Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance

Jiayuan Meng
Department of Computer Science
University of Virginia
jm6dg@virginia.edu

David Tarjan*
Department of Computer Science
University of Virginia
dtarjan@cs.virginia.edu

Kevin Skadron
Department of Computer Science
University of Virginia
skadron@cs.virginia.edu

ABSTRACT

SIMD organizations amortize the area and power of fetch, decode, and issue logic across multiple processing units in order to maximize throughput for a given area and power budget. However, throughput is reduced when a set of threads operating in lockstep (a warp) are stalled due to long latency memory accesses. The resulting idle cycles are extremely costly. Multi-threading can hide latencies by interleaving the execution of multiple warps, but deep multi-threading using many warps dramatically increases the cost of the register files (multi-threading depth × SIMD width), and cache contention can make performance worse. Instead, intra-warp latency hiding should first be exploited. This allows threads that are ready but stalled by SIMD restrictions to use these idle cycles and reduces the need for multi-threading among warps. This paper introduces *dynamic warp subdivision* (DWS), which allows a single warp to occupy more than one slot in the scheduler without requiring extra register file space. Independent scheduling entities allow divergent branch paths to interleave their execution, and allow threads that hit to run ahead. The result is improved latency hiding and memory level parallelism (MLP). We evaluate the technique on a coherent cache hierarchy with private L1 caches and a shared L2 cache. With an area overhead of less than 1%, experiments with eight data-parallel benchmarks show our technique improves performance on average by 1.7X.

Categories and Subject Descriptors: C.1.2 PROCESSOR ARCHITECTURES: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Design, Performance

Keywords: SIMD, Branch Divergence, Latency Hiding, Memory Divergence, Warp

1. INTRODUCTION

Single instruction, multiple data (SIMD) organizations use a single instruction sequencer to operate multiple datapaths or "lanes" in lockstep. SIMD is generally more efficient than *multiple instruction, multiple data* (MIMD) in exploiting data parallelism, because it allows greater throughput within a given area and power budget by amortizing the cost of the instruction sequencing over multiple datapaths. This observation is becoming important, both because data parallelism is common across a wide range of applications; and because data-parallel throughput is increasingly important for high performance as single-thread performance improvement slows.

SIMD lockstep operation of multiple datapaths can be implemented with vector units, where the SIMD operation is explicit in the instruction set and a single thread operates on wide vector registers. SIMD lockstep can also be *implicit*, where each lane executes distinct threads that operate on scalar registers, but the threads progress in lockstep. The latter is also referred to by NVIDIA as *single instruction multiple threads* (SIMT). For purpose of generality in this paper, we refer to the set of operations happening in lockstep as a *warp* and the application of an instruction sequence to a single lane as a *thread*. We refer to a set of hardware units under SIMD control as a *warp processing unit* or WPU.[1] SIMD organizations of both types are increasingly common in architectures for high throughput computing, exemplified today in STI's Cell Broadband Engine (CBE) [11], Clearspeed's CSX600 [20], Intel's Larrabee [27], Cray [26], and Tarantula [9]. Graphics processors (GPUs), including NVIDIA's Tesla [8] and Fermi [1], and AMD/ATI's recent architectures [2] also employ SIMD organizations and are increasingly used for general-purpose computing. Academic researchers have also proposed stream architectures that employ SIMD organizations [13, 25], and in fact this history goes back to the Illiac project [21]. For both productivity and performance purposes, an increasing number of SIMD organizations support *gather and scatter*, where each lane can load from or store to unrelated addresses. (Vector architectures implement this by loading or storing a vector of data from or to a vector of addresses [7, 8, 27].) This introduces the possibility of "divergent" memory access latency, because a SIMD gather or scatter may access a set of data that is not fully in a particular level of the cache.

As in other throughput-oriented organizations that try to maximize thread concurrency and hence do not waste area on discovering instruction level parallelism, WPUs typically employ in-order pipelines that have limited ability to execute past L1 cache misses or other long latency events. To hide memory latencies, WPUs instead time-multiplex among multiple concurrent warps, each with their own PCs and registers. However, the multi-threading depth (i.e., number of warps) is limited because adding more warps multiplies the area overhead in register files, and may increase cache contention as well. As a result of this limited multi-threading depth, the WPU may run out of work. This can occur even when there are runnable threads that are stalled *only* due to SIMD lockstep restrictions. For example, some threads in a

*Now with NVIDIA Research

[1]We invent a new term here to distinguish it from "cores", "lanes" or "PEs," terms which are sometimes used to refer to individual scalar pipelines that constitutes the WPU.

warp may be ready while others are still stalled on a cache miss.

This paper observes that when a WPU does not have enough warps with all threads ready to execute, some individual threads may still be able to proceed. This occurs in two cases:

- Branch divergence: Branch divergence occurs when threads in the same warp take different paths upon a conditional branch. In current organizations, the WPU can only execute one path of the branch at a time for a given warp, with some threads masked off if they took the branch in the alternate direction. In array organizations, this is handled in hardware by a re-convergence stack [10] or conditional streams [12]; in vector organizations, this is handled in software by using the branch outcomes as a set of predicates [15, 28, 29]. In either case, allowing both paths to run creates problems in re-converging the warp.
- Memory latency divergence: Memory latency divergence occurs when threads from a single warp experience different memory-reference latencies caused by cache misses or accessing different DRAM banks. In current organizations, the entire warp must wait until the last thread has its reference satisfied. This occurs in both array and vector organizations (if the vector instruction set allows gather/scatter).

We propose *dynamic warp subdivision* (DWS) to utilize *both* thread categories. Warps are selectively subdivided into *warp-splits*. Each has fewer threads than the available SIMD width, but can be *individually regarded as an additional scheduling entity* to hide latency and can both be active:

- Upon branch divergence, a warp can be divided into two *active* warp-splits, each representing threads that fall into one of the branch paths. The WPU can then interleave the computation of different branch paths to hide memory latency.
- Upon memory latency divergence, a warp can be divided into two warp-splits as well: one represents threads whose memory operations have completed, the other represents threads that are still stalled (e.g., on a cache miss). The former warp-split need not suspend; it can run ahead non-speculatively and in the process touch and potentially prefetch cache lines that may also be needed by threads that fell behind.

In both cases, stall cycles are reduced, latency hiding is improved, and the ability to overlap more outgoing memory requests increases memory level parallelism (MLP). The challenge is to manage this process without reducing overall throughput: aggressive subdivision may result in performance degradation because it may lead to a large number of narrow warp-splits that only exploit a fraction of the SIMD computation resources. A dynamic mechanism is needed because the divergence pattern depends on run-time dynamics such as cache misses and may vary across applications, phases of execution, and even different inputs.

We evaluate several strategies for dynamic warp subdivision based upon eight distinct data-parallel benchmarks. Experiments are conducted by simulating WPUs operating over a two-level cache hierarchy that has private L1 caches sharing an inclusive, on-chip L2. The results show that our technique improves the average performance across a diverse set of parallel benchmarks by 1.7X. It is robust and shows no performance degradation on the benchmarks that were tested. We estimate that dynamic warp subdivision adds less than 1% area overhead to a WPU.

2. IMPACT OF MEMORY LATENCY DIVERGENCE

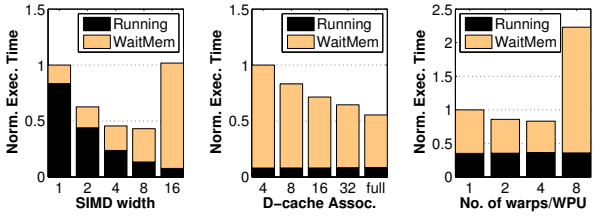


Figure 1: (a) A wider SIMD organization does not always improve performance due to increased time spent waiting for memory. The number of warps is fixed at four. (b) 16-wide WPUs with 4 warps even suffer from highly associative D-caches. (c) A few 8-wide warps can beneficially hide latency, but too many warps eventually exacerbates cache contention and increases the time spent waiting for memory. Results are harmonic means across the benchmarks listed in Table 2 on the system configuration in Table 3.

As SIMD width increases, the likelihood that at least one thread will stall the entire warp increases. However, *inter*-warp latency tolerance (*i.e.*, deeper multi-threading via more warps) is not sufficient to hide these latencies. The number of threads whose state fits in an L1 cache is limited. That is why *intra*-warp latency tolerance is needed. Intra-warp latency also provides opportunities for memory-level parallelism (MLP) that conventional SIMD organizations do not.

