Robust SIMD: Dynamically Adapted SIMD Width and Multi-Threading Depth

Jiayuan Meng
Leadership Computing Facility Division
Argonne National Laboratory
Argonne, Illinois
jmeng@alcf.anl.gov

Jeremy W. Sheaffer
Department of Computer Science
University of Virginia
Charlottesville, Virginia
jws9c@cs.virginia.edu

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

***Abstract*—Architectures that aggressively exploit SIMD often have many datapaths execute in lockstep and use multi-threading to hide latency. They can yield high throughput in terms of area- and energy-efficiency for many data-parallel applications. To balance productivity and performance, many recent SIMD organizations incorporate implicit cache hierarchies. Exaples of such architectures include Intel's MIC, AMD's Fusion, and NVIDIA's Fermi. However, unlike software-managed streaming memories used in conventional graphics processors (GPUs), hardware-managed caches are more disruptive to SIMD execution; therefore the interaction between implicit caching and aggressive SIMD execution may no longer follow the conventional wisdom gained from streaming memories. We show that due to more frequent memory latency divergence, lower latency in non-L1 data accesses, and relatively unpredictable L1 contention, cache hierarchies favor different SIMD widths and multi-threading depths than streaming memories. In fact, because the above effects are subject to runtime dynamics, a fixed combination of SIMD width and multi-threading depth no longer works ubiquitously across diverse applications or when cache capacities are reduced due to pollution or power saving.**

To address the above issues and reduce design risks, this paper proposes *Robust SIMD*, which provides wide SIMD and then dynamically adjusts SIMD width and multi-threading depth according to performance feedback. Robust SIMD can trade wider SIMD for deeper multi-threading by splitting a wider SIMD group into multiple narrower SIMD groups. Compared to the performance generated by running every benchmark on its individually preferred SIMD organization, the *same* Robust SIMD organization performs similarly—sometimes even better due to phase adaptation—and outperforms the best fixed SIMD organization by 17%. When D-cache capacity is reduced due to runtime disruptiveness, Robust SIMD offers graceful performance degradation; with 25% polluted cache lines in a 32 KB D-cache, Robust SIMD performs 1.4× better compared to a conventional SIMD architecture.

I. Introduction

By using a single instruction sequencer to control multiple datapaths, *Single instruction, multiple data* (SIMD) organizations save both area and power. Several throughput-oriented processors have been aggressively exploiting SIMD. Examples include graphics processors (GPUs) [11] and the Cell Broadband Engine (CBE) [15]. Such architectures have been increasingly used for general purpose, data parallel applications, including scientific computation, media processing, signal analysis, and data mining [8], [37]. Recently, several SIMD organizations have employed hardware-managed cache hierarchies in order to offer both throughput and productivity. Specifically, *gather loads* (*i.e.* load a vector from a vector of arbitrary addresses) or *scatter stores* (*i.e.* store a vector to a vector of arbitrary addresses) are supported in hardware and do not require explicit programmer effort. Such architectures include Intel's Many Integrated Core [32], [10] and NVIDIA's Fermi [2].[1] This paper studies SIMD organizations with cache hierarchies, as illustrated in Figure 1.

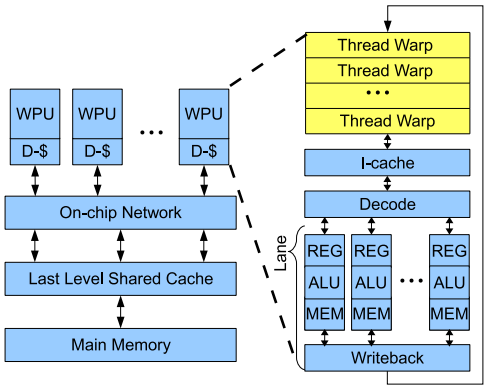


Figure 1: The baseline SIMD architecture groups scalar threads into warps and executes them using the same instruction sequencer.

