SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores

ANITA TINO, INRIA, France
CAROLINE COLLANGE, INRIA, France
ANDRÉ SEZNEC, INRIA, France

This work introduces Single Instruction Multi-Thread Express (SIMT-X), a general-purpose Central Processing Unit (CPU) microarchitecture that enables Graphics Processing Units (GPUs)-style SIMT execution across multiple threads of the same program for high throughput, while retaining the latency benefits of out-of-order execution, and the programming convenience of homogeneous multi-thread processors. SIMT-X leverages the existing Single Instruction Multiple Data (SIMD) back-end to provide CPU/GPU-like processing on a single core with minimal overhead. We demonstrate that although SIMT-X invokes a restricted form of Out-of-Order (OoO), the microarchitecture successfully captures a majority of the benefits of aggressive OoO execution using at most two concurrent register mappings per architectural register, while addressing issues of partial dependencies and supporting a general-purpose Instruction Set Architecture (ISA).

CCS Concepts: • **Computer systems organization** → **Single instruction, multiple data**; **Superscalar architectures**;

Additional Key Words and Phrases: SIMT, out-of-order, microarchitecture, computer architecture, multi-threading, hardware

ACM Reference format:
Anita Tino, Caroline Collange, and André Seznec. 2020. SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores. *ACM Trans. Archit. Code Optim.* 17, 2, Article 15 (May 2020), 23 pages.
https://doi.org/10.1145/3392032

1 INTRODUCTION

Heterogeneous multi-core processors are a computing architecture design standard. Out-of-Order (OoO) processors are the computing model of choice for achieving latency-sensitive performance of common applications, whereas Graphics Processing Units (GPUs) are used to improve upon parallel computation with the Single Instruction Multi-Thread (SIMT) model. Heterogeneous systems consisting of both types of cores are designed to compromise between throughput- and latency-oriented performance. This collaborative computing, however, remains a challenge as Central Processing Units (CPUs) and GPUs have vastly different architectures and computing models [Nickolls and Dally 2010; Mittal and Vetter 2015]. Consequently, writing software for heterogeneous

This work is supported by Mitacs.
Authors' address: A. Tino, (Current Affl) School of Engineering Science, Simon Fraser University, 8888 University Drive, Burnaby BC, Canada, V5A 1S6; email: anita_tino@sfu.ca; C. Collange and A. Seznec, Inria Centre de recherche Rennes — Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes Cedex, France; emails: {caroline.collange, andre.seznec}@inria.fr.

1544-3566/2020/05-ART15
https://doi.org/10.1145/3392032

platforms has a cost, both in programmability and efficiency. For instance, the energy efficiency benefits of heterogeneous systems are negated due to the communication and synchronization overhead incurred by hybrid algorithms [Haidar et al. 2017]. Likewise, the serial performance provided by GPUs demonstrates an impact on overall system performance [Wong and Aamodt 2009].

As a way to mitigate heterogeneous issues, OoO CPUs integrate increasingly powerful Single Instruction Multiple Data (SIMD) instruction sets, such as Intel Advanced Vector Extensions (AVX)-512 and ARM Scalable Vector Extension (SVE), for vector-like processing of parallel applications. However, taking advantage of these SIMD extensions demands compiler-based or manual vectorization, demanding extra effort from programmers.

In this article, we push the envelope for the latency/bandwidth trade-off by introducing microarchitectural support for transparent SIMT-style vectorization across different CPU threads within a multi-threaded OoO pipeline, which supports a general purpose ISA. It enables dynamic vectorization of existing multi-thread binaries compiled for general-purpose processors, without any compiler intervention. Our SIMT Express (SIMT-X) microarchitecture closely couples latency- and throughput-oriented compute on a single core using a restricted form of OoO, which sufficiently captures majority of the benefits of aggressive, speculative execution. We are not aware of any prior work combining SIMT and OoO execution. We believe this is due to two challenges that have prevented so far register renaming and speculative execution in SIMT architectures.

- The *partial dependency problem*: SIMT execution works by emitting vector instructions predicated with an implicit mask, so that threads that do not follow the current control-flow path cause no architecturally-visible effect. Masking introduces extra dependencies between instructions that existing register renaming mechanisms are unable to eliminate.
- The *multi-path tracking problem*: SIMT microarchitectures aggregate threads that take the same control-flow paths through the program, dynamically forming groups as threads diverge and converge. The thread grouping mechanisms of existing or proposed GPUs do not support speculative instruction fetch and execution.

SIMT-X addresses both the partial dependency and multi-path tracking problem dynamically. To address the multi-path tracking problem, SIMT-X implements a front-end that predicts divergence and detects convergence between threads of a warp, managing many speculative and committed paths in the pipeline. For the partial dependency problem, a Path-Identifying Register Alias Table (PIRAT) supports multiple concurrent physical register mappings per architectural register. The PIRAT is a generalized approach to register renaming for masked vector execution, derived from a scheme originally proposed by Wang et al. [2001] for predicated scalar execution. To limit complexity, our PIRAT implementation supports up to two concurrent register mappings to eliminate the vast majority of artificial dependencies that are induced by OoO SIMT register merging. Control dependencies that cannot be eliminated are transformed into data dependencies using micro-operations that are dynamically injected into the pipeline.

The rest of this article is organized as follows: Section 2 provides motivation for an architecture that compromises between sequential and parallel performance. Section 3 provides insight on the partial dependency problem and challenges considering generalized SIMT execution, while outlining the SIMT-X microarchitecture. Section 4 evaluates performance and design tradeoffs, where Section 5 reviews related work, and a conclusion is provided in Section 6.

2 BACKGROUND

2.1 Revisiting Closely-coupled Parallel Accelerators

Heterogeneous multiprocessor systems are seen as a way to unite single- and multi-thread efficiency. CPUs and GPUs have vastly different architectures and programming models that present several challenges in hybrid computing [Mittal and Vetter 2015], including programming model and software stack inefficiencies. Hybrid CPU-GPU algorithms, for instance, incur memory transfer and synchronization overhead. For smaller batch sizes, GPU-only implementations have been shown to offer better overall energy efficiency on mixed parallel/serial algorithms [Haidar et al. 2017]. Likewise, the latency of GPU tasks and data communication has been shown as an important factor affecting hybrid performance [Wong and Aamodt 2009].

Architectures that incorporate closely-coupled SIMD or vector extensions within a superscalar core hold the potential to address the shortcomings of these coarse-grain heterogeneous architectures. The concept of closely-coupled parallel accelerators is not new: out-of-order vector processors were originally proposed in the late 1990s [Espasa et al. 1997], and thereafter applied in the Tarantula design [Espasa et al. 2002]. However, SIMD and vector architectures have long been split between special-purpose in-order architectures supporting powerful SIMD/vector instruction sets with masked execution such as GPUs, and general-purpose out-of-order cores possessing short-vector SIMD instruction sets without masking support such as Intel AVX2. More recent incarnations of closely-coupled vector/SIMD engines, including Intel AVX-512, ARM SVE, and RISC-V vector extensions, have finally implemented masked execution within general-purpose cores. By providing tight integration with a superscalar core, vector units benefit from a full range of microarchitecture mechanisms including dynamic scheduling, register renaming, branch prediction, and memory disambiguation. Through multiple computations, the coarse granularity of vector instructions amortizes the overhead of OoO logic.

2.2 Thread-centric Parallel Environments

The parallel computing landscape has undergone radical changes with the advent of multi-core CPU and GPUs. The *vector-centric* programming models of the 1990s gave way to current *thread-centric* programming models. Popular programming platforms include OpenMP and OpenACC for directive-based language, and CUDA and OpenCL for explicitly-parallel languages. These programming models are classified as Single-Program Multiple-Data (SPMD) applications.

SPMD applications express parallelism with multiple threads running the same program on different data. The parallelism exhibited in these SPMD-based programs is appropriate for multi-core, hardware multi-threading, and/or vector execution. For the latter, fine-grained *threads*, or work items, are statically grouped into *warps*. The runtime systems that execute SPMD applications run each thread of a warp on a single lane of a SIMD unit. Differentiated control-flow is implemented by means of masked execution. Depending on the implementation, masks may be managed by the compiler [Karrenberg and Hack 2011; Pharr and Mark 2012; Lee et al. 2014] or by the microarchitecture [Lindholm et al. 2008; Fung et al. 2009]. In both cases, instructions are explicitly or implicitly predicated using activity mask registers.

2.3 Masked Vectors Thwart Renaming

Although masked vectors and SIMD execution are well-suited for in-order cores, OoO architectures present a challenge. OoO processors use register renaming to remove false dependencies and provide flexible instruction scheduling. This renaming process assigns a new physical register to each instruction's destination register, while renaming its sources according to prior physical-to-architectural register mappings.

When considering control-flow execution for an OoO model, masked SIMD instructions only affect a subset of the destination registers while the masked lanes are left untouched. In this case, multiple definitions of the same architectural register exist; the idle lanes must preserve the former content of the destination architectural register, whereas the active lanes now have new content that will be stored to newly allocated physical register.

This challenge of OoO masked SIMD execution has similarities with the multiple definition problem in the context of OoO execution of predicated scalar instructions [Wang et al. 2001; Prémillieu and Seznec 2014]. To compound the scalar multiple definition problem, masked SIMD execution requires merging the results of active lanes with the former register value, rather than merely selecting the source. Indeed, to preserve a single register definition among the lanes, the instruction takes the older physical destination register as input, and *merges* the contents with new values using the mask vector produced by the instruction. This makes the instruction data-dependent on both older branches through the path mask, and on prior writes to the architectural destination register.