In order to illustrate the impact of memory latency divergence, the limitations on warp count, and the need for intra-warp latency tolerance, Figure 1a shows the performance scaling when varying the SIMD width from 1 to 16 with four warps sharing a 32 KB L1 D-cache. As the SIMD width grows, the overall execution time first improves (thanks to greater throughput) but eventually starts to get worse even though this experiment is increasing the total computation resources; although the time spent in SIMD computation keeps decreasing, the time spent waiting for memory drastically increases and eventually dominates. The reasons for this trend are two-fold: first, a wider SIMD organization incorporates more threads, which increase D-cache contention; second, wider warps are more likely to incur memory divergence and suspend due to individual cache misses. The overall effect is fewer active warps but more latency to hide. Figure 1c shows that adding more warps does exacerbate L1 contention. This is a capacity limitation, not just an associativity problem, as shown in Figure 1b, where time waiting on memory is still significant even with full associativity.[2] These results are all averages across the benchmarks described in Section 3.2 and obtained with the configuration described in Section 3.

Intra-warp latency tolerance hides latencies without requiring extra threads. However, intra-warp latency tolerance is only beneficial when threads within the same warp

[2]It is true that current GPU architectures such as Tesla [8] and Fermi [1] have much higher warp counts because they reference global memory frequently (no L2 for Tesla and L2 is only a victim cache in Fermi), and have very long WPU latencies to issue back to back instructions from the same thread. If we increase L1 miss latency to 300 cycles (similar to having no L2), our architecture's optimal warp count also jumps to 8 or 16; the extra cache contention is now justified by the dramatic need for extra latency hiding. Intra-warp latency hiding is still beneficial in this case, of course.

	FFT	Filter	HotSpot	LU	Merge	Short	KMeans	SVM
Avg. instruction count between branches	59	12	16	53	9	19	10	12
Percentage of divergent branches	3.4%	0%	1.4%	4.3%	13.1%	22.0%	2.0%	4.3%
Avg. instruction count between misses	7	27	7	5	45	6	47	11
Avg. instruction count between div. misses	10	30	10	6	75	8	57	17
Percentage of divergent memory accesses	69%	88%	77%	81%	60%	79%	83%	62%

Table 1: Characterization of the frequency of branch divergence and SIMD cache misses.

exhibit divergent behavior. Table 1 shows that many benchmarks exhibit frequent memory divergence. A further advantage of intra-warp latency tolerance is that the same mechanisms also improve throughput in the presence of branch divergence.

In summary, scaling SIMD width is not always beneficial in systems with limited on-chip storage capacity. In order to host wider SIMD organizations and further improve throughput, it is necessary to hide such latency without additional threads. Section 4 and Section 5 show how DWS can address both branch and memory divergence. DWS is the first technique that can address both sources of divergence using the same hardware mechanism.

3. METHODOLOGY

3.1 Overall Architecture

Branch and memory-latency divergence can affect a variety of architectures. In order to draw more general conclusions, this paper models a general system that blends important aspects of modern CPU and GPU architectures: general-purpose ISAs (represented here by the Alpha ISA), SIMD organization, multi-threading, and a cache-coherent memory hierarchy. We model implicit, array-style SIMD instead of a vector architecture because this places fewer burdens on software. While there are simulators available for vector architectures, traditional CPUs, and GPUs, we are not aware of any that combine these aspects. We therefore developed MV5 [17] to simulate WPUs. MV5 is a cycle-accurate, event-driven simulator based on M5 [3]. Because existing thread schedulers do not directly support the management of SIMD threads, applications are simulated in system emulation mode with a set of primitives to create and schedule threads on SIMD resources.

The simulated applications are programmed in an OpenMP-style API implemented in MV5, where data parallelism is expressed in `parallel for` loops. Applications are cross-compiled to the Alpha ISA using G++ 4.1.0, with new instructions recognized by MV5 inserted to signal SIMD thread management to the simulator. The code for a thread is compiled as a scalar code, and the programming model then launches multiple copies for SIMD execution. Neighboring tasks are assigned to threads in the same warp in a way that exploits both spatial and temporal data locality [16].

3.2 Benchmarks

We simulate a set of benchmarks shown in Table 2. The benchmarks are selected from several benchmark suites including MineBench [19], Splash2 [32], and Rodinia [4]. No consideration of SIMD divergence was present in selecting the benchmarks. The benchmarks are the same as those used to study thread scheduling when many threads share a cache [16]. These benchmarks are common, data-parallel kernels and applications. They cover the application domains of scientific computing, image processing, physics simulation, machine learning and data mining. They demonstrate varied data access and communication patterns. We increase the input sizes from the original benchmarks so that we have reasonable simulation times of within six hours. All means reported in this paper are harmonic means.

	Benchmark Description
FFT	Fast Fourier Transform (Splash2 [32]). Spectral methods. Butterfly computation Input: a 1-D array of 65,536 (2^{16}) numbers
Filter	Edge Detection of an Input Image. Convolution. Gathering a 3-by-3 neighborhood Input: a gray scale image of size 500×500
HotSpot	Thermal Simulation (Rodinia [4]). Iterative partial differential equation solver Input: a 300×300 2-D grid, 100 iterations
LU	LU Decomposition (Splash2 [32]). Dense linear algebra. Alternating row-major and column-major computation Input: a 300×300 matrix
Merge	Merge Sort. Element aggregation and reordering Input: a 1-D array of 300,000 integers
Short	Winning Path Search for Chess. Dynamic programming. Neighborhood calculation based on the the previous row Input: 6 steps each with 150,000 choices
KMeans	Unsupervised Classification (MineBench [19]). Distance aggregation using Map-Reduce. Input: 10,000 points in a 20-D space
SVM	Supervised Learning (MineBench [19]). Support vector machine's kernel computation. Input: 100,000 vectors with a 20-D space

Table 2: Simulated benchmarks with descriptions and input sizes.

3.3 Architecture Details

The baseline SIMD architecture used in this study is illustrated in Figure 2. The system has a two level coherent cache hierarchy. Each WPU has private I- and D-caches which interact with an on-chip, shared, last-level cache (LLC). Only the LLC communicates with the off-chip memory. Examples of SIMD organizations that use cache hierarchies include Larrabee [27] and Fermi [1], both of which support general purpose computation. (Fermi's caches, however, are not cache-coherent). WPUs are simulated with up to 64 thread contexts and 16 lanes. The experiments in this paper are limited to four WPUs because this provide reasonable simulation time.

A WPU groups scalar threads into warps. A fetched instruction is executed by threads within the same (active portion of a) warp simultaneously. Each thread's register state resides in a particular lane and the thread must execute in the corresponding lane. The register file is highly banked, with banks corresponding to lanes, so that multiple threads can access their operands at the same time without requiring deep multi-porting. Branch divergence is enabled by a re-convergence stack [10] — using a bit mask, threads that do not fall into the current control path are not executed. The mechanism is described in more detail in Section 4.1. Due to the lack of compiler support, we manually instrument the application code with post-dominators to signal control flow re-convergence after branches.

For the WPU lanes, we model in-order issue of one instruction per cycle. All instructions have latency of one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory controller reordering effects). The 4-way 16 KB L1

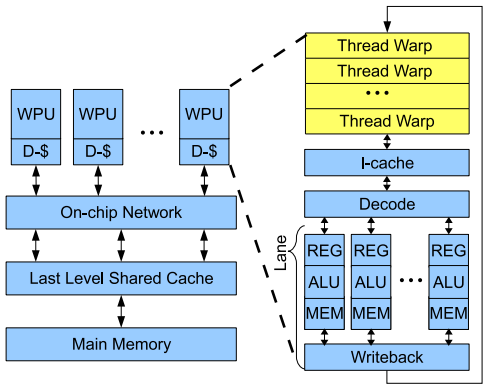


Figure 2: The baseline SIMD architecture groups scalar threads into warps and executes them using the same instruction sequencer. A thread operates over a scalar pipeline or lane that consists of register files, ALUs, and memory units.

I-cache has a 1-cycle latency, while the baseline 8-way, 32 KB, L1 D-cache has a 3-cycle latency. Both caches use 128-byte lines. To hide latency on cache misses, a WPU switches to execute a different warp when the current warp accesses the cache. Switching warps takes no extra latency; the next warp is scheduled while the current warp issues.

This study does not model non-blocking loads whose benefits are limited with in-order issue. The hardware overhead is also considerable as the number of threads (SIMD width × multi-threading depth) goes up. DWS would further increase the state for tracking non-blocking loads by the number of warp-splits allowed. These considerations are interesting areas for future work.