In this paper, the set of hardware units under SIMD control is referred to as a *warp processing unit* or WPU [24]. The banked register files and execute units are divided into *lanes*. Each hardware thread context has its register file residing in one of the lanes. The number of threads that operate in SIMD is referred to as *SIMD width*, or *width* for short in this paper. Instead of using out-of-order pipelines, SIMD processors are often in-order so that they can accommodate more threads and improve throughput. They hide latency by having multiple groups of SIMD threads, or *warps*, and time-multiplexing warps to overlap memory accesses of one warp with computation of another. The number of warps is referred to as *multi-threading depth* or *depth*. SIMD is not limited to vectors; it can also come in the form of arrays of scalar datapaths (*i.e.* single instruction, multiple threads (SIMT) organizations) in which divergent branches, scattered loads and stores are handled implicitly by the underlying hardware.

It may seem that SIMD organizations with cache hierarchies would demonstrate similar traits to those that use streaming memories (*e.g.* NVIDIA's Tesla [2], [11]): moderate SIMD width, deep multi-threading, and a single

[1]Intel's SSE2 instruction sets operate over cache hierarchies but require explicit programmer effort to perform gather and scatter operations and to align the vectors.

configuration that works consistently well across a variety of applications. However, we show that these ideas do not apply when cache hierarchies are used instead. The reasons are threefold.

- **Memory latency divergence** (*i.e.* when a warp executes a memory instruction, those threads that finish early have to wait for those that finish late) occurs more frequently and affects performance more significantly in cache hierarchies than in streaming memories, making wide SIMD a risky design decision. When threads are stalled due to memory latency divergence, the full SIMD width is no longer utilized.
- **Non-L1 data accesses** are faster for cache hierarchies than for streaming memories, therefore fewer warps are needed to hide latency in cache hierarchies.
- **Cache contention** is difficult to predict, and the intensity of such contention limits the total number of active threads. When cache thrashes, adding more warps would only impede performance rather than hide latency.

Even worse, the effects of the above factors vary across applications; therefore, different applications prefer different SIMD widths and multi-threading depths, as we show in Section III. As a result, there is a high risk in using a fixed SIMD organization to run various applications on cache hierarchies. In addition, application-specific static designs are not sufficient; as shown by the last few years of general-purpose GPU computing work, there is a large variety of general-purpose applications that can benefit from SIMD computation.

Furthermore, the effects of the above factors also depend on the available cache capacity. Runtime systems may pollute the cache, and faulty cache lines can be turned off as well. In some circumstances, cache lines can be powered down to save power consumption, or replicated for the purpose of error-resilience [13], [20], [36], [38]. Moreover, as NVIDIA's Fermi [2] demonstrates, multiple kernels may execute simultaneously over the *same* L1 storage (*i.e.*, thread blocks from different kernels can execute on the same streaming multiprocessor). While such mechanisms may improve memory efficiency, increase occupancy of hardware thread-contexts, and enable software pipelining, it also introduces L1 contention among different kernels. As we show in Section VII-B, the runtime variation of the cache capacity available to a kernel not only undermines performance for an individual workload, but also changes the preferred SIMD width and depth. Unfortunately, conventional SIMD architectures are not able to adapt to such runtime dynamics.

To address the above issues, this paper proposes *Robust SIMD*. Robust SIMD learns the preferred SIMD width and depth at runtime. Robust SIMD provides wide SIMD in the first place and offers the flexibility to re-configure itself to narrower or deeper SIMD. Consequently, the same SIMD organization can adapt to diverse applications or cache capacities and yield robust performance. Robust SIMD can trade wide SIMD for deep multi-threading. Several gradient-based adaptation strategies are investigated. Experiments show that with Robust SIMD, the same architecture can achieve near-optimal performance for diverse applications that prefer different SIMD widths and depths. Robust SIMD outperforms the best fixed SIMD organization by 17%. When the available D-cache capacity is reduced due to runtime disruptiveness, Robust SIMD offers graceful performance degradation and performs 1.3× better compared to dynamic warp subdivision (DWS) [24] in terms of execution time. Robust SIMD can also be integrated with DWS to further improve performance. The area overhead is less than 1%.