Introducing such extra artificial data dependencies into the rename stage may affect performance. For instance, the Intel software Optimization Reference Manual for the Skylake microarchitecture warns that *"Using masking may result in lower performance than the corresponding non-masked code. This may be caused by [. . .] dependency on the destination when using merge masking"* [Intel Corporation 2017].

Consequently, performing standard register renaming on masked instructions introduces artificial chains of Write-After-Write (WAW) dependencies, serializing register accesses and defeating the purpose of register renaming. Invoking such a systematically predicated approach introduces a partial dependency problem for OoO SIMT, and is therefore impractical considering multiple warp divergences and active threads that may coexist simultaneously.

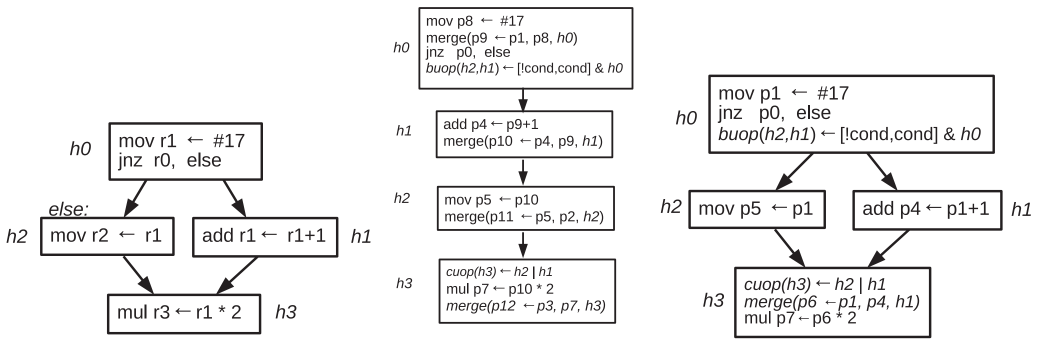
3 THE SIMT-X MICROARCHITECTURE

A SIMT microarchitecture exposes a set of hardware threads, which is statically partitioned into *warps*, such that the warp size matches the vector width of the SIMD back-end. Each thread of a warp executes on its given lane, and multiple warps are kept in flight for latency hiding purposes. From the point of view of the pipeline back-end, each SIMT warp is essentially equivalent to an SMT thread running SIMD instructions. Threads of a warp, however, may traverse different paths of execution. The concept of an OoO SIMT architecture, such as SIMT-X, therefore brings forth two major challenges: (1) attending to partial register dependencies with minimal serialization, and (2) managing multiple diverging and converging control flow paths considering warp-like, speculative execution.

The SIMT-X microarchitecture presents solutions to both register renaming and path tracking for OoO SIMT applicability. To address the partial dependency problem, SIMT-X allocates full-vector registers to instructions similar to SIMD, but defers register merging only until necessary. The key idea behind SIMT-X's merging is to convert associated control-flow to dataflow using micro-instructions generated dynamically by the microarchitecture. To manage thread divergence and re-convergence within a warp, SIMT-X introduces the concept of active *path* tracking using two simple hardware structures that (1) avoid mask dependencies, (2) eliminate mask meta-data within instructions, and (3) easily invalidate inactive execution paths during divergence and/or mispeculation.

3.1 From SPMD Control Flow to SIMD Data-flow

Figure 1(a) provides an `if-then-else` conditional structure typical of an SPMD application. A *path* corresponds to a sequence of instructions that execute under the same mask, and each path is assigned a vector mask register. The example has four path mask registers named `h0` to `h3`. As seen in the figure, register `r1` is set at the beginning of the code, and only overwritten by one side of the branch (path `h1`) during divergence. When warp paths `h1` and `h2` re-converge at `h3`, register `r1` must be merged to maintain a single register definition. Considering multiple threads per warp, it is possible that both paths `h1` and `h2` may be traversed.



(a) SPMD Application Example (b) Naive Register Merging for SIMT Execution in Divergence (c) SIMT-X register merging

Fig. 1. SPMD control-flow and register merging.

SIMT-X's dynamic microarchitecture uses two micro-operations (μops) to manage paths: branch μops (buops) to create divergent paths, and convergent μops (cuop) for re-convergent paths, which will be further detailed in Section 3.5. The μops are injected dynamically into the pipeline by the microarchitecture in order to compute the actual path masks at execute time, and perform state bookkeeping at retirement time. This leverages the OoO engine ability to schedule these μops following actual data and control dependencies. Assume that architectural registers `r0` to `r3` are initially mapped to their respective physical registers `p0` to `p3` in the example.

The straightforward case displayed in Figure 1(b) demonstrates how μops are used to manage paths, and how registers may be naively merged after every register write to maintain the single register definition property. Such a naive approach serializes instructions according to write-after-read dependencies, as only one physical definition may represent an architectural register at a time, thus serializing `h1` and `h2`. This serialization defeats the purpose of register renaming and the data-flow execution sought by OoO SIMT.

Figure 1(c) presents the register renaming method employed by the SIMT-X microarchitecture. In this example, register `r1` is mapped to `p4` on the "if" side of the branch (`h1`), and `p1` on the "else" side (`h2`). Thus, multiple physical register definitions may be associated with a single architectural register simultaneously. A technique implementing multiple register associations allows control-flow to be treated individually, while promoting dataflow for OoO SIMT execution. As seen in the figure, register `r1` is read when the paths re-converge; hence, it is only necessary that `p1` and `p4` be merged at path `h3` to maintain a single vector register definition, `p6`, based on the path mask `h1`. The goal of SIMT-X is therefore to defer register merging only until necessary to reduce the number of merges, where multiple physical definitions may be assigned per architectural register simultaneously.

3.2 Implementing SIMT-X

Figure 2 presents the SIMT-X microarchitecture for OoO SIMT execution. The pipeline stages in SIMT-X are similar to a conventional OoO Simultaneous Multi-Threading (SMT) processor: schedule, instruction fetch and decode, rename, issue/dispatch, execute, writeback, and commit. As seen in the figure, the front-end consists of: (1) a *Path Scheduling Table (PST)* for scheduling warp paths; (2) a *convergence detector* to facilitate path re-convergence [Kalathingal et al. 2017]; (3) a *Pathtable,* which tracks path masks as the paths diverge and re-converge; (4) a *Divergence Predictor* for speculating warp divergences; and (5) a *PIRAT* for orchestrating SIMT-X renaming.

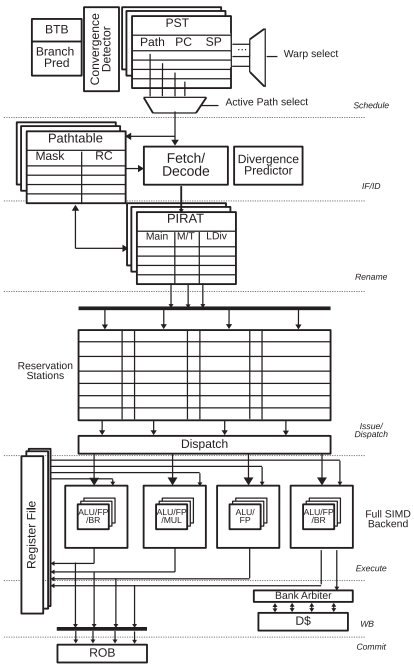


Fig. 2. SIMT-X Pipeline.

SIMT-X has the capability to execute both statically vectorized SIMD instructions generated by the compiler, and instructions dynamically formed by the microarchitecture from scalar threads. To satisfy simultaneous warp execution, the PIRAT, Pathtable, and PST are partitioned between warps, where each warp may monitor its respective paths, and every pipeline stage operates on vector instructions (i.e., vector of instruction instances). The scalar functional units are also replicated so that SIMT-X's back-end may leverage both the scalarized functional units and general purpose SIMD units for execution of implicitly vectorized instructions. The physical register file is shared between all warps, and contains warp-sized vector registers. Each sub-file corresponds to a register context for one scalar thread of a given warp. Micro-operations are generated dynamically by SIMT-X to manage control-flow dependencies.

The following sections provide details pertaining to the microarchitectural features listed.

3.3 Tracking Paths

A conventional OoO processor maintains one speculative and one committed instruction stream per thread. Such a core relies on sequential program ordering to recover from mis-speculation and preserve a consistent architectural state. Considering the SIMT model, threads of a warp may converge and diverge, creating and traversing different paths (streams) of execution for a given application. As the case for certain GPUs, imposing an arbitrary serial ordering for all divergent branch paths [Nickolls and Dally 2010] would make speculation difficult and recovery costly considering a general-purpose OoO core. In contrast, the SIMT-X microarchitecture integrates dynamic mechanisms to independently track speculative and committed states for all concurrently executing paths, while supporting a traditional software stack.

Although threads within a warp are expected to have similar control-flow, they may still follow different paths of execution during runtime. To keep *track* of divergent paths, SIMT-X integrates a *Pathtable* that maintains all active paths for each warp, and their speculative or committed state.