Each WPU has a private I-cache and D-cache. I-caches are not banked because only one instruction is fetched every cycle and distributed across all lanes. D-caches are always banked according to the number of lanes to cater to the high bandwidth demand of memory accesses. We assume there is a crossbar connecting the lanes with the D-cache banks. If bank conflicts occur, these memory requests are serialized and a small queuing overhead (one cycle) is charged. The queuing overhead can be smaller than the hit latency because we assume requests can be pipelined. Each cache bank performs its own tag match to support gather and scatter. All caches are physically indexed and physically tagged and use an LRU replacement policy. Memory coalescing is performed at the L1. All requests from a warp to the same cache line are coalesced in the MSHR. Otherwise, requests to the L2 are serialized. Each MSHR hosts a cache line and can track as many requests to that line as the SIMD width requires. Because the WPU supports gather/scatter, the TLB is multi-ported according to the number of lanes. The L1 caches connect to L2 banks through a crossbar. Coherence uses a directory-based MESI protocol.

Table 3 summarizes the main architectural parameters. Note that the aggregate L2 access latency is broken down into L1 lookup latency, crossbar latency, and the L2 tag and data lookup. The L2 then connects to the main memory through a 266 MHz memory bus with a bandwidth of 16 GB/s. The latency in accessing the main memory is assumed to be 100 cycles, and the memory controller is able to pipeline the requests.

4. DYNAMIC WARP SUBDIVISION UPON BRANCH DIVERGENCE

To hide more latency in the case of insufficient warps, we identify two categories of threads that may be unnecessarily suspended and can actually continue to execute. In this section, we discuss threads suspended due to branch divergence. Section 5 discusses threads suspended due to memory divergence.

Tech. Node	65 nm
WPU	1 GHz, 0.9 V Vdd, Alpha ISA, in-order 64 hardware thread contexts 4 warps with a SIMD width of 16 SIMD threads translate addresses simultaneously
I-Cache	16 KB, 4-way associative, 128 B line size 1 cycle hit latency, 4 MSHRs, LRU, write-back physically tagged, physically indexed
D-Cache	32 KB, 8-way associative, 128 B line size MESI directory-based coherence 3 cycle hit latency, LRU, write-back 32 MSHRs each hosts up to 8 requests physically tagged, physically indexed
L2 Cache	4096 KB, 16-way associative, 128 B line size 30 cycle hit latency, LRU, write-back 256 MSHRs each hosts up to 8 requests
Crossbar	300 MHz, 57 Gbytes/s
Memory	100 cycles access latency

Table 3: Parameters for the two-level coherent cache hierarchy.

4.1 Branch Divergence, Memory Latency and MLP

Figure 3 illustrates the conventional mechanism presented by Fung *et al.* [10] to handle divergent branches. Assume that a warp with a SIMD width of eight encounters a conditional branch at the end of code block A. Six of the threads branch to code block B (indicated by bit mask 11111001 in Figure 3), while the other two threads branch to code block C (indicated by bit mask 00000110 in Figure 3). The WPU first chooses to execute code block B. It then pushes the warp's re-convergence stack with the above two sets of bit masks, with the one that corresponds to code block B on the top of the stack (TOS). The first PC in code block D becomes the *immediate post-dominator* of B. Not until this post-dominator is reached can the re-convergence stack pop and switch to activate threads executing the alternate path (code block C) — in this way, the re-convergence stack is able to handle potentially nested branches within code block B. Eventually code block C re-converges at the same post-dominator into code block D.

The fact that the re-convergence stack can only activate one branch path at a time may limit a WPU's ability to hide latency and leverage MLP. Figure 4 illustrates this scenario. If threads executing code block B miss the cache and all other warps are waiting for memory as well, the WPU has to stall even though the threads that branched into code block C do not suffer from cache misses; these threads could actually continue to execute if it were not for the re-convergence mechanism.

The post-dominator based re-convergence may also undermine MLP when threads that have reached their post-dominator have to wait for those that have not. As shown in Figure 5, if threads executing code block C miss the cache and all other warps are waiting for memory as well, the WPU has to stall even though the threads that finished code block B do not suffer from cache misses. If re-convergence can be relaxed, these threads can make progress themselves, getting their own memory requests issued earlier, as well as prefetching for the others.

It is important to note that DWS upon branch divergence can hide latency and improve MLP *with or without the presence of memory divergence*. As we will discuss in Section 5, DWS upon branch divergence and DWS upon memory divergence are independent, complementary techniques, but DWS allows them to be integrated using the same hard-

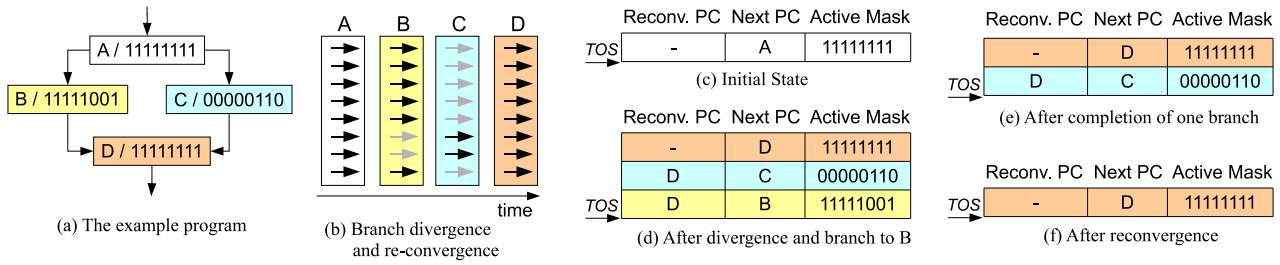


Figure 3: Conventional mechanism to handle branch divergence and re-convergence for a SIMD organization. (a) An example program; (b) the execution trace of diverged threads; (c)-(f) the state of the re-convergence stack. This is adapted from a figure by Fung *et al.* [10].

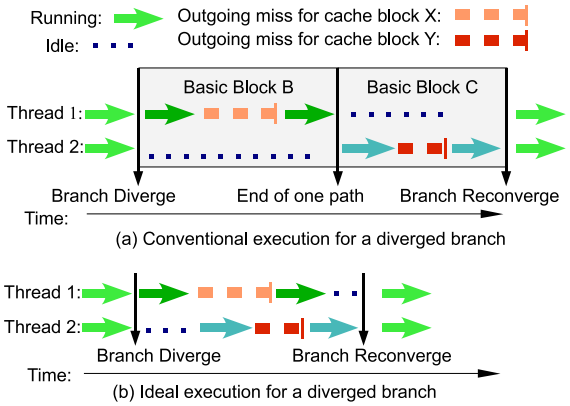


Figure 4: (a) The conventional mechanism serializes the execution of different branch paths. (b) By allowing threads that take different paths to interleave their execution, more latency can be hidden and memory level parallelism can be improved.

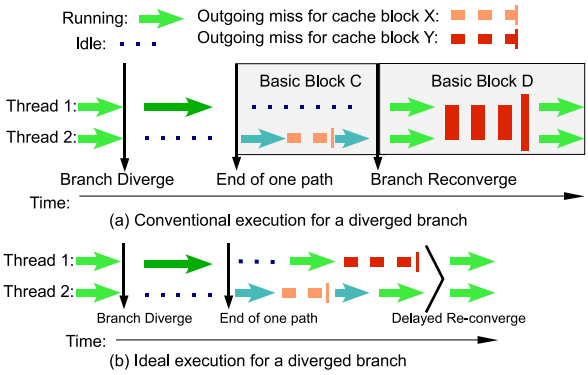


Figure 5: (a) The conventional mechanism forces diverged threads to re-converge at the post-dominator. (b) By allowing threads that reach the post-dominator first to run ahead, they can issue more outgoing requests to leverage memory level parallelism.

ware mechanisms and re-convergence policies. In order to isolate the role of branch divergence, results in this section disable the ability to perform DWS on memory divergence (i.e., warps cannot proceed until all threads' memory operations have completed).

4.2 Relaxing the Re-convergence Stack for Memory Throughput

We propose to subdivide a warp into two warp-splits upon branch divergence. Warp-splits are *independent scheduling entities* and are treated equally as warps by the scheduler. A warp-split can conceivably be recursively subdivided in future divergence events until it consists of only one thread. A full warp can be viewed as a *root warp-split* with full SIMD width. We use the term *SIMD groups* to refer to both full warps and warp-splits in the rest of the paper.

After a divergent branch subdivides a warp, the resulting warp-splits can interleave their computation to hide latency and improve memory level parallelism. This is demonstrated by an example in Figure 6. After a divergent branch in code block B, the conventional approach uses the re-convergence stack to serialize different branch paths (Figure 6b). Alternatively, dynamic warp subdivision uses a warp-split table described in Section 4.4 so that a warp can progress like a binary tree with simultaneously active warp-splits (Figure 6d).