II. Related Work

Several techniques have been proposed to adjust a vector organization according to traits in the workloads. Both vector lane threading [31] and the vector-thread (VT) architecture [19] are able to change SIMD width (but not multi-threading depth). Nevertheless, both techniques use the vector length or the available data parallelism provided by the program as the desirable SIMD width. Such static approaches fail to adapt to runtime dynamics brought by coherent caches.

Liquid SIMD [9] and Vapor SIMD [26] advocate static and just-in-time compilation of SIMD programs for the purpose of forward migration to newer hardware generations. Using these compilation techniques, the same program can be executed with various SIMD widths, which is necessary for Robust SIMD; however, Liquid SIMD and Vapor SIMD do not dynamically adapt SIMD width or multi-threading depth in the hardware.

Fung *et al.* [14] addressed under-utilization of SIMD resources due to branch divergence. They proposed dynamic warp formation (DWF), in which divergent threads that happen to arrive at the same PC, even though they belong to different warps, can be grouped and run as a wider warp; however, DWF does not adapt overall SIMD width or multi-threading depth. For applications with no branch divergence, DWF would execute in the conventional way, despite that adjusting SIMD width and multi-threading depth may improve latency hiding or reduce cache contention.

Memory latency divergence for SIMD architectures are addressed by adaptive slipping [35] and dynamic warp subdivision (DWS) [24]. These techniques allow threads that hit the cache to run ahead while those that miss are suspended. The key difference between Robust SIMD and these techniques is that Robust SIMD has a feedback learning loop that takes the actual performance into consideration when searching for the best width and depth. In contrast, previous techniques apply heuristics, such as splitting warps according to cache hits or misses, which may risk performance degradation due to over-subdivision [24]. There are also a few other issues that are not addressed by previous techniques. First, all threads are actively executed until they terminate, and there is no way to turn off some lanes or deactivate a few warps to reduce cache contention. Second, if divergence is rare and cache misses usually occur for all threads in the same warp, previous techniques would be unable to adjust SIMD width or depth. All of the above factors make previous techniques susceptible to runtime dynamics. Section VII-C shows that when cache capacity is reduced, the SIMD organization that DWS originally chose would suffer drastic performance degradation, while

the same Robust SIMD organization would still generate near-optimal performance.

There also exist many scheduling techniques to reduce cache contention for non-SIMD chip multi-processors (CMPs). These techniques typically select a few threads from a pool of threads whose joint working set minimizes cache contention [27], [33], [34]. These techniques exploit the heterogeneity in threads' working sets and they do not consider data-parallel, homogeneous tasks. Moreover, they aim to find *which* threads to execute concurrently, without considering *how* threads are mapped to hardware thread contexts. SIMD organizations raise the issue of how to organize concurrent thread contexts into warps, requiring specifications of both SIMD width and multi-threading depth, which are not addressed by previous techniques.

There are a wide range of adaptive techniques, from cache replacement policies [29] to thread scheduling [33], that follow the procedure of sampling and adjusting according to performance feedback. Compared to these techniques, the novelty of our work lies in identifying the need for adaptation of SIMD configurations, addressing the challenges in dynamically modifying SIMD width and depth, and proposing convergence mechanisms to ensure adaptation robustness.

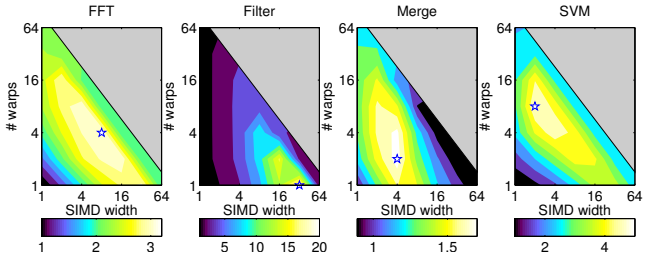
III. Motivation

Coherent caches introduce several more runtime dynamics than streaming memories, which affect the desired SIMD width and multi-threading depth in several ways. Using the simulation infrastructure described in Section V, we demonstrate and discuss these effects individually. Note that in our experiments, less than 1% of the messages among caches are coherence messages (*i.e.*, invalidations and fetches); therefore the phenomenon presented in this study is not caused by coherence and incoherent cache hierarchies can be subject to the same issue. Due to space limitations we only show four benchmarks with different data access patterns.