An index into the Pathtable, or *path ID*, is assigned to each path as it is created in the microarchitecture through divergence or re-convergence. The path ID is used thereafter throughout the pipeline to infer a path's state by indexing into the Pathtable. The table allows paths to speculatively execute their instructions, where the actual mask is resolved at execution. The pathtable allows SIMT-X to easily invalidate and/or update warp threads without the need to buffer mask meta-data throughout the pipeline.

3.4 Scheduling Paths

SIMT-X *schedules active paths* in the front-end using a PST. On account that the Pathtable tracks path masks, the PST must only maintain path PCs and similar information required for scheduling, such as call-return depth and barriers. Each entry in the PST may directly infer its path's state using its respective path ID.

The fetch steering policy used by SIMT-X's scheduler is based on a revised heuristic of MinSP-PC [Milanez et al. 2014]. MinSP-PC is a simple fetch policy used to obtain efficient lockstep execution between threads of a warp. The highest priority is given to the path with the deepest call stack. If two paths possess the same minimum stack pointer, the minimum PC criteria is used. Thus, MinSP monitors call depth to promote re-convergence, while MinPC allows for fine-grained synchronization. Considering general-purpose parallel applications, it is possible that MinSP-PC may cause livelocks [ElTantawy and Aamodt 2016]. Accordingly, a hybrid Round-Robin (RR) approach was proposed by Kalathingal et al. [2017] to guarantee forward progress, activated periodically every $(m + 1) \times n$ cycles, where m is the number of threads/warp with a total of n warps.

SIMT-X adopts this RR/MinSP-PC policy, applying a conventional RR algorithm for interleaving and scheduling individual *warps*. The original RR/MinSP-PC policy [Kalathingal et al. 2017], however, is used by the *paths* of a warp to guarantee forward progress and lockstep execution.

3.5 Control-flow Micro-operations

Micro-operations (μops) generated by SIMT-X manage OoO SIMT dependencies by operating on masks maintained by the Pathtable. The μops do not require alterations or extensions to the existing ISA and maintain binary, compiler, and software compatibility. Rather, SIMT-X μops are generated dynamically at the decode and rename stages of the pipeline. Dynamic generation takes advantage of the fact most branches do not diverge to only compute predicate mask and merge vector registers when necessary, saving execution resources.

SIMT-X employs three types of μops: (1) *convergent μops (cuops)* and (2) *branch μops (buops)*, which compute each path mask after convergence and divergence, respectively, and (3) *merge μops* which perform register merging dynamically to preserve single register definitions. Figure 1(c) provides an example of (1) a buop that computes the paths `h2` and `h1` from `h0`, and branch condition (*cond*), (2) a cuop that computes path `h3` from `h2` and `h1`, and (3) a merge uop, merging the register definitions `p1` and `p4` into `p6` based on the mask `h1`.

3.5.1 Branch Divergence Micro-operations (buop). To prepare for the possibility of a divergent path, a branch that is predicted divergent is transformed into a buop. A new path must be created for the divergent path, where a new path ID is obtained and its respective entry allocated in the Pathtable. A buop possesses two source operands: (1) the previous path ID (h_i), and (2) the instruction's branch condition, c, which are used to compute the path's new mask (h_j). The previous path operand is used to guarantee that the preceding path is valid prior to its execution, where c is used to compute the new path mask. A buop computes the bitwise AND of the path mask and c, such that $h_j \leftarrow h_i \wedge c$, where the alternate path (once resolved as divergent) may be computed as $h_k \leftarrow h_i \wedge \neg c$

3.5.2 Convergence Micro-operations (cuop). SIMT-X conducts a convergence check in the front-end during the scheduling stage. If convergence is detected between two paths, a new path ID is obtained and entry allocated in the Pathtable for the new re-convergent path. At this time, a *Convergence Micro-operation (cuop)* is also injected into the pipeline, taking the two path IDs as its source operands (h_i, h_j), and a new path ID (h_k) as its target operand. The cuop guarantees that the paths h_i and h_j are valid prior to h_k's validation. At execution time, the cuop merges the path masks using a bitwise OR operation, such that $h_k \leftarrow h_i \vee h_j$.

3.5.3 Merge Micro-operations. Given that many active paths may traverse the pipeline concurrently with multiple architectural register definitions in-flight, their physical register mappings must be merged (as required) to dynamically maintain the single-definition property for a given consumer path. Merge μops allow SIMT-X to dynamically manage and defer merging only when necessary. A merge uop possesses three source operands: the two register addresses to be merged (R_i *and* R_j), and a corresponding path ID (h_i). A merge uop contains a target operand R$_k$ representative of the destination register, which the source registers will move their content once all operands are valid. Accordingly, a merge uop is computed as $R_k \leftarrow (\neg h_i \wedge R_i) \vee (h_i \wedge R_j)$.

In general, the merging of physical registers allows SIMT-X to maintain a manageable amount of free physical registers for all warps, where threads of a warp may read and write to the same physical registers with low hardware complexity and higher energy efficiency in comparison to other systematically predicated approaches. Further details pertaining to SIMT-X's renaming strategies are provided in Section 3.8.

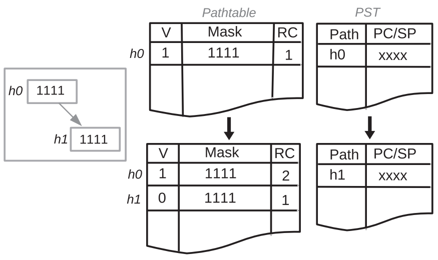
3.6 Convergence Detection

As previously mentioned, SIMT-X adopts path re-convergence detection using the RR/MinSP-PC policy implemented by Kalathingal et al. [2017], however with PST hardware for OoO SIMT applicability. Instruction PCs are speculated in the front-end based on branch history and the first active scalar thread in a given path. These PCs are compared to other path PCs in the PST, where a match indicates that the two paths may be merged and a cuop injected into the pipeline as detailed in Section 3.5.2. A new path ID is obtained for the two re-convergent paths, where the PST and Pathtable are updated accordingly. Since two active paths exist in the PST prior to the convergence of h_i and h_j, one PST entry is overwritten with the new active convergent path h_k, and the alternate path is evicted.

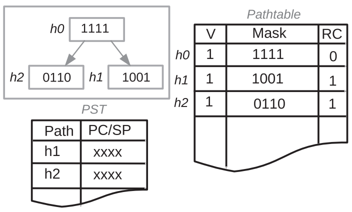
3.7 Divergence and Branch Prediction

Divergence occurs when branch instructions are encountered, bringing forth the possibility of altering thread paths within a warp. The key idea of SIMT-X divergence and branch prediction is that steering the front-end does not require knowledge of the vector of branch conditions for all threads. Instead, it merely requires a single prediction for the whole warp to estimate which of three cases is more likely: whether (1) all threads of the warp will take the branch, if (2) none of the threads will take the branch, or if (3) the branch will diverge.

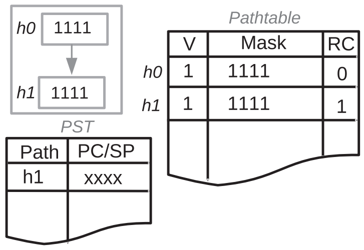
3.7.1 Pathtable and Divergence. Figure 3 illustrates SIMT-X's Pathtable and PST functionality during divergence prediction, corresponding to the example presented in Figure 1(a). In this example, path *h0* is non-speculative (i.e., v = 1 in the Pathtable) and fully convergent. In the case of Figure 3(a), a branch is encountered on this path and predicted divergent, where the speculative path ID `h1` is allocated in the Pathtable. Since the precise set of threads that follow path `h1` at divergence is unknown at this time, the speculative mask of `h1` is conservatively copied from `h0`. In order to steer the front-end onto the new active path, `h0`'s entry in the PST is overwritten by the new path ID `h1`, where `h1` may be used by all subsequent warp instructions to reference their path state. A branch micro-operation (buop) is also injected into the pipeline at this point, where the reference counter for `h0` is incremented as the buop has referenced the path.



(a) Path predicted divergent, renewal of path



(b) Divergence prediction resolved as correct



(c) Divergence prediction resolved as incorrect, actually uniform-execution proceeds

Fig. 3. Pathtable divergence example.

If the branch is resolved as divergent once executed (Figure 3(b)), a new Pathtable entry, h2, is created for the alternate path while h1 and h2's masks are updated to reflect the outcome of the threads. A new PST entry is also allocated for h2 so that it may be scheduled in the front-end. Once the buop is committed, both IDs are marked as valid, signifying a non-speculative path. The path h0 is then completely de-referenced by the paths, and may be safely evicted as subsequent instructions rely on the new paths.

Accordingly, the renewal of a warp's path during divergence prediction allows alternate paths to be easily created while forgoing the need to adjust other paths. In the case that divergence was mispredicted assuming correct branch prediction, a useless Pathtable entry was created as illustrated in Figure 3(c). In this case, the divergence predictor's misprediction is updated, where the path *h1* is still validated and *h0* de-referenced once the uop commits. Therefore such a path renewal technique avoids the need to flush the pipeline upon a misprediction, where execution may proceed.

In the case that multiple divergent and speculative paths exist in a given Pathtable, a free and commit pointer are maintained to track younger path entries. Thus, when a divergent path is resolved and the mask is calculated, younger entries that conservatively copied their masks from the given path may also invalidate their inactive thread lanes.