4.3 Over-subdivision

Warps may not benefit from subdivision upon *every* divergent branch — such aggressive subdivision may lead to a large number of narrow warp-splits, which can otherwise run altogether in a wider SIMD group. This phenomenon is referred to in the rest of the paper as *over-subdivision*. Due to the potential for over-subdivision, we subdivide warps conditionally upon selected branches.

A static approach is used to select which branches are allowed to subdivide warps. As a side effect of subdivision, SIMD resources may be under-utilized if warp-splits are not re-converged in time. This occurs when two warp-splits have both passed their common post-dominator, but are not able to re-converge. This is illustrated in Figure 6 when the execution reaches time T2. We therefore use the heuristic that warps only subdivide upon branches whose associate post-dominator is followed up by a short basic block (F in the example of Figure 6) of no more than 50 instructions. This value was chosen because it takes roughly the same time to execute that many instructions as to handle an L1 miss, so run-ahead threads resulting from subdivision can hide some latency and initiate some further memory requests without running too far ahead and potentially inhibiting re-convergence. We manually instrumented the code to identify branches that subdivide warps, but in practice this process would be automated by the compiler.

4.4 Unrelenting Subdivision

When warp-splits independently execute the same instruction sequence while there is not much latency to hide, SIMD resources are under-utilized for little benefit. This scenario

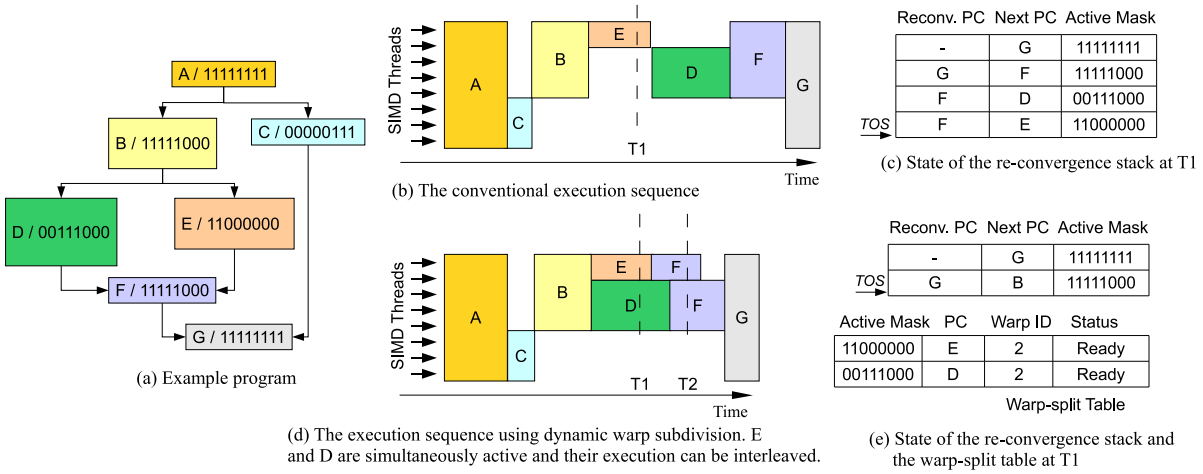


Figure 6: An example of dynamic warp subdivision upon branch divergence. Conventionally, different branch paths in the example program (a) are pushed to different layers in the re-convergence stack (c) and the execution of different paths is serialized (b). With dynamic warp subdivision, branch paths create entries in the warp-split table instead (e), denoting SIMD thread groups that can be simultaneously active. Their immediate post-dominator is ignored as well and no longer enforces re-convergence. As a result, diverged threads can interleave their execution (d), and they no longer have to re-converge at the beginning of code block F. Therefore, F is executed twice by different warp-splits; this may improve MLP but it also risks pipeline under-utilization.

is referred to in the rest of the paper as *unrelenting subdivision*. Under such circumstances, it may be better to re-converge the warp-splits as soon as possible so they can execute together as a wider SIMD group. Therefore it is important *both* to create warp-splits and to re-converge them appropriately.

To meet such requirement, a warp-split table (WST) is introduced *in addition to* the re-convergence stack to keep track of subdivision while preserving the re-convergence mechanism. Each entry in the WST corresponds to a warp-split. It records the warp-split's parent warp, the current PC, and the active mask that indicates the associated threads. Figure 6 illustrates an example of warp subdivision. Upon a divergent branch at the end of code block B, the warp can choose to push the re-convergence stack to serialize code blocks E and D (Figure 6b, c), or it can choose to be subdivided according to the branch outcome so that code blocks E and D can be interleaved (Figure 6d, e). If a warp is subdivided, two additional entries are created in the warp-split table. The re-convergence stack remains intact to avoid imposing a particular execution order of the paths; it can therefore no longer keep track of this and future branches nested within the branch registered on its TOS. As a result, future nested branches as well as their post-dominators are ignored by the re-convergence stack; warp-splits continue executing asynchronously and keep being subdivided upon future divergent branches until they reach the post-dominator associated with the top of the re-convergence stack. At this point, the warp-split stalls waiting to be re-united with other warp-splits once they reach the same post-dominator. The re-convergence is performed by the re-convergence stack in the conventional manner. We name this scheme *stack-based re-convergence*.

4.5 PC-based Re-convergence

Using stack-based re-convergence, warp-splits can eventually re-converge; however, they can re-converge earlier when they happen to execute the same instruction. For example, at time T2 in Figure 6d, both warp-splits are asynchronously executing the same code block and they *might* arrive at the same PC. In such cases, the two warp-splits can be re-united naturally without stalling any of them.

By recording the PC of each warp-split in the warp-split table, the WPU can identify and re-unite warp-splits belonging to the *same* warp that are *ready* to execute and share the same PC. We name this scheme *PC-based re-convergence*. Note that stack-based re-convergence is still used if PC-based re-convergence does not occur. A similar principle has been used in Fung et al.'s dynamic warp formation, although it was used to group *any* threads from *different* warps that execute the same branch path [10].

While comparing multiple PCs may take three to four cycles, such PC-based re-convergence does not have to be checked every cycle, and the latency can usually be hidden. Because the scheduler only switches SIMD groups upon cache accesses, resumed warp-splits from the ready queue always start with memory instructions. As a result, PCs need only be compared when the running warp-split executes a memory instruction. Furthermore, because re-convergence takes place only when there are one or more SIMD groups in the ready queue, the WPU can immediately switch to execute another SIMD group, preferably from a different warp, to hide the latency PC comparisons.

4.6 Results

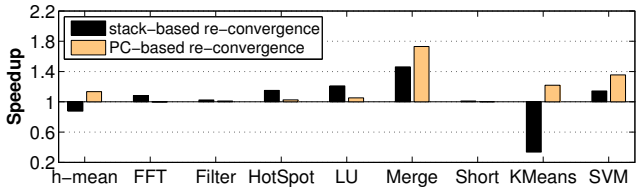


Figure 7: Performance gained by dynamic warp subdivision upon branch divergence. Speedups are normalized to that of the equivalent WPUs without DWS. Compared to stack-based re-convergence, PC-based re-convergence reduces unrelenting subdivision and improves performance significantly without ever making performance worse.

Dynamic warp subdivision upon branch divergence is evaluated with both stack-based re-convergence and PC-based

re-convergence. Figure 7 compares their speedups over a conventional architecture with no DWS. While stack-based re-convergence demonstrates performance gains for some applications, it penalizes the performance of KMeans significantly due to over-subdivision and unrelenting subdivision — the average SIMD width for executed instructions are reduced to 4 for 16-wide WPUs. By introducing PC-based re-convergence, the average SIMD width for executed instructions increases to 9. This drastically reduces the under-utilization of SIMD resources. However, without memory divergence handling, warp-splits re-converged upon the same PC are not able to split again upon cache misses to hide intra-warp latency. As a result, for some applications, stack-based re-convergence performs slightly better than PC-based re-convergence. Overall, PC-based re-convergence outperforms stack-based re-convergence and it generates an average speedup of 1.13X.

Not all benchmarks are sensitive to branch divergence. Table 1 shows that the benchmarks benefiting from DWS frequently encounter conditional branches and the branch outcomes exhibit a significant fraction of divergence. Besides the occurrence of divergent branches, the benefit of DWS is also affected by other run-time dynamics. In the case of KMeans and Merge, memory divergence occurs frequently and there are usually ample instructions for latency hiding between cache misses. Therefore, they benefit significantly from having more warp-splits to hide latency.