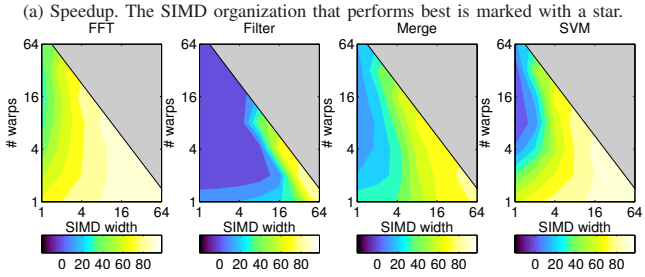
A. Memory Latency Divergence

Memory latency divergence occurs more frequently and affects performance more significantly in cache hierarchies than in streaming memories. With streaming memories, threads in the same warp access the same memory unit upon a SIMD memory instruction. Data is guaranteed to reside in the memory unit, and these threads experience similar, if not identical, memory latencies. However, with cache hierarchies, it is possible that some threads would hit in L1 while others miss. In such cases, memory latencies may differ by orders of magnitude. As a result of memory latency divergence, the full SIMD width is often not utilized.

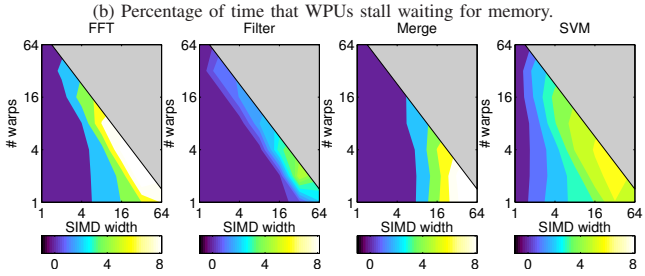
As Figure 2c shows, memory latency divergence is more likely to occur with wider SIMD, making wide SIMD a risky design decision. Wider SIMD is also more likely to suffer from branch divergence; we do not discuss branch divergence in detail because it occurs far less frequently than memory latency divergence. In our experiments corresponding to Figure 2, using WPUs with four 16-wide warps, our benchmarks show that memory latency divergences occur every 13 instructions on average, 18 times more frequently than branch divergences.



(a) Speedup. The SIMD organization that performs best is marked with a star.



(b) Percentage of time that WPUs stall waiting for memory.



(c) Avg. number of memory latency divergences per 100 instructions.

Figure 2: Analysis and characterization of SIMD organizations with cache hierarchies. Speedup is normalized to a system with four single-threaded, in-order cores. With 32 KB, 8-way associative D-caches whose cache line size is 128 B, we test various widths and depths at powers of two. Details are described in Section V. We chose four benchmarks that demonstrate diverse behavior.

B. Lower Latency in Non-L1 Data Accesses

Because streaming memories often bulk-load data with long latency memory accesses, they require deep multi-threading to hide latency. The long memory latency is caused by the fact that streaming memory has only one level of storage (*i.e.* the L1 scratchpad) above the device memory. Multiple on-chip memory hierarchies are viable but they would increase the complexity with explicit data management. When data is not present in the L1 scratchpad, all threads wait for hundreds of cycles to fetch from the device memory, which warrants deep multi-threading. However, cache hierarchies usually have two or three levels on-chip, and most L1 misses (more than 96% in our experiments) can be captured by the L2 cache. The overall L1 miss latency can be only tens of cycles. Therefore, SIMD with cache hierarchies may not need so many warps as with streaming memories to hide latency.

C. Unpredictable Cache Contention Intensity

For caches that are not fully associative, the occurrence of cache contention depends on application-specific data access patterns in addition to the amount of data in use. We are not aware of any technique that can estimate cache contention statically and the appropriate number of active threads accordingly.