3.7.2 Handling Mispeculation & Exceptions. Based on SIMT-X's implementation, it is safer to predict that a path will be divergent rather than uniform. That is, if a branch is predicted divergent, the path is renewed in the Pathtable to prepare for the new path. In the worst case, a useless entry was created for a uniform path incorrectly, predicted as divergent. On the other hand, we find that only 1.4% of branches diverge on average (protocol described in Section 4), so considering all branches as divergent would cause too many bookkeeping operations. To accommodate these characteristics, SIMT-X integrates a global bi-modal divergence predictor biased toward divergence, i.e., divergence is always predicted unless the predictor suggests that the branch is strongly uniform.

A summary of other branch/divergent prediction outcomes are provided below:

- **Predicted divergent, actually uniform but incorrect branch direction—**In this case, the path mask created during divergence prediction is nullified in the Pathtable, and removed from the PST. The front-end creates and fetches from the new correct path with the updated mask. Instructions fetched previously on the wrong path eventually commit with the null mask, but do not affect the architectural state of SIMT-X.
- **Predicted non-divergent and correct, with correct branch direction—** Execution proceeds as normal.
- **Predicted non-divergent and correct, but incorrect branch direction, OR predicted non-divergent and incorrect—** As the case of a conventional OoO branch mispredicition, SIMT-X must rollback to its checkpointed state, prior to the offending branch, and flush the pipeline.

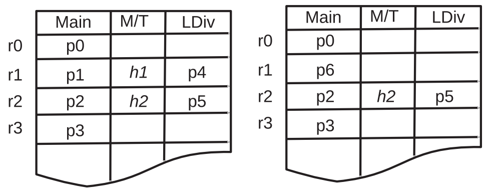
Only a complete divergence misprediction or uniform branch misprediction therefore requires SIMT-X to rollback to a given warp's checkpointed stated prior to the offending branch. In order to rollback considering both mispredictions and exceptions for a given warp, the PIRAT and PST possess retirement copies. The Pathtable, however, does not require a retirement copy as it may simply invalidate and nullify the path mask while draining the pipeline.

3.8 Register Renaming

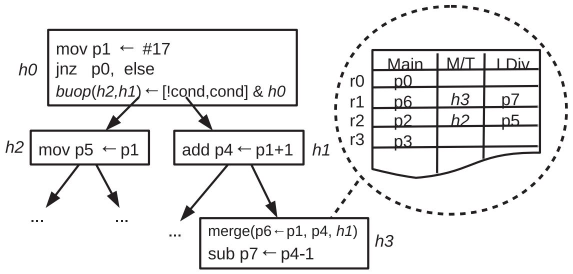
Monitoring warp states and paths in a pipeline is important to ensure single register definitions in OoO SIMT. A PIRAT is used by SIMT-X to manage multiple physical register mappings, and associate each to their architectural definition. The PIRAT is similar to a conventional Register Alias Table (RAT); however, the structure allows multiple register definitions to coexist simultaneously considering various control paths of coalesced SPMD threads.

Unlike prior work on multiple renaming [Wang et al. 2001], we chose to allow at most two physical registers per architectural register. Although this two physical register assignment may restrict out-of-order execution, it greatly simplifies OoO SIMT renaming, and limits the operand count within a uop. As demonstrated in the experimental results of Section 4, the PIRAT achieves performance within 3.86% of a theoretical OoO SIMT model with cost-free merging. Our experiments also demonstrate that restricting the PIRAT to two entries significantly reduces the number of merge μops and yields higher Instructions Per Cycle (IPC) than a naive (RAT) approach. Since most read-after-write dependencies occur over short distances with no divergence/ re-convergence points between instructions [Prémillieu and Seznec 2012], the PIRAT is optimized both qualitatively and quantitatively for OoO SIMT execution.

The two physical registers used by each PIRAT entry are referred to as *Main* and *Leading Divergent (LDiv)*, as illustrated in Figure 4. To record individual thread mappings for a given architectural register, the *LDiv* mapping is associated either with a mask (once the producer instruction has retired and the actual mask is known) or a path ID, which references a given Pathtable entry (when the producer is still in the pipeline with a speculative mask). The register's mask is reflected in the entry's Mask/Tag (M/T) field mapping. The active bits in the mask represent the *LDiv* register's latest physical mapping for the given threads, whereas the complementary threads are represented by the *Main* register. Let the following terminology represent a PIRAT entry: $(P_{main}, Path, P_{ld})$ where P_{main} is the Main register mapping, *Path* is the LDiv register's *Path ID (h)* or *Mask (m)*, and P_{ld} is the LDiv register mapping.



(a) PIRAT prior to `h3` (Fig. 1c) (b) PIRAT at `h3` convergence



(c) Divergence at `h3` scenario

Fig. 4. PIRAT renaming example.

As discussed in Section 3.5, a path's speculative mask, Wmask, in SIMT-X is constructed to be equal or a superset of the actual mask. As convergence between paths is detected early in the front-end, only divergence can affect the execution mask after the instruction fetch stage. A path mask may therefore only become less populated, that is, Wmask ⊇ ActualMask. This property is useful in the case of source renaming: a physical vector register can be used as a source operand whenever its valid lanes cover a superset of the instruction's mask. For destination renaming, a stronger equality property must be maintained such that Wmask = ActualMask. This equality property, however, can only be guaranteed at the rename stage in the absence of warp divergence. When there is a possibility for divergence, the PIRAT must preserve both the previous register mapping and its new definition.

3.8.1 PIRAT Renaming Rules. To simplify OoO SIMT renaming, the PIRAT's main objective is to dominate a consumer instruction's path with a given physical register definition, such that all threads within a warp may use the rename mapping represented by **either** the *Main* **or** the *LDiv* register definition. This condition guarantees that consumer instructions access a single physical register for each source operand. Conversely, producer instructions may write to a single physical register definition without depending on the previous architectural register mapping. Therefore, if the active threads within a warp do not fall directly into one register definition, the microarchitecture performs dynamic bookeeping through register merging to maintain the single-definition property.

It is assumed that all architectural registers are initially mapped to their corresponding physical (*Main*) register at startup. The following sections specify the set of rules used by SIMT-X for rename.

Reading and Renaming Source Operands. When a given architectural register's PIRAT entry consists solely of a *Main* register mapping, then all threads in the warp share the same physical-to-architectural register mapping. In this case, the physical register specified in the Main field is used to read a given warp's source operand.

If the architectural source register's PIRAT entry consists both of a *Main* and *LDiv* register mapping, then the microarchitecture must check which entry(ies) possesses valid physical mappings for the active path threads. The PIRAT's last know mask, Register Mask (Rmask), represented either as a path ID or mask in the M/T field, must be compared to the current path's speculative mask (Wmask):

- If Wmask $\subseteq$ Rmask, the *LDiv* register entry is used for the source's renaming.
- If Wmask $\subseteq \overline{\text{Rmask}}$, the *Main* register entry is used for the source's renaming.
- Else, the source operand is mapped to both the Main and LDiv physical registers. A new physical register, Pϕ must be obtained, and used as a target in the merge uop. The uop is injected into the pipeline so that the two register definitions may be merged to Pϕ, where the instruction's source operand is also renamed using Pϕ.

When a merge uop is injected into the pipeline, the Main, LDiv, and newly allocated "merged" register mappings are taken as operands, in addition to the path as discussed in Section 3.5.3. Let the merged register be represented as Pϕ, where a merge (ϕ) uop is expressed as P$\phi \leftarrow \phi$(P$_{main}$, m, P$_{ld}$) for a given (immediate) mask, or P$\phi \leftarrow \phi$ (P$_{main}$, h, P$_{ld}$) in the case of a path ID. Pϕ is then mapped to the Main register, where the LDiv entry and path/mask in the Pathtable are updated in the PIRAT such that ϕ(P$_{main}$, h/m, P$_{ld}$) $\rightarrow$ (Pϕ, -, -).

Renaming Destination Operands. If a given PIRAT entry for an architectural destination register consists solely of a valid *Main* register during destination rename, then a free physical register may be obtained from the freelist, P$_{dest}$, to rename the instruction and update the PIRAT's LDiv definition such that $(P_{main}, \text{-}, \text{-}) \rightarrow (P_{main}, h/m, P_{dest})$. If the PIRAT entry for the given architectural destination register consists of both a Main and LDiv register, the M/T bit must be consulted to determine whether the warp is currently traversing a speculative or committed path. If the PIRAT entry for the rename register consists of P$_{main}$, P$_{ld}$, and a **mask**, i.e., (P_{main}, m, P_{ld}), then:

- If no pending divergence exists **and** Wmask $\supseteq m$:
  - The LDiv register P$_{ld}$ may be safely overwritten with the free physical register P$_{dest}$, such that $(P_{main}, m, P_{ld}) \rightarrow (P_{main}, h, P_{dest})$.
- If no pending divergence exists **and** Wmask $\supseteq \overline{m}$:
  - The content of Main is replaced with P$_{ld}$, where LDiv is updated with P$_{dest}$ to reflect the new path, such that $(P_{main}, m, P_{ld}) \rightarrow (P_{ld}, h, P_{dest})$.
- Else, there is a possibility that the result's writeback overlaps with both P$_{main}$ and P$_{ld}$, or that a pending divergence exists. In this case:
  - Registers P$_{main}$ and P$_{ld}$ must be merged. Two free registers, P$_{dest}$ and P$'_{dest}$ are obtained from the freelist, where a merge uop is also injected into the pipeline such that $P_{dest} \leftarrow \phi(P_{main}, m, P_{ld})$, where $m(h) \supseteq m$
  - The destination is renamed with P$'_{dest}$ where the PIRAT is updated as $(P_{main}, m, P_{ld}) \rightarrow (P_{dest}, h, P'_{dest})$. Subsequent consumer instructions are renamed as P$'_{dest}$.