5. DYNAMIC WARP SUBDIVISION UPON MEMORY DIVERGENCE

In conventional SIMD implementations, memory divergence stalls an entire warp. We propose to dynamically subdivide warps upon memory divergence so threads that hit can continue execution to hide latency for the threads that missed. This also offers the possibility for run-ahead threads to bring in cache lines that the fall-behind threads will also need. Warp-splits can be subdivided recursively upon future memory divergence.

5.1 Improve Performance upon Memory Divergence

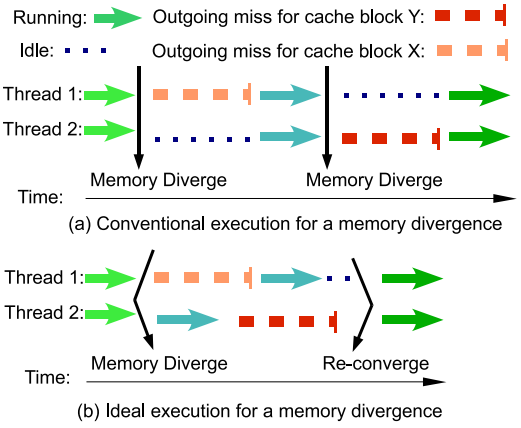


Figure 8: Dynamic warp subdivision upon memory divergence can reduce pipeline stalls and improve MLP. For illustration purposes, the SIMD width is shown as two but similar scenarios exist for wider SIMD organizations as well.

Figures 8 and 9 compare conventional SIMD execution with dynamic warp subdivision upon memory divergence.

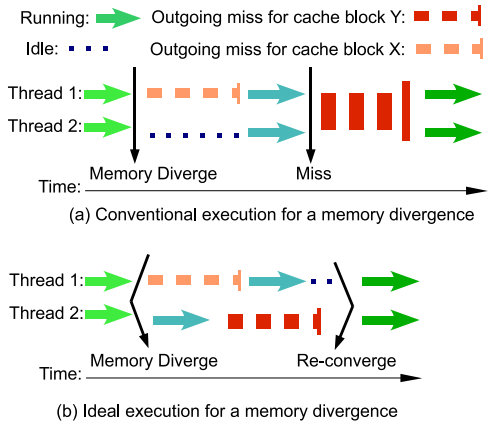


Figure 9: Dynamic warp subdivision upon memory divergence allows run-ahead threads to prefetch cache blocks for the fall-behind.

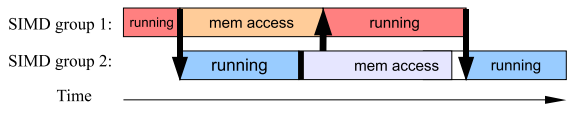
Consider a warp with two memory instructions (not necessarily consecutive). Assuming all other warps have been suspended already, DWS can reduce pipeline stalls and leverage MLP in two scenarios:

- Threads that miss upon the former instruction hit the cache with the latter instruction, while threads that hit first later miss the cache. In this case, DWS allows run-ahead threads to initiate their misses earlier (Figure 8).
- Some threads hit and some miss on the first instruction, but all threads miss on the same cache block with the latter instruction. In this case, the run-ahead warp-splits in DWS not only initiate their misses earlier; they also prefetch data for the fall-behind warp-split (Figure 9). In contrast to speculative precomputation [6] or run-ahead simultaneous threads [23], the run-ahead warp-split always performs *useful* computation and threads' states are saved right away, requiring no ROB or dependency analysis that would otherwise complicate the design of the simple, in-order WPU.

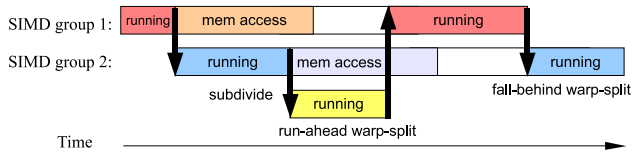
The above scenario is applicable to both array and vector organizations if they support gather loads or scatter stores. Using the same principle as DWS in array organizations, a WPU can use a set of bit masks to mark out vector components that hit the cache so that they can continue their execution and issue more memory requests. The PC upon which the memory divergence occurs can be stored in a table so that those vector components that miss can resume later at the recorded PC afterwards.

However, DWS may not improve performance if the same subset of threads keep missing the cache and requesting disparate addresses. If other SIMD groups are not able to hide sufficient memory latency, the computation *and* the memory latency incurred by those fall-behind threads become the critical path of the execution. As a result, the overall execution time would stay the same even if DWS allows other threads to run ahead.

Similar to branch divergence, unconstrained warp subdivision may lead to over-subdivision. Therefore, we investigate several methods in Section 5.2 to selectively subdivide warps upon memory divergence. On the other hand, to reduce unrelenting subdivision, we exploit different mechanisms in Section 5.3.1 to re-converge warp-splits. Finally, it is important to handle branch divergence correctly even with the introduction of warp subdivision upon memory divergence. This is ensured by re-convergence techniques described in Section 5.3.1.



(a) Example case for conventional SIMT execution



(b) Subdivision upon misses may lead to performance degradation

Figure 10: Performance may degrade if the run-ahead warp-split does not generate long latency memory requests before any outgoing request completes. Such case may take place with LazySplit and ReviveSplit. We demonstrate the case with two warps with arrows pointing to the warp-split that the WPU executes.

5.2 Preventing Over-subdivision

Warp subdivision is not likely to provide benefits when there are already sufficient active warps to hide latency. Aggressively subdividing warps upon every memory divergence regardless of the existence of other active SIMD groups may unnecessarily generate many narrow warp-splits which under-utilize SIMD resources. This brute force subdivision scheme is referred to in the rest of the paper as *AggressSplit*.

To lower the chance of unnecessary warp subdivision, the WPU can create more warp-splits only when there are no other SIMD groups to hide latency. Upon memory divergence, the WPU checks whether all other SIMD groups are waiting for memory. If so, the WPU subdivides the current active warp or warp-split to allow threads that hit to run ahead. This subdivision scheme is referred to as *LazySplit*.

However, LazySplit risks the inability to effectively subdivide warps. If the last active SIMD group has all its threads miss the cache, LazySplit has to stall the WPU's pipeline even though there may be other SIMD groups that have previously incurred memory divergence but were not subdivided and are waiting for memory. We therefore extend LazySplit by allowing it to look for those suspended SIMD groups that are eligible for subdivision when the pipeline is stalled. To avoid over-subdivision, only one SIMD group is subdivided at a time. We name this subdivision scheme *ReviveSplit*.

While ReviveSplit can leverage all SIMD groups that can be subdivided upon memory divergence, its performance may still be suboptimal. Since ReviveSplit attempts to subdivide warps whenever the pipeline is stalled, performance may degrade if the resulting run-ahead warp-split is not able to issue subsequent long latency memory requests in time (*i.e.*, before a suspended SIMD group is resumed by a completed request), as illustrated in Figure 10. In this case, the run-ahead warp-split may occupy the pipeline, keeping those resumed warp-splits from making progress while exploiting no more MLP. Afterwards, the same instruction stream will be executed again by the fall-behind warp-split, increasing the number of executed cycles. However, to make an optimal decision to subdivide warps, foreknowledge or speculation is required to estimate how soon a run-ahead warp-split will encounter another cache miss and how soon outgoing requests will complete and resume awaiting SIMD groups. Compiler analysis, dynamic optimization, or prediction hardware may be able to provide such information, and this is an interesting area for future work.

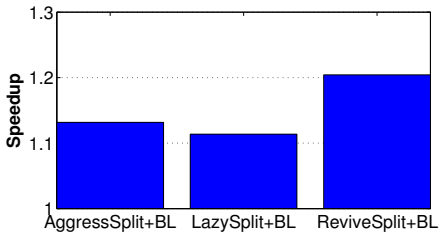


Figure 11: Harmonic means of speedups across all benchmarks. Dynamic warp subdivision upon memory divergence alone has limited benefits due to frequent branches. The BranchLimited re-convergence (specified by *BL*) results in little performance gains for all subdivision schemes, including AggressSplit, LazySplit, and ReviveSplit.

5.3 To Re-converge or To Run Ahead

If warp-splits are not re-converged, they will individually execute the same instruction sequence that could otherwise be run together in a wider warp. If such overhead outweighs the benefits of latency hiding and MLP brought by warp-slits, warp subdivision may penalize performance. On the other hand, if re-convergence is enforced too early, the run-ahead warp-split is likely to stall waiting for the fall-behind warp-split before it can beneficially issue another memory request to overlap outgoing requests. Nevertheless, PC-based re-convergence introduced in Section 4.5 allows multiple warp-splits to run together as a wider SIMD group without the cost of stalling any warp-split. Therefore it is always used in DWS upon memory divergence. However, since PC-based re-convergence does not force a run-ahead warp-split to wait for the fall-behind, it alone may still lead to unrelenting subdivision.

