu, Z. Wang, H. Lin, J. Li, H. Xian, H. Dong, T. Yang, N. Jing, X. Liang, and L. Jiang. Um-pim: Dram-based pim with uniform & shared memory space. In *International Symposium on Computer Architecture*, 2024.