Figure 2b demonstrates that cache contention limits the total number of active threads. With 32 threads or more per WPU, the SIMD pipeline stalls waiting for memory more than 40% of the time. Moreover, adding more warps can only hide latency effectively when the thread count per core is small; with wide SIMD, more warps exacerbate cache contention and increase pipeline stalls.

D. The Lack of a Single Best SIMD Width and Depth

The frequency of memory latency divergence, length of L1 miss latency, and intensity of cache contention all affect the desirable SIMD width and depth; yet, they all depend on runtime dynamics. As a result, different applications, or even different execution phases, prefer different SIMD widths and depths. As Figure 2a shows, some benchmarks prefer few wide warps and some prefer a combination of modest width and depth. Moreover, performance is sensitive to SIMD configuration; the optimal configuration for one application may work poorly with another.

Even worse, when cache capacity is reduced dynamically to save power, the preferred width and depth will change as well, as illustrated in Figure 2a, such performance effects also occur when the cache is shared by multiple applications [30] or some segment of the cache is reserved for other purposes (*e.g.* error resilience [36] and on-chip communication [12]).

All the above studies lead to our conclusion that designing a SIMD architecture over cache hierarchies requires the ability to adjust the SIMD configuration and adapt to various runtime dynamics. We therefore propose Robust SIMD, which dynamically learns the desired width and depth at runtime and yields robust performance. The alternative is to select SIMD parameters for each application based on offline profiling; however, this static approach incurs additional profiling overhead, and does not take into account phase changes, different input data, and runtime changes in the cache capacity.

IV. Hardware Support for Runtime SIMD Reconfiguration

Figure 3 illustrates the process of a WPU adapting to a data-parallel code section. Adaptation is triggered by the runtime *every time* a SIMD kernel is launched. There are two phases to execute data-parallel tasks within a parallel `for` loop. First, in the *adaptation phase*, a WPU executes a few tasks during one sampling interval, measures the performance within this period, and feeds this measurement back to an *Adaptation State Unit (ASU)* which evaluates the previous configuration and suggests a new one. The WPU then reconfigures itself according to the suggested SIMD width and multi-threading depth and repeats this process until the ASU converges on a preferred configuration. Second, in the *execution phase*, the WPU maintains the converged configuration until all tasks in the current parallel section are completed. Our study shows that data-parallel tasks exhibit homogeneous behavior so that the SIMD configuration preferred by the sampled tasks will also be preferred later by other tasks belonging to the same parallel `for` loop. If programmers are familiar with the application and would like to manually set the desired SIMD

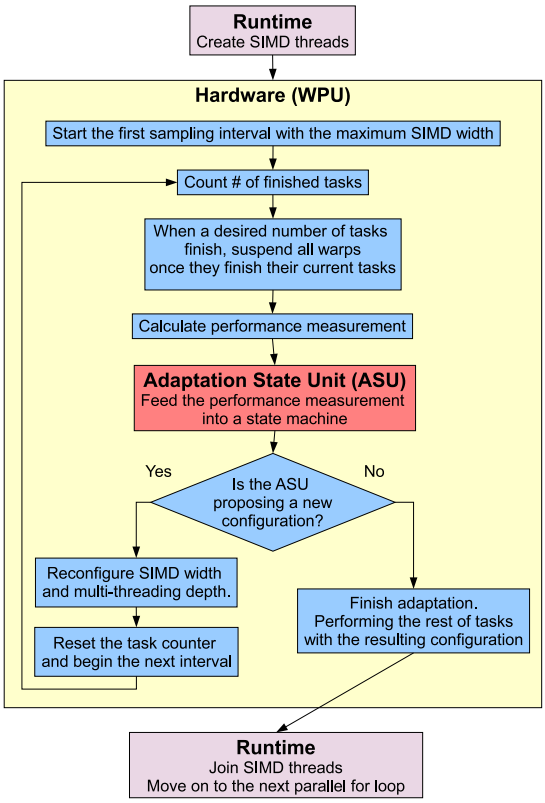


Figure 3: The process of adapting a WPU's SIMD width and multi-threading depth. The process is repeated for *every* data-parallel section.

width and depth, an API can be provided to do so and skip the adaptation phase. The rest of this section focuses on the hardware modifications required for Robust SIMD. Section VI compares several adaptation strategies that can be employed by the ASU.