Similar to the process of subdivision, determining the optimal timing to enforce re-converge requires foreknowledge about whether future cache misses can occur in time to overlap with outgoing requests. In this paper, we exploit several heuristics to effectively address re-convergence; some natural places to force warp-splits to re-converge are branches and post-dominators, as discussed below.

5.3.1 Branch Handling after Memory Divergence

Branches may interfere with the re-convergence process of warp-splits that divided upon memory divergence. Upon branches or post-dominators, the re-convergence stack acts implicitly as a barrier for the subdivided warp-splits to re-converge; when warp-splits are synchronized and re-converged, the re-convergence stack can be pushed or popped in a conventional manner. Afterwards, the warp can be subdivided again upon future memory divergence. In other words, only those threads masked on the TOS are subdivided into warp-splits. Since this re-convergence scheme limits a warp-split's lifespan between branches and post-dominators, we name it *BranchLimited re-convergence*.

Although BranchLimited re-convergence allows DWS upon memory divergence while preserving the structure of the re-convergence stack, characterizations show it is likely to limit the benefit in latency hiding and MLP. As shown in Table 1, most benchmarks experience frequent branch instructions with only tens of instructions in between. Given such small basic blocks, a run-ahead warp-split is likely to reach the end of the basic block immediately and stall waiting for the fall-behind warp-split before beneficially issuing further memory requests. As a result, the performance gains from BranchLimited re-convergence are limited, as shown in Figure 11.

5.3.2 Run Ahead Beyond Branches and Loops

If run-ahead warp-splits can proceed beyond branches, they would have a better chance of issuing and overlap-

ping further memory requests with the stall experienced by the fall-behind threads. Fortunately, we have already discussed in Section 4.2 how to use DWS to relax the re-convergence stack for branch divergence. Upon a future divergent branch, the run-ahead warp-split is subdivided into two warp-splits. As a result, the status of the re-convergence stack remains intact while more entries are added to the warp-split table. These warp-splits re-converge when their PCs met. Otherwise, they are forced to wait for others to re-converge once they hit the post-dominator denoted by the re-convergence stack's TOS. We name this re-convergence scheme *BranchBypass*. Note that with BranchBypass, warps can be subdivided upon both memory divergence *and* branch divergence.

By allowing run-ahead warp-splits to proceed beyond branches, it may appear harder for the fall-behind warp-split to catch up with the run-ahead. However, it is important to note that the ability to bypass branches naturally entitles the run-ahead warp-split to bypass loop boundaries into the next iteration. In such cases, the fall-behind warp-split may not have to execute the same number of instructions as the run-ahead to get re-united with it, especially if the loop body is short (*i.e.*, contains only a few instructions). In such a scenario, the run-ahead warp-split may frequently revisit the PC at which the fall-behind warp-split stopped. Once this PC is revisited and the run-ahead warp-split finds the fall-behind split is ready to execute, it re-unites with the fall-behind split immediately, despite potentially being a few iterations ahead of the fall-behind. In contrast to adaptive slip [30], which exploits this same opportunity, DWS warp-splits do not rely on short loops to re-converge: the fall-behind warp-split always resumes itself upon completion of data requests so that it can catch up with the run-ahead split in the case of a large loop body.

5.4 Implementation

Dynamic warp subdivision upon memory divergence is handled *in the same way* as DWS for branch divergence, except that the branch outcome that is used to divide threads is replaced by a *hit mask* that marks threads that hit the cache. Figure 12 illustrates the process. After memory divergence, two warp-splits are created: one with threads that hit and are ready to execute; another with threads that miss and are waiting for memory. Note that after subdivision, the entry of the obsolete parent warp-split is overwritten by one of the resulting child warp-splits so that *the WST only records existing warp-splits.* Moreover, creating a new scheduling entity does not require any other new state. Warp-splits still share the same register file resources, etc. Once the two warp-splits re-unite, their WST entries are merged into one by taking an "or" operation on their active masks. The hardware overhead is discussed in more detail in Section 5.6. There is no need to differentiate warp-splits resulting from memory divergence with those resulting from branch divergence. *Any* warp-split can be further subdivided upon either branch or memory divergence.

Divergence does not break synchronization or communication semantics if programming models do not implicitly *guarantee* threads in a warp to operate in lockstep. Otherwise, DWS is not compatible because it allows warp-splits to proceed asynchronously for variable periods of time. With DWS, inter-thread synchronization or communication is only guaranteed through the use of explicit synchronization primitives, upon which warp-splits simply re-converge. Of course, frequent synchronization will limit intra-warp latency tolerance.

Compared to the baseline architecture, warp-splits merely change the ordering of execution for threads within the same warp. This does not affect memory exceptions. It only affects consistency for threads within the same warp. However, most SIMD programming models do not impose such sequential consistency. Finally, precise traps can still be handled for each individual warp-split.

5.5 Results

In Figure 13, we compare variations of dynamic warp subdivision and characterize the benefits of individual optimizations. DWS upon branch divergence (*DWS.BranchOnly*) alone reaches a speedup of 1.13X. DWS upon memory divergence alone using ReviveSplit (*DWS.ReviveSplit.MemOnly*) achieves a speedup of 1.20X. While DWS for branch and memory divergence are complementary and can be integrated, the overall benefit of DWS is sensitive to the combination of specific subdivision and re-convergence schemes. For example, *DWS.AggressSplit* and *DWS.LazySplit* combine BranchBypass with AggressSplit and LazySplit respectively, and they both lead to performance degradation. Nevertheless, the combination of the best subdivision scheme (ReviveSplit) and re-convergence scheme (BranchBypass) does not harm performance in any case and it achieves an overall speedup of 1.71X (*DWS.ReviveSplit*). On average, *DWS.ReviveSplit* reduces the percentage of time in which WPUs stall waiting for memory from 76% to 36%; in the mean time, the average SIMD width per instruction drops from 14 to 4.

Different benchmarks exhibit different responses to dynamic warp subdivision. Merge benefits mainly from DWS upon branch divergence. KMeans, HotSpot, LU, and Filter benefit mainly from DWS upon memory divergence. FFT and SVM, on the other hand, have many cache misses but a large portion of these misses are not divergent, therefore warp subdivision occurs less often compared to other benchmarks. It is also observed that memory divergence may not be uniformly distributed across SIMD threads due to runtime dynamics. In fact, the pattern varies across benchmarks or even phases of execution. It is therefore difficult to statically pinpoint threads or lanes for warp subdivision.

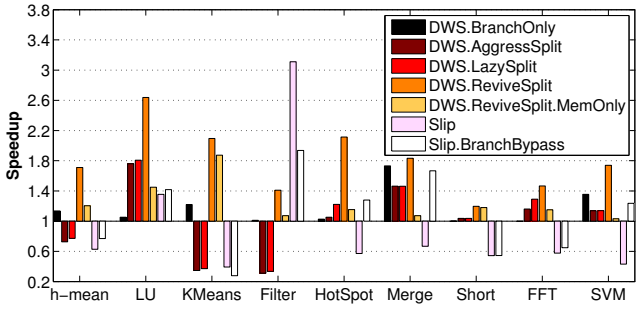


Figure 13: Comparing various DWS schemes. Speedups are normalized to that of equivalent baseline WPUs without DWS. Modest speedups are achieved when DWS applies to branch divergence alone (*DWS.BranchOnly*) or memory divergence alone (*DWS.ReviveSplit.MemOnly*). However, the combination of the two achieves a speedup of 1.71X (*DWS.ReviveSplit*). Slip only outperforms DWS in Filter and it is often subject to performance degradation, even after being modified to bypass branches (*Slip.BranchBypass*). The harmonic mean of the speedups for all benchmarks is shown as `h-mean`.