If the PIRAT entry for the given architectural destination register consists of a Main and LDiv register mapping, and a **path ID** h, such that (P_{main}, h, P_{ld}), the mask must be obtained from the Pathtable using the path ID h, thereafter applying the same rules as the case of a mask.

An example of PIRAT renaming is give in Figure 4(a) and (b), with reference to the SPMD application example provided in Figure 1(a) and (c). In this example, the path `h1` renames `r1` with `p4`, where the PIRAT reflects this by updating the M/T field with path ID `h1`, and LDiv field with the register `p4`. In this case, two physical mappings are associated with architectural register `r1` as shown in Figure 4(a). Each path may use their individual register definition for `r1` simultaneously.

Similarly, path h2 renames the architectural register r2 to p5, which is updated in the PIRAT's LDiv entry to reflect the new physical mapping. r1 is required thereafter by h3 once h1 and h2 re-converge. Since h3's definition for r1 is mapped to both the Main and LDiv definitions, the registers must merge using a merge uop and a new register definition p6, which is updated in the Main field as shown in Figure 4(b).

Figure 4(c) illustrates the example of Figure 1(c), however, considering another divergence after path h2; the PIRAT at h2 is therefore identical to that of Figure 4(a). In the case of divergence at h3, the path inherits the mask of h1, where h3 contains the instruction sub r1 $\leftarrow$ r1-1. During rename, the PIRAT reflects r1's source mapping as p4 such that h3 $\supseteq$ h1. Since pending divergences exist when renaming the destination register, two new registers (p6 and p7) are obtained from the freelist, merging p1 and p4 to p6 such that p6 $\leftarrow \phi$ (p1, h1, p4), renaming the destination r1 with p7 for sub p7 $\leftarrow$ p4-1, resulting in the PIRAT of Figure 4(c).

3.8.2 Anticipating Divergence at Rename. To track speculative divergence, referred to as *pending divergences*, SIMT-X employs a global divergence counter. When a divergence is predicted in the front-end, the counter is incremented and a buop is injected into the pipeline. Likewise, when the buop is committed, the counter is decremented. If the counter does not maintain a zero state, it may be assumed that a pending divergence exists in the pipeline. Considering a divergence misprediction, the given branch may be transformed (into a buop or back to a branch) and the divergence counter updated to reflect the correct outcome, proceeding thereafter in the pipeline until committed. The PIRAT uses this counter to maintain its renaming properties during speculative execution, as discussed in Section 3.8.1.

3.8.3 Pipelining & Physical Implementation. The PIRAT's physical implementation for rename is similar to a conventional pipelined RAT implementation; however, it must also perform automatic merges for destination writes and source reads as required. Similar to a conventional rename pipeline [Safi et al. 2011], once operands have been decoded, WAW and RAW hazards are detected concurrently to source reads. In the next cycle, the PIRAT may update writes and reads according to any RAW present, where merge uop generation is also handled. The PIRAT, however, must stall in the case that multiple merges are generated for a given architectural entry, that is, if: (1) multiple source and/or destination merges are required for a given architectural register of a given rename block, or (2) a WAW requires both a merge and/or LDiv write(s) for the given architectural register. In such cases, the PIRAT must serialize its merges to guarantee correct renaming for the architectural register, which is an intentional limitation to maintain low renaming complexity. Nonetheless, results presented in Section 4 demonstrate that these cases occur infrequently and do not effect performance as rename stalls are sufficiently concealed through pipeline buffering.

3.8.4 PIRAT Register Release. Similar to conventional OoO models, SIMT-X maintains a Commit PIRAT to release physical registers. In the classic renaming model, a physical register may not be released until a subsequent instruction overwrites the given architectural register entry at commit. SIMT-X follows similar rules, however, considering the PIRAT's structure and renaming process. Only masks are placed in the Commit PIRAT as instructions are non-speculative, and retiring merge μops release the register definitions as required.

Similar to the renaming procedure in the front-end, when an instruction that obtained a physical register, P_{dest}, commits, if the Commit PIRAT for the given architectural register consists only of a *Main* register P_{main}, P_{dest} may be placed in the *LDiv* field, such that *(P_{main}, -, -)* $\rightarrow$ *(P_{main},m, P_{dest})*. If both the Main and LDiv fields possess register mappings, subsequent instructions will either (1) release the Main or LDiv's physical register based on the non-speculative instruction and register mask, or (2) release both registers in the case of a merge uop, such that *(P_{main}, m, P_{ld})*$\rightarrow$

(P_{dest},-, -). Similar to OoO processors, the Commit PIRAT is also used to establish checkpointing for mispredictions and exceptions.

3.9 Supporting Vector-SIMD and Thread-SIMD Vectorized Instruction Execution

SIMT-X has the capability to execute both statically vectorized SIMD instructions generated by the compiler, and the vector instructions implicitly formed by the microarchitecture. The instructions vectorized across threads may execute on the vectorized Arithmetic Logic Unit (ALU) functional units and/or FP/SIMD units whenever available.

To support both static vector instructions and thread warp execution on the SIMD units, SIMT-X's register file is banked using the free banking method employed by the Dynamic Inter-Thread Vectorization Architecture (DITVA) architecture [Kalathingal et al. 2017]. With the free banking method, a hash function Exclusive Or (XOR) is used to establish slice distribution across the lanes, providing a unique order per thread. That is, for a given thread *i*, the lane *j* will be responsible for executing the thread *i XOR j*. Using this method, a given thread of a warp will be allocated to the same bank, and will not depend on the register index.

For scalar instructions that SIMT-X vectorizes across threads, all lanes may be issued concurrently, each lane reading from an instance of the vector register file and writing back to another. So that lanes 0 and 2, or 1 and 3 may be executed in the same cycle, 128-bit SSE instructions are hashed . Accordingly, depending on the warp mask, which consists of two threads or less, the SSE instruction may execute in a single clock cycle (i.e., a warp with mask 0101 or 1010), whereas warps consisting of two or more active threads/warps may take up to two clock cycles to execute a group of SSE instructions. Full 256-bit AVX instructions are serialized in SIMT-X, one thread at a time, similar to a conventional SMT processor. In this case, a vector instruction consisting of m threads is serialized to issue with a maximum of m successive cycles on the pipelined functional units, where inactive threads are skipped using time compaction.

3.10 Data Memory Accesses

Load and store-based instructions require concurrent access to cache lines, where k threads/warp may request up to k distinct cache lines and/or virtual pages simultaneously. Implementing a fully multi-ported data cache and/or Translation Lookaside Buffers (TLB) to support these concurrent accesses is not feasible. For this reason, SIMT-X employs a banked data cache. Banking in SIMT-X is performed at cache line granularity, where loads support concurrent access to different banks. The bank is selected from a hash of the set number to reduce bank conflicts, considering a virtually indexed L1 cache. As illustrated in Kalathingal et al. [2017], this technique sufficiently reduces data access alignment conflicts for thread stack bottoms on page boundaries.

Similar to conventional SMT OoO cores, SIMT-X uses a Load Store Queue (LSQ) to manage cache misses. All scalar threads monitor the load-store queue to observe total store ordering consistency. Threads of a warp that access the same element in regular memory operations only make a single cache request to effectively reduce the number of loads and stores required. Threads that experience atomic memory operation conflicts spanning several cycles must be stalled and buffered in the pipeline until resolved. As demonstrated in Kalathingal et al. [2017] , a 64-entry split TLB with four lanes provides the same performance as a 256-entry unified multi-ported TLB using a similar SIMT model. SIMT-X with k threads/warp follows this approach by replacing the SMT TLB with k smaller independent single-ported data TLBs.

4 EVALUATION

We model SIMT-X using an in-house trace-driven x86_64 simulator. The Pin Tool [Luk et al. 2005] was used to record execution traces for each thread of a SPMD application. Traces are consumed

Table 1. Simulator Specifications

Parameter	**Specifications**
Scheduling, Predictors	RR warp/cycle with RR/MinSP-PC 64-Kbit TAGE [Seznec 2011] branch predictor+BTB 2b Bi-modal divergence predictor for SIMT-X
Fetch & Decode	4 instructions/cycle, 64-entry buffer
Memory	L1 data cache: 32KB 16-way LRU, 2 cc L2 cache: 4MB LRU, 15 cycles, 64-entry TLB L2 miss: 215 cycles, 114 entry LSQ, 2 GB/s/core
Rename	4 instructions/cycle, 168 int and FP physical registers (x86_64), 2-entry PIRAT
Issue/Dispatch	4 instructions/cycle, 60-entry queue
Functional Units	1 256b ld/str, 1 branch, 1 mul/div, 4 x 64b ALUs, 2 x 256b AVX/FPUs, Vectorized FUs replicated 64b x *k* threads per warp, except AVX where insts serialized
Commit	192-entry ROB, retires 4 instructions/cycle

by the simulator simultaneously, where instructions are scheduled according to specifications of the SIMT-X microarchitecture and its fetch policy. All calls to synchronization primitives such as barriers are maintained in the simulator to guarantee valid scheduling and replay order. Simulation is therefore execution-driven for synchronization-based instructions, and trace-driven for all other instructions.