The SIMD width and multi-threading depth are only adjusted at the end of a sampling interval. The number of lanes in each WPU sets the upper bound of its SIMD width. Within this limit, an adaptive WPU supports the following operations:

- Lanes can be turned off and on to adjust SIMD width.
- Warps can be suspended or resumed to adjust multi-threading depth.
- SIMD width can be traded for multi-threading depth by breaking a wide warp into multiple narrower *warp-slices* that can interleave their execution to hide each other's latency. These warp-slices can be reunited in future intervals as a wider warp. An original warp with full SIMD width can be regarded as a warp-slice as well.

Note that SIMD can operate multiple datapaths in the form of a vector or in the form of an array with a set of scalar datapaths. The latter is referred to as *single instruction, multiple threads* (SIMT). In this paper, we demonstrate with a SIMT organization where branch divergence is handled by a *re-convergence stack* [14]: when threads in the same warp branch to different control paths, the WPU first executes threads falling onto one path and suspends others; a bit

mask representing those active threads is pushed onto the re-convergence stack and popped when the corresponding threads finish their current branch path; the WPU can then switch to execute threads falling into the alternate path. The re-convergence stack is popped upon *post-dominators* that signal control flow re-convergence at the end of each branch path. Our technique also applies to vector organizations. In fact, as we discuss in Section IV-A, the implementation would be simpler in vector organizations which have no re-convergence stacks.

A. Reconfiguring SIMD Width and Multi-threading Depth

Adjusting SIMD width and depth is effectively the same as partitioning available warps into subsets of SIMD threads called *warp-slices*. Warp-slices are treated as independent scheduling entities similar to warps. When executing a warp-slice, the WPU proceeds as if it executes an entire warp, except that threads not belonging to the same warp-slice have their corresponding lanes clock-gated according to an *active mask*. The active mask marks the warp-slice's associated lanes so that irrelevant lanes are not accessed at all. In one cycle, only one warp or warp-slice can execute. The process of dividing a warp is illustrated in Figure 4.

The warp-slice table is used to keep track of warp-slices. Each warp-slice occupies an entry in the WST that records the active masks that identify associated threads. Creating a warp-slice also requires duplicating its re-convergence stack for SIMT or array organizations (for vector organizations where branches are predicated, the WST alone would suffice). The overhead of such copying is at most $\log_2(W)$ cycles, where W is the maximum SIMD width available. We set this reconfiguration overhead to 10 cycles in our experiments; such overhead is negligible compared to the time spent in the execution phase, therefore it hardly affects the overall execution time. Physically, the duplicated re-convergence stacks can be combined, as shown in Figure 4(c).

At the end of each sampling interval and before the ASU suggests a different SIMD configuration for the next interval, all threads finish their current tasks and arrive at the same PC with empty re-convergence stacks. Warp-slices can then merge into complete warps by taking the logical "or" of their active masks in WST entries. Warps are then divided again; neighboring lanes form warp-slices with the desired SIMD width. The number of active warp-slices are determined by the desired multi-threading depth; the remaining, inactive warp-slices are suspended, computing no tasks in the subsequent interval, and are referred to as *dormant warp-slices*. These dormant warp-slices will be resumed once others finish or hit a synchronization point. During this reconfiguration step, the WPU's pipeline is stalled; however, this only occurs during the adaptation phase, whose overhead is negligible for a reasonable workload. Note that a wide warp can be divided into multiple active warp-slices to trade SIMD width for multi-threading depth.