5.6 Hardware Overhead

Because DWS does not increase the demand for tags and TLB ports, the hardware cost only lies in the WST and the scheduler. For a given WPU, the maximum number of warp-splits may become as large as the number of threads if each warp-split consists of a single thread. With a large number

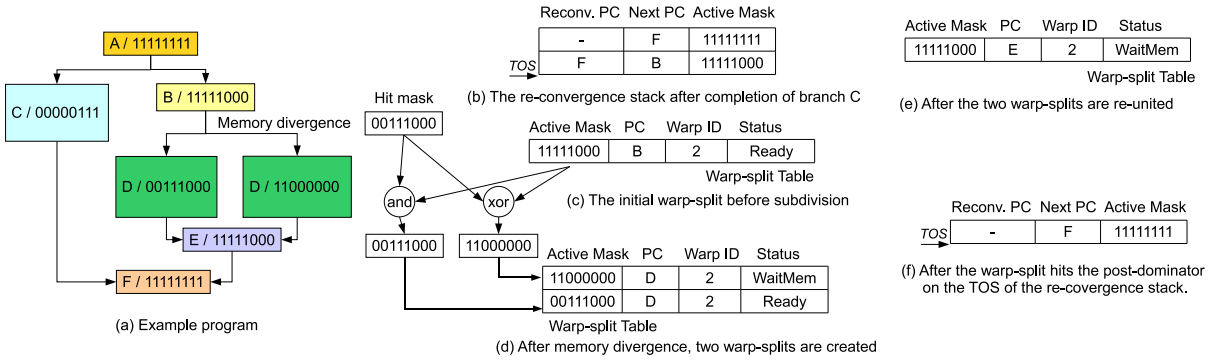


Figure 12: An example of dynamic warp subdivision upon memory divergence. While divergent branches can still be handled using the re-convergence stack (b, f), the warp-split table can be used to create warp-splits using the *hit mask* that marks threads that hit the cache (c-e).

of threads per WPU, this may drastically increase the complexity of the hardware scheduler (*e.g.*, a priority encoder) in order to identify the next ready SIMD group in one cycle. We therefore model a scheduler that only doubles the number of entries in a conventional setting. This approximately doubles the cost of any scheduling structure, but can accommodate more scheduling entities resulting from DWS. In case there are more warp-splits than scheduling slots, the extra warp-splits sit idle until a scheduling slot is available. We also limit the number of WST entries to 16 to reduce its storage cost; warps are not able to be subdivided when the WST is already full. Sensitivity study shows that the above limitations only cost less than 1% of the overall performance.

To estimate area overhead, we measure the realistic sizes for the different units of a core according to a publicly available die photo of the AMD Opteron processor in 130nm technology. We scale the functional unit areas to 65nm, assuming a 0.7 scaling factor per generation. We assume each SIMD lane has a 32 bit data path (adjacent lanes are combined if 64 bit results are needed). We also measure the cache area per 1 KB of capacity and scale that according to the cache capacity. If the WPU has four warps with a SIMD width of 16, each entry needs 16 bits for the active mask, 2 bits for the warp ID, 64 bits for the PC, and 2 bits for the warp status. The resulting size for each WST entry is then 84 bits or 11B. With up to 16 WST entries, the WST state consumes less than 1% of the storage area in a WPU, assuming the WPU has a 32 KB D-cache and a 16 KB I-cache.

5.7 Comparison with Adaptive Slip

Tarjan *et al.* [30] proposed *adaptive slip*, which allows a subset of threads to continue while other threads in the same warp are waiting for memory. Those threads that wait for memory stay suspended until the run-ahead threads in that warp revisit the same memory instruction again, presuming memory divergence mainly occurs within iterative short loops. Their approach presumes aggressive predication so that run-ahead threads can always "slip" into the next iteration regardless of branch divergence.

We compare the performance of DWS to that of adaptive slip without aggressive branch predication, which we refer to in Figure 13 as *Slip*. To adaptively find out the maximum allowed thread divergence for adaptive slip, an interval of 100,000 cycles is used for dynamic profiling. The maximum allowed thread divergence is incremented if the WPU spends more than 70% of the time waiting for memory, and it is decremented if the pipeline actively executes for more than 50% of the time. These thresholds are obtained by experimenting with various combinations and selecting the combination that yields the best performance.

While adaptive slip leads to significant speedup for Filter, it results in performance degradation for many other benchmarks. The reasons are three-fold:

- Without aggressive predication, the run-ahead threads have to stall waiting for the fall-behind upon a conditional branch. This is because the re-convergence stack needs to generate the branch outcome for all threads masked by the TOS. As a result, adaptive slip is hardly effective when the main computation involves frequent conditional branches inside loops. Such is the case with HotSpot, Merge and Short.

- A diverged warp may not be re-united in time. This can be caused by long loops (*i.e.*, loops with a large section of code in each iteration) where slipping into the next iteration may take a long time (*e.g.*, FFT). The same situation may also occur in the case of nested loops where memory divergence takes place inside the outer loop but outside of the inner loop; the fall-behind threads cannot be resumed until the run-ahead threads jump out of the inner loop (*e.g.*, KMeans).

- The appropriate thresholds to increase or decrease the maximum allowed thread divergence vary across benchmarks and hardware configurations. The same thresholds may work well for Filter while penalizing the performances of SVM.

We also attempt to combine adaptive slip with DWS upon branch divergence so that run-ahead threads can continue beyond branches to "slip" into subsequent iterations. This scheme is referred to in Figure 13 as *Slip.BranchBypass*. While this scheme significantly improves performance for many benchmarks, it does not address all the issues with adaptive slip and still harms performance for KMeans, Short, and FFT. Moreover, if memory divergence occurs within a branch path that is rarely visited and the run-ahead threads proceed to the next iteration, it may take a long time before the run-ahead can branch into the same path and reconverge with the fall-behind threads. In this case, this scheme can even perform worse than adaptive slip, as can be observed from KMeans.

6. SENSITIVITY ANALYSIS

The benefit of dynamic warp subdivision in latency hiding and memory level parallelism depends on various factors: the frequency of branch and memory divergence, the length of memory latencies, and the WPU's ability to hide latency with existing warps. We study its sensitivity to various architectural factors. We show that DWS can bring

performance gains in a wide range of configurations. In this section, we use the configuration *DWS.ReviveSplit* to represent the performance of DWS which can subdivide warps upon both branch and memory divergence. We refer to the conventional architecture without DWS as *Conv*. Further sensitivity analysis results appear in an extended, technical report version of this paper [18].

6.1 Cache Misses and Memory Divergence

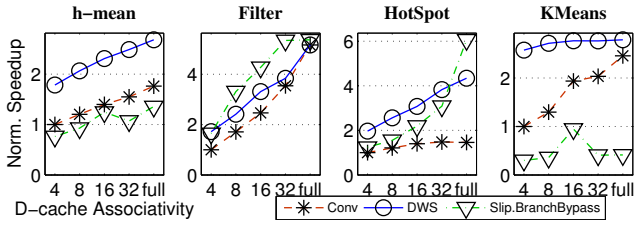


Figure 14: Speedup vs. D-cache associativity. DWS refers to *DWS.ReviveSplit* in Figure 13. Systems are configured according to Table 3. Performance is normalized to each benchmark's execution time under *Conv*. The harmonic mean for the normalized speedup of all benchmarks is shown as `h-mean`. Several individual applications with diverse behavior are also shown.

With larger D-cache associativity, both cache miss rate and memory divergence tend to decrease, and hence there is less latency to hide and less memory level parallelism to exploit. As a result, DWS may occur less often and its benefit may decrease. Figure 14 demonstrates this trend when the D-cache associativity is varied from four to fully-associative.

It can also be observed from Figure 14 that the benefit from DWS does not always increase with smaller D-cache associativity. Because small D-cache associativity leads to more cache misses, threads in a SIMD group are more likely to miss the cache simultaneously, and therefore the the occurrence of divergent memory accesses may decrease. As a result, DWS upon memory divergence is less likely to take effect — in benchmarks such HotSpot, performance of DWS drops more than that of the conventional architecture when the D-cache associativity decreases to four.

6.2 Miss Latency

The amount of latency to hide also affects the effectiveness of DWS. The D-cache miss latency is largely dependent on the memory hierarchy. While most of the evaluation in this paper is based upon a shared L2 cache as the last-level cache, there are various alternatives. Some architectures use private L2 caches while others do not have L2 caches at all and L1 misses are sent directly to the main memory [8]. The L1 miss latency can therefore vary from a few cycles to hundreds of cycles.

We study the sensitivity of the L1 miss latency by varying the L2 lookup latency from 10 cycles to 300 cycles. Results in Figure 15 shows that while DWS also suffers from longer miss latency, its speedup compared to the equivalent conventional architecture *increases*. This is not surprising because more SIMD groups are needed to hide longer latency; DWS achieves this by generating more warp-splits on demand.