Table 1 lists simulation parameters for SIMT-X (based on recent Intel/x86_64 processors) which are used for 4-, 8-, and 16-thread processor configurations represented as *Warps(W)*x*Threads(T)* in the figures. Two warp configurations are used during testing, varying the number of threads per warp. An in-order SIMT simulator based on the DITVA [Kalathingal et al. 2017] microarchitecture was also used for comparison, maintaining the same applicable parameters as listed in Table 1 with binary compatibility. Although all microarchitectures are considered as a single core in this study, we simulate the benchmarks using a throughput-limited DRAM memory of 2GB/s bandwidth per core to account for multi-core limitations.

SIMT-X is evaluated using SPMD benchmarks from the PARSEC [Bienia et al. 2008] and Rodinia [Che et al. 2009] suites. All applications have been parallelized with the pthread or OpenMP versions of the benchmark, and AVX vectorization enabled. The following bechmarks are used during experimental evaluation with the simsmall input set: *backprop, barnes, bfs, blackscholes, fft, fluidanmate, fmm, hotspot, kmeans, lavamd, pathfinder, radix, srad, streamcluster, and volrend.*

4.1 SIMT-X Performance

As SIMT-X combines OoO and SIMT execution for a general-purpose ISA, we compare it to the two closest design points: an OoO SMT core supporting explicit SIMD instructions (AVX and SSE), which treats all concurrent threads individually, and an in-order general-purpose SIMT core from the literature, DITVA [Kalathingal et al. 2017]. All three architectures run the same x86-64 executables.

Figure 5 presents the performance for OoO SMT, DITVA and SIMT-X, with geometric means provided in the last columns. Figure 6 illustrates the average thread occupancy per warp for the same SIMT-X and DITVA models. As seen in Figure 5, SIMT-X provides IPC gains over DITVA across the

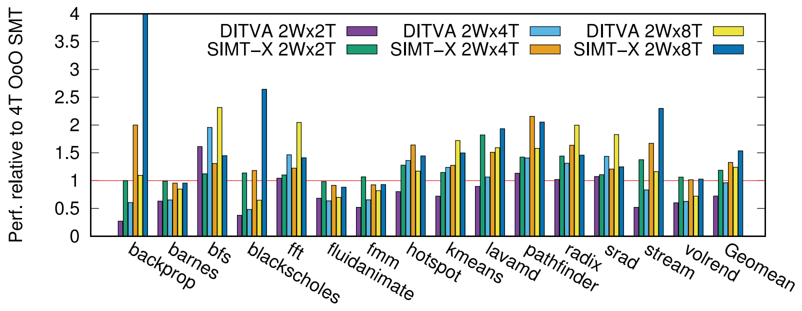


Fig. 5. Performance of OoO SIMT-X and in-order DITVA relative to OoO SMT baseline.

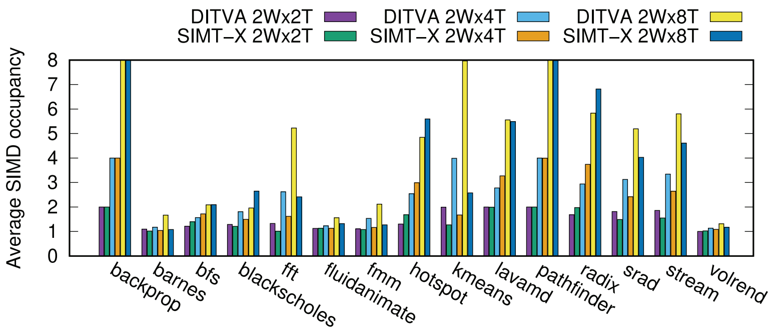


Fig. 6. Average Thread Occupancy per Warp, OoO SIMT-X and in-order DITVA.

majority of configurations, with 20.0% performance improvement for the 2Wx2T OoO configuration vs. in-order, 35.8% for 2Wx4T, and 23.7% for 2Wx8T models. Similarly, Figure 6 demonstrates the potential for dynamic thread vectorization across a majority of the SPMD applications, and its clear advantage over conventional SMT models to improve performance. IPC gains are observed for OoO SIMT for similar reasons to that of traditional OoO architectures: dynamic instruction execution based on data dependencies and better latency tolerance, versus the strict program ordering of in-order execution.

The benchmarks *blackscholes* and *stream* provided the best performance for SIMT-X due to higher ILP and latency tolerance exploited through OoO execution. 2Wx8T demonstrated the best results for these benchmarks as substantial ILP was coupled with higher thread occupancy per warp. *Fmm*, *lavamd*, *pathfinder*, and *volrend* demonstrated higher performance for 2Wx2T and 2Wx4T SIMT-X models, thereafter plateauing for 2Wx8T. For *fmm* and *lavamd*, the 2Wx2T OoO model generally displayed the best outcomes as the microarchitectural overhead for divergent behavior was minimal for two threaded warps. Conversely, *Srad* and *fft* possessed generally low inherent ILP, and when paired with slightly better thread occupancy, as seen in Figure 6, in-order SIMT provided favorable results. In the case of Barnes, lower thread occupancy per warp was observed (1.04 threads per warp), which was fairly on par with SMT. Consequently, lower thread level parallelism limited the performance advantage of 2-warp SIMT-X and displayed similar results to an OoO SMT with four independent threads, however, with an improvement over in-order DITVA.

Bfs was an exceptional case that demonstrated better performance with in-order SIMT. *Bfs*' critical path consisted of a set of nested branches and conditional statements, all relying on a series of memory accesses. As discussed in the work of McFarlin et al. [2013], OoO and in-order models

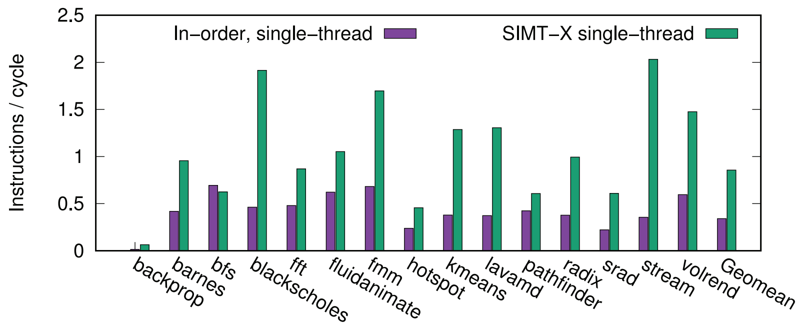


Fig. 7. Single-thread performance of OoO SIMT-X and in-order DITVA.

exhibit varying execution schedules when cache misses or branch mispredictions are encountered. Accordingly, SIMT-X experienced a higher branch misprediction rate on the critical path in comparison to DITVA, negatively affecting cache locality and memory accesses. For this reason, DITVA was able to achieve better performance with in-order execution.

Much like OoO SMT processors, less performance benefits exist for SIMT-X as threads scale to 16T. As described by Hily and Seznec [1999], throughput of an in-order SMT core approaches that of an OoO equivalent as threads scale. Although OoO hardware enables early issue, a significant fraction of the instructions are only ready for issue once all false dependencies have resolved, thus issuing in-order as threads scale [Sleiman and Wenisch 2016]. Consequently, when testing for the case of 4Tx4W (not shown here), performance degradation for SIMT-X (*−14%* vs. DITVA) became evident as instruction flow transitioned into a rather in-order state, and the microarchitecture incurred the overhead associated with OoO SIMT.

Overall, the results and geometric means presented in Figure 5 demonstrate that OoO SIMT execution enables similar or better performance to traditional in-order SIMT with fewer threads per core. For instance, the 4T OoO SIMT configuration, 2Wx2T, achieves better performance on average than 2Wx4T in-order SIMT, and similarly for 2Wx4T OoO SIMT in comparison to 2Wx8T in-order. Although performance plateaus toward 16T, SIMT-X's 2Wx4T configuration is able to achieve the performance of 2Wx8T in-order SIMT.

4.2 Single-thread Performance

As the purpose of SIMT-X is to bridge the latency-throughput gap, Figure 7 presents the performance of OoO SIMT-X and in-order DITVA for the serial versions of the same benchmarks. As seen in the figure, SIMT-X successfully provides higher single-thread performance in comparison to DITVA, with a geometric improvement of 1.5× across all benchmarks.

A significant single-thread performance gain is observed for blackscholes, lavamd, stream, and volrend as the benchmarks possess higher inherent ILP, and benefit from memory latency tolerance. As previously discussed, *fft* and *srad* possessed lower ILP and thus has limited OoO execution benefits, whereas *bfs* was marginal due to restrained memory-bandwidth.

SIMT-X performance on single-thread workload is virtually identical to an equivalently-configured superscalar core, as the PIRAT holds a single physical mapping per architectural register, and no extra μops are emitted. In general, as the divergence predictor quickly trains for single-thread performance, SIMT-X's front-end logic avoids uop generation and transitions into the role of a conventional OoO processor (PIRAT transforming into a RAT) to improve latency-critical sections of SPMD applications. On heterogeneous multi-thread workloads, SIMT-X can run a single thread per warp to behave as a conventional SMT core.