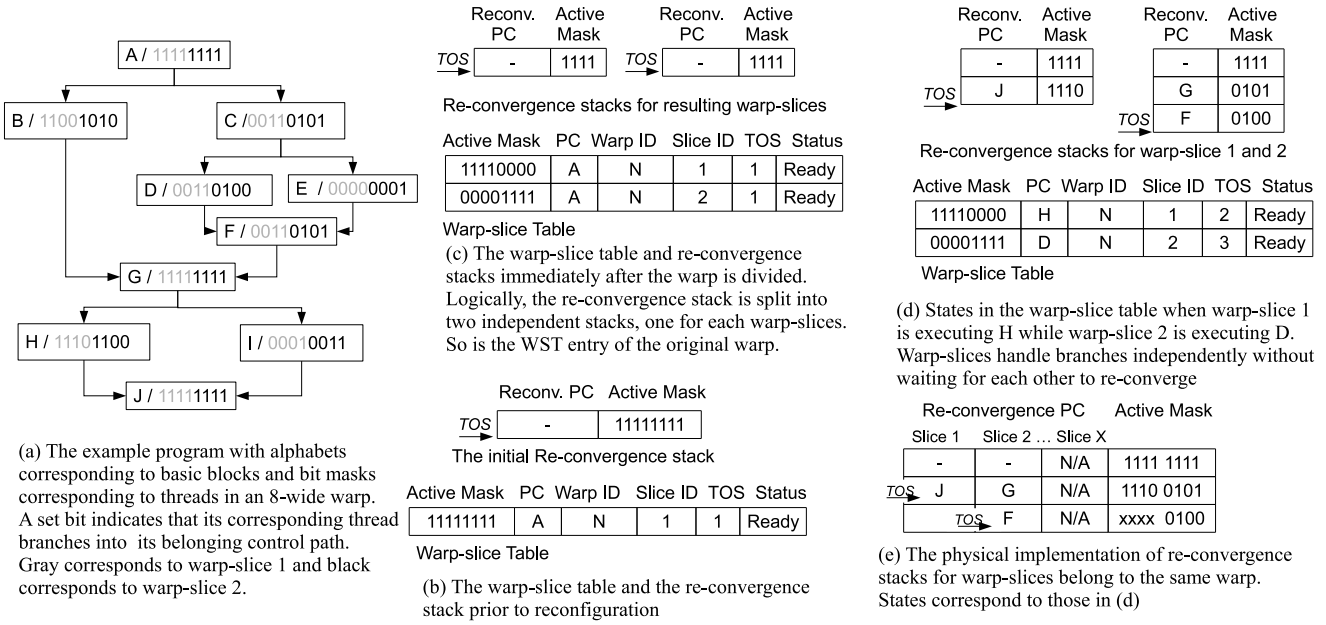
We impose the constraint that all warp-slices on a WPU share the same width, so that a SIMD configuration can be simply parameterized as a tuple, denoted by $(width \times depth)$, which in turn simplifies adaptation. Otherwise, the number of possible SIMD configurations will explode. Only when the available SIMD width is not divisible by the suggested SIMD width would those residue threads form warp-slices that are narrower than others. Adapting to warp-slices with different widths is interesting future work.

It may seem that by sometimes reducing SIMD width or suspending warps, we are underutilizing hardware to begin with. In fact, doing this actually increases caching efficiency, thereby reducing pipeline stalls upon memory accesses and improving the overall utilization and throughput. Section VII shows that to maximize throughput across different applications, it is better to provide the option of wider SIMD and allow it to become narrower and deeper, than stick to a narrower but fully utilized static configuration.

Because a warp-slice is never constructed with threads belonging to different warps, Robust SIMD preserves the register banks. Decreasing or increasing SIMD width is basically suspending or resuming the corresponding SIMD threads, respectively. The suspended threads will be eventually resumed when other threads exit or reach a synchronization barrier. As a result, there is no need to adjust and migrate data in register files. In other words, Robust SIMD need not modify conventional datapaths or lanes—we only change how datapaths are logically grouped, which is managed by a warp-slice table (WST).

Splitting warps does not affect explicit synchronization. We assume that any