7. RELATED WORK

Fung *et al.* [10] addressed the under-utilization of the SIMD resources due to branch divergence. They proposed dynamic warp formation (DWF) in which diverged threads that happen to arrive at the same PC, *even though they belong to different warps*, can be grouped and run together as a wider SIMD group. However, if a subset of threads in a

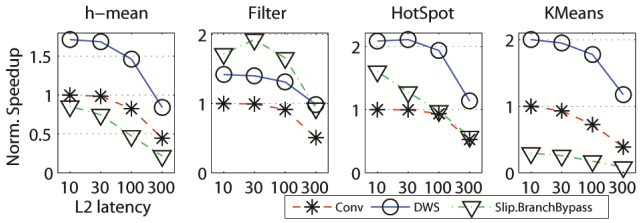


Figure 15: Speedup vs. L2 lookup latency. DWS refers to *DWS.ReviveSplit* in Figure 13. The speedup of DWS compared to *Conv* increases with longer L2 latency. Systems are configured according to Table 3. Performance is normalized to each benchmark's execution time under *Conv* with an L2 lookup latency of 10 cycles. The harmonic mean for the normalized speedup of all benchmarks is shown as `h-mean`.

warp miss the cache and stall, there is still no way to allow the threads that hit to continue: this behavior is not compatible with the active mask on top of the re-convergence stack. As Table 1 has shown, divergent memory accesses can be more common than divergent branches in some applications. Furthermore, dynamic warp formation only occurs when multiple warps arrive at the same PC; if such condition is not met, the conventional re-convergence stack still prevents threads in the same warp from hiding each other's latency upon branch divergence. Our approach instead creates separate scheduling entities so that each warp-split can execute independently. This means that even after a divergent branch, threads on both paths are able to make progress and hide each others' latency. By creating new scheduling entities after each branch or miss divergence, our approach fully integrates branch and miss handling.

Similar to dynamic warp formation, stream processors such as Imagine can split streams conditionally according to branch outcomes [12], and each resulting stream follows the same control flow. Although conditional streams reduce unnecessarily predicated computation and communication, they do not address latency hiding. In fact, latency hiding is not necessary for stream processors; data is always available through the stream register file. For this reason, memory divergence never occurs as well.

Our technique is complementary to Vector Lane Threading (VLT) [24] and the Vector-Thread (VT) Architecture [14]. VLT assigns groups of lanes to different user-level threads; lanes belonging to the same user-level thread execute in SIMD, but they do not need to execute in lockstep with lanes in other groups. However, lanes belonging to the same SIMD group may still stall unnecessarily due to branch and memory divergence. VT is a MIMD implementation of vector semantics, where each lane has the autonomy and overhead of its own fetch/decode/scoreboard logic. It is therefore not a strict SIMD organization and not subject to the divergence phenomena addressed in this paper. Nevertheless, the principles of DWS may also apply to VT when it executes data-parallel applications.

Adaptive slip [30] addresses memory divergence to improve latency hiding for SIMD organizations. It employs a mechanism that is similar to a re-convergence stack, by simply using the cache access outcome instead of branch outcome so that threads that hit can continue. Similar to using the re-convergence stack, threads exhibiting divergent behavior cannot interleave their execution to hide latency, which limits benefits. It also assumes aggressive branch predication to avoid forced re-convergence at conditional branches within a loop body. Section 5.7 compares adaptive slip with DWS in more detail.

Existing techniques that address long latency memory accesses in the context of simultaneous multi-threading (SMT) do not help in the case of SIMD because SMT threads share

a single datapath. With SMT, the problem is resource contention and reduced instruction level parallelism (ILP), since a stalled thread occupies expensive issue queues, rename-registers, and reorder-buffer entries, which limits ILP discovery for the other thread. Techniques for SMT resource distribution [5, 31] do not apply to in-order SIMD organizations, since SIMD threads operating on different lanes do not compete for pipeline resources. Instead, the main problem raised by long latency memory accesses is the risk of stall cycles due to inadequate latency hiding. Pre-computation using speculative threads [6] or runahead threads [23] improves MLP. However, these speculative approaches require run-time dependency analysis among instructions as well as the ability for out-of-order execution and commit. These requirements are usually not met with simple, in-order SIMD hardware. We provide a solution designed specifically for SIMD hardware that allows it to exploit more MLP without speculative execution.

We have not considered the effect of prefetching or non-blocking loads. Nevertheless, DWS is complementary to both. It helps even in cases where the access pattern is not easily prefetched. Moreover, prefetching may increase contention for cache and bandwidth, especially when contention among many threads is severe. On the other hand, non-blocking loads employed by Tesla [8] and Fermi [1] are still in-order issue so they improve MLP but provide minimal latency hiding benefit. In both cases, memory divergence still occurs if threads in a warp are not all runnable at the same time.

Dynamic warp subdivision is also complementary to MLP-aware cache replacement [22]. While DWS creates additional scheduling entities to hide latency and issue more over-lapping requests, MLP-aware cache replacement smartly prioritizes these outgoing requests to reduce their aggregate latency.

8. CONCLUSIONS & FUTURE WORK

This paper proposes dynamic warp subdivision to create warp-splits in the case that the SIMD processor does not have sufficient warps to hide latency and leverage MLP. Warp-splits are simply extra schedulable entities that do not require extra register file state. They can be generated upon branch divergence or memory divergence. The conventional re-convergence stack is amended with a warp-split table to relax the serialization of different branch paths and it allows threads in the same warp to interleave their execution in an asynchronous manner.

We evaluate the technique using a general purpose multicore processor with four 16-wide WPUs, each with four warps operating over a coherent cache hierarchy. Using a two level cache hierarchy, DWS generates an average speedup of 1.7X. We further study the sensitivity of DWS to various architectural parameters and demonstrate that the performance of DWS is robust and it does not degrade performance on any of the benchmarks tested.

One opportunity for future work is to integrate DWS with dynamic warp formation [10]. It may further improve performance by speculating cache miss frequency and miss latencies in order to better decide when to subdivide warps. Finally, it would be interesting to study applications quantitatively to identify those that would benefit from particular SIMD architectures according to their degree of divergence.

9. ACKNOWLEDGEMENTS

This work was supported in part by SRC grant no. 1607, NSF grant nos. IIS-0612049 and CNS-0615277, a grant from Intel Research, a professor partnership award from NVIDIA Research, and an NVIDIA Ph.D. fellowship (Meng). We would like to thank Michael Boyer, Mario D. Marino, and Gregory Faust for their suggestions. We would also like to thank the anonymous reviewers for their helpful comments.

10. REFERENCES

[1] NVIDIA's next generation CUDA compute architecture: Fermi. *NVIDIA Corporation*, 2009.
[2] ATI. Radeon 9700 Pro. http://mirror.ati.com/products/pc/radeon9700pro, 2002.
[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4), 2006.
[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors using CUDA. *JPDC*, 2008.
[5] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *ISCA*, 2006.
[6] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *MICRO 34*, 2001.
[7] Intel Corporation. Intel AVX: New frontiers in performance improvements and energy efficiency, 2009.
[8] NVIDIA Corporation. GeForce GTX 280 specifications. 2008.
[9] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hern, T. Juan, G. Lowney, M Mattina, and A. Seznec. Tarantula: A vector extension to the Alpha architecture. In *ISCA*, 2002.
[10] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO*, 2007.
[11] M. Gschwind. Chip multiprocessing and the Cell Broadband Engine. In *CF*, 2006.
[12] U. J. Kapasi, J. Dally, W, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In *MICRO 33*, 2000.
[13] C. Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Technical report, University of California, Berkeley, 1999.
[14] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread architecture. In *ISCA*, 2004.
[15] R. A. Lorie and H. R. Strong. Method for conditional branch execution in SIMD vector processors. *US Patent 4,435,758*, 1984.
[16] J. Meng, J. W. Sheaffer, and K. Skadron. Exploiting inter-thread temporal locality for chip multithreading. In *IPDPS*, 2010.
[17] J. Meng and K. Skadron. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In *ICCD*, 2007.
[18] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance: Extended results. *U.Va. Tech. Report CS-2010-5*, 2010.
[19] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. *IISWC*, 2006.
[20] Y. Nishikawa, M. Koibuchi, M. Yoshimi, K. Miura, and H. Amano. Performance improvement methodology for ClearSpeed's CSX600. In *ICPP*, 2007.
[21] S. E. Orcutt. Implementation of permutation functions in illiac iv-type computers. *IEEE Trans. Comput.*, 25(9), 1976.
[22] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA*, 2006.
[23] T. Ramirez, A. Pajuelo, O.J. Santana, and M. Valero. Runahead threads to improve SMT performance. *HPCA*, 2008.
[24] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector lane threading. In *ICPP*, 2006.
[25] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *MICRO 31*, 1998.
[26] R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1), 1978.
[27] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3), 2008.
[28] Y. Takahashi. A mechanism for SIMD execution of SPMD programs. In *HPC-ASIA*, 1997.
[29] D. Talla and L. K. John. Cost-effective hardware acceleration of multimedia applications. In *ICCD*, 2001.
[30] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for SIMD cores. In *SC*, 2009.
[31] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO 34*, 2001.
[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *ISCA*, 1995.