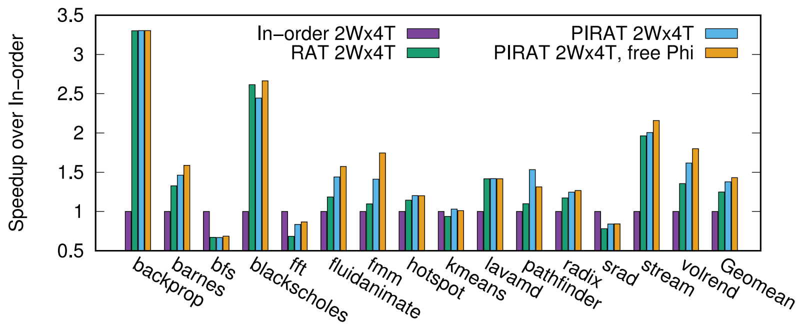


Fig. 8. Performance comparison for renaming methods of 2W×4T, SIMT-X.

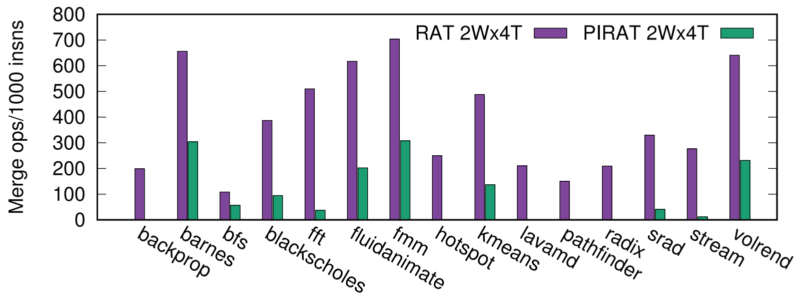


Fig. 9. RAT vs. PIRAT merge operation frequency for 2W×4T, SIMT-X.

4.3 PIRAT vs. Naive Rename

This section provides a comparison of renaming and merging strategies for OoO SIMT. Specifically, the PIRAT implementation of SIMT-X is compared to: (1) a "RAT" implementation signifying the naive register renaming method presented in Section 3.1, and (2) a "Free merge" representation of SIMT-X, which performs cost-free merges, avoiding rename stalling, and any performance loss due to merge-uop injection within the pipeline (elimination of reservation station, functional unit, and ROB occupancy effects). As >2 merges are generated infrequently per rename block (~0.238%), "Free merge" presents a close approximation to cost-free merging.

Figure 8 presents the speedup for the 2Wx4T SIMT-X over the in-order SIMT architecture DITVA, for the renaming strategies listed. Similarly, Figure 9 outlines the frequency of merges required per 1K instructions for both the naive and PIRAT implementations of SIMT-X.

Figure 8 illustrates the advantage of PIRAT renaming over the RAT, providing a 10.31% speedup. As previously discussed in Section 3.1, the naive register merging approach serializes the rename stage, where a significant amount of merge μops (a geometric mean of 4.21× more merges, as seen in Figure 9) are required for processing in comparison to the PIRAT implementation. Nevertheless, the RAT implementation of an 8T SIMT-X provided an average speedup of approximately 23.1% over in-order SIMT, with exceptional in-order cases as discussed in the previous section. Figure 8 also outlines that the PIRAT performance is within 3.86% of a theoretical "free merge," cost-less merging SIMT-X microarchitecture. Therefore, although SIMT-X implements a restricted form of OoO execution, results suggest that a two-entry PIRAT is both a sufficient and efficient approach to renaming in the OoO SIMT context.

In the case of *pathfinder* and *kmeans*, the free merge approach contributed to a significant amount of rename stalls (approximately 3.5x more) generated by the lack of physical registers,

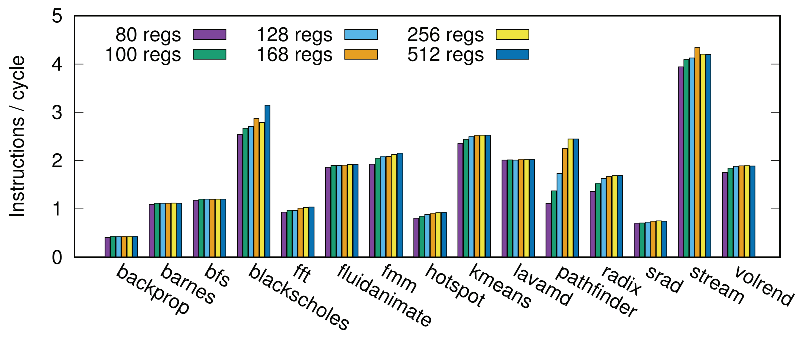


Fig. 10. Performance scaling with physical register count for 2W×4T, SIMT-X.

where PIRAT renaming provided better performance. Although the PIRAT may require WAW and/or >1 merge per architectural register stalls, these cases remain negligible (roughly 3.77% of all PIRAT accesses) and have a minor effect on performance due to pipeline buffering.

The main reason why so few merge operations are actually needed is that the vast majority of branches do not diverge. The ratio of divergent branches ranges from 0.003% (radix) to 6.7% (srad), with an average of 1.4%. The performance results presented in these sections provide motivation for OoO SIMT integration within heterogeneous systems: better latency and throughput may be supported on a core when directly compared to in-order SIMT, while avoiding common issues of data offloading, communication overhead, and programmability. Thus, SIMT-X successfully closes in on the latency-throughput gap with a manageable amount of threads per core.

4.4 Performance Scaling with Physical Registers

Maintaining up to two concurrent physical register mappings per architectural register may increase pressure on the physical register file. The results presented in the previous sections implement SIMT-X with a 168 physical register pool for renaming (Table 1), as the case of many commercial x86_64 processors. Figure 10 examines SIMT-X's performance by scaling the number of physical registers from 80 registers to 512 to evaluate the sensibility to register file size, considering a 2Wx4T SIMT-X.

As seen throughout the figure, most benchmarks can tolerate a reduced register file size. For the memory-bound benchmarks such as *backprop* and *bfs*, performance shows no sensible variation after 100 registers. There are certain cases, however, such as *blackscholes*, *pathfinder*, and *stream* where performance loss is exhibited due to more PIRAT/WAW induced stalls. Overall, results demonstrate that the standard 168 physical register baseline for x86_64 is efficient for SIMT-X performance.

4.5 Area and Energy

Compared to the baseline 4-thread SMT core, $2W \times 4T$ SIMT-X widens scalar execution units and physical register files, banks the L1 data cache, and has all threads monitoring the load-store queue. It also adds 8x84-bit PST, 16x9-bit Pathtable, 32x2-bit divergence predictor, and the 128x8-bit RATs become 64x21-bit PIRATs. We modeled these changes using the McPAT power estimation tool [Li et al. 2009] to analyze the overhead of additional hardware structures required in the pipeline. We consider the 45nm technology node.

Using McPat, the SIMT-X processor area and peak power respectively increase by 7% and 16%, mostly due to the larger FP-SIMD register file and load-store unit. Figure 11 presents the energy of

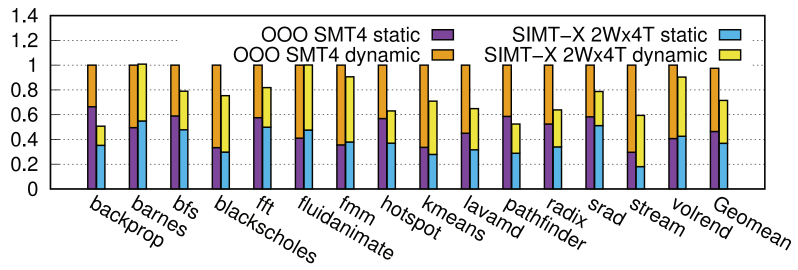


Fig. 11. Energy for 2W×4T, SIMT-X relative to 4T SMT.

SIMT-X execution relative to the OoO SMT baseline. The reduction in dynamic instruction count and overall speedup largely offset the overhead to yield a mean 22% energy reduction.

5 RELATED WORK

Multiple other research efforts have sought to generalize the SIMT execution model for general-purpose multi-thread workloads. Minimal Multi-Threading (MMT) is an OoO SMT model that synchronizes application threads to avoid redundant instruction fetch and decode in the front-end [Long et al. 2010]. Instructions execute as one in the back-end if threads possess the same data; otherwise, the instruction is disassembled and treated as independent scalar instructions. SIMT-X always preserves vector instructions throughout the pipeline, taking advantage of SIMD execution units.

Independent thread scheduling, as implemented in the Nvidia Volta and Turing GPUs, addresses some of the issues of compiler-based vectorization by allowing fine-grained synchronization between threads in a warp [ElTantawy and Aamodt 2016; NVIDIA 2017]. The flexibility comes at the cost of limiting opportunities for the compiler to avoid false dependencies by zeroing-out unused vector lanes, as in explicit SIMD instruction sets like AVX-512. Previously proposed extensions to the SIMT model also execute instructions in-order: Execution Drafting [Mckeown et al. 2014] synchronizes multi-threaded applications running identical code using an in-order core, where lockstep execution is allowed at arbitrary addresses. DITVA [Kalathingal et al. 2017] extends upon an in-order SMT to dynamically vectorize SPMD binaries for SIMT-like execution. Loop-Task Accelerators [Kim et al. 2017] are closely-coupled accelerators that compromise between spatial and temporal decoupling. SIMT-X addresses various SIMT issues using an OoO model and supporting a traditional software stack.

Compilers have been proposed to target CPU SIMD instruction sets from C languages with SPMD constructs [Pharr and Mark 2012; Karrenberg and Hack 2011]. They support control divergence by statically emitting code that performs lane masking. Although recent instruction sets like AVX-512 offer masked instructions for this purpose, OoO microarchitectures that implement these instruction sets still face the challenge of partial dependencies that we present in Section 2.3. Conversely, SIMT-X addresses the partial dependency problem and supports SPMD applications without special compilers, language extensions, where transformations of control-flow, vectorization, and register definitions are handled by the microarchitecture.

The augmented RAT renaming mechanism used by Wang et al. [2001] for predicated OoO execution is closest to the PIRAT, but only considers scalar code. The augmented RAT stores up to four physical register definitions per architectural register and their physical identifiers. A merge is requested when two or more slots have been occupied after predication resolution. SIMT-X's PIRAT deliberately limits renaming to two physical registers per architectural register for vector-like execution, while deferring merging only until necessary, enabling a much simpler implementation that still captures most of the benefits of OoO execution.

OoO vector architectures, introduced in the 1990s [Espasa et al. 1997], have demonstrated that performance may be greatly improved when compared to in-order counterparts. These machines consider contiguous wide vectors and custom vector instruction sets. However, support for today's irregular parallel application makes vector masking support a necessity. As discussed in Section 2.3, such masked instructions cause implicit dependencies with older register definitions, incurring high performance penalties [Intel Corporation 2017]. SIMT-X's microarchitecture avoids the vast majority of artificial dependencies induced by register merging through dynamic masking logic, supporting ISA extensions and dynamically vectorizing instructions for OoO SIMT execution.

Finally, another approach to vectorization is speculative vectorization from serial code. Microarchitecture-level implementations of speculative vectorization let independent instructions from multiple iterations of a loop execute as a single vector and commit as scalar instructions in program order [Vajapeyam et al. 1999; Pajuelo et al. 2002]. Microarchitecture mechanisms to detect and preserve control-flow independent instructions have been proposed [Pajuelo et al. 2005]. However, we are not aware of any equivalent to the data merging after control-flow reconvergence mechanism of SIMT-X in that context. The ARM SVE instruction set offers architectural support for speculative vectorization [Stephens et al. 2017]. As it relies on masked instructions, it faces the challenges exposed in Section 2.3 with respect to partial dependencies.

6 CONCLUSION

Heterogeneous architectures consist of CPUs that execute latency-critical application phases, and GPUs that are used for throughput-oriented execution. These architectures often incur communication and synchronization overhead, programmability issues, and insufficient GPU single-thread performance. This work proposed a general purpose OoO SIMT microarchitecture, referred to as SIMT Express (SIMT-X), to bridge the latency/throughput gap on a single core.

SIMT-X applies a multi-threaded OoO pipeline to the SIMT model, vectorizing the front-end and leveraging existing SIMD units in the back-end for warp-like execution. The SIMT-X microarchitecture presented in this work provides a solution for dynamically managing concurrent paths/warps in an OoO SIMT context, while handling partial dependency issues using a PIRAT. Although SIMT-X implements a restricted form of OoO SIMT, results demonstrate that the simplified microarchitecture successfully captures majority of the benefits of OoO execution while maintaining full binary compatibility with general-purpose ISAs.

Experimental results display that SIMT-X provides better throughput performance than a conventional in-order SIMT for up to 16 threads per core. The 2Wx4T SIMT-X configuration was the most cost-effective design, yielding higher performance than in-order SIMT equivalents, and providing comparable performance with 16T in-order models. On average, the 2Wx4T provided 35.8% higher IPC than in-order SIMT and 1.5x better single-thread latency. SIMT-X therefore provides strong motivation for considering OoO SIMT integration within heterogeneous multi-core systems, bridging the throughput and latency gap on a single core.

REFERENCES

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*. ACM, 72–81.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. *IEEE Workload Characterization Symposium* 0 (2009), 44–54. DOI: https://doi.org/10.1109/IISWC.2009.5306797

Ahmed ElTantawy and Tor M. Aamodt. 2016. MIMD synchronization on SIMT architectures. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*.

Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, et al. 2002. Tarantula: A vector extension to the alpha architecture. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. IEEE, 281–292.

Roger Espasa, Mateo Valero, and James E. Smith. 1997. Out-of-order vector architectures. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, 160–170.

Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2009. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Transactions on Architecture and Code Optimization (TACO)* 6, 2 (2009), 7.

Azzam Haidar, Ahmad Abdelfatah, Stanimire Tomov, and Jack Dongarra. 2017. High-performance Cholesky factorization for GPU-only execution. In *Proceedings of the General Purpose GPUs*. ACM, 42–52.

Sébastien Hily and André Seznec. 1999. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture, 1999*. IEEE, 64–67.

Intel Corporation. 2017. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

S. Kalathingal, S. Collange, B. N. Swamy, and A. Seznec. 2017. DITVA: Dynamic inter-thread vectorization architecture. *J. Parallel and Distrib. Comput.* (2017).

Ralf Karrenberg and Sebastian Hack. 2011. Whole-function vectorization. In *CGO*. IEEE, 141–150.

Ji Kim, Shunning Jiang, Christopher Torng, Moyang Wang, Shreesha Srinath, Berkin Ilbeyi, Khalid Al-Hawaj, and Christopher Batten. 2017. Using intra-core loop-task accelerators to improve the productivity and performance of task-based parallel programs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 759–773.

Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Stephen W. Keckler, and Krste Asanovic. 2014. Exploring the design space of SPMD divergence management on data-parallel architectures. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 101–113.

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009. MICRO-42*. IEEE, 469–480.

John Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro.* 28, 2 (2008), 39–55. DOI: https://doi.org/10.1109/MM.2008.31

Guoping Long, Diana Franklin, Susmit Biswas, Pablo Ortiz, Jason Oberg, Dongrui Fan, and Frederic T. Chong. 2010. Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 337–348.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.* 40, 6 (2005), 190–200. DOI: https://doi.org/10.1145/1064978.1065034

Daniel S. McFarlin, Charles Tucker, and Craig Zilles. 2013. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. 241–252.

Michael Mckeown, Jonathan Balkind, and David Wentzlaff. 2014. Execution drafting: Energy efficiency through computation deduplication. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. 432–444. DOI: https://doi.org/10.1109/MICRO.2014.43

T. Milanez, S. Collange, F. M. Q. Pereira, W. Meira, and R. Ferreira. 2014. Thread scheduling and memory coalescing for dynamic vectorization of SPMD workloads. *Parallel Comput.* 40, 9 (2014), 548–558.

Sparsh Mittal and Jeffrey S. Vetter. 2015. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.* 47, 4, Article 69 (July 2015), 35 pages. DOI: https://doi.org/10.1145/2788396

John Nickolls and William J. Dally. 2010. The GPU computing era. *IEEE Micro.* 30 (March 2010), 56–69. Issue 2. http://dx.doi.org/10.1109/MM.2010.41.

NVIDIA2017. *NVIDIA Tesla V100 GPU Architecture Whitepaper*. NVIDIA.

Alex Pajuelo, Antonio González, and Mateo Valero. 2002. Speculative dynamic vectorization. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. IEEE, 271–280.

Alex Pajuelo, Antonio González, and Mateo Valero. 2005. Control-flow independence reuse via dynamic vectorization. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 10pp.

Matt Pharr and William R. Mark. 2012. ISPC: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*. IEEE.

Nathanaël Prémillieu and André Seznec. 2012. SYRANT: SYmmetric resource allocation on not-taken and taken paths. *ACM Trans. Archit. Code Optim. (TACO)—HIPEAC Papers* 8, 4 (2012). DOI: https://doi.org/10.1145/2086696.2086722

Nathanael Prémillieu and André Seznec. 2014. Efficient out-of-order execution of guarded ISAs. *ACM Trans. Archit. Code Optimization* 11 (12 2014), 1–21. DOI: https://doi.org/10.1145/2677037

E. Safi, A. Moshovos, and A. Veneris. 2011. Two-stage, pipelined register renaming. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* 19, 10 (2011), 1926–1931. DOI: https://doi.org/10.1109/TVLSI.2010.2062545

André Seznec. 2011. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. 117–127.

Faissal M. Sleiman and Thomas F. Wenisch. 2016. Efficiently scaling out-of-order cores for simultaneous multithreading. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 431–443.

N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. 2017. The ARM scalable vector extension. *IEEE Micro.* 37, 2 (Mar 2017), 26–39. DOI: https://doi.org/10.1109/MM.2017.35

Sriram Vajapeyam, P. J. Joseph, and Tulika Mitra. 1999. Dynamic vectorization: A mechanism for exploiting far-flung ILP in ordinary programs. In *Proceedings of the 26th International Symposium on Computer Architecture*. 16–27.

Perry H. Wang, Hong Wang, Ralph-Michael Kling, Kalpana Ramakrishnan, and John Paul Shen. 2001. Register renaming and scheduling for dynamic execution of predicated code. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 15–25.

Henry Wong and Tor M. Aamodt. 2009. The performance potential for single application heterogeneous systems. In *Proceedings of the 8th Workshop on Duplicating, Deconstructing, and Debunking*.

Received September 2019; revised March 2020; accepted March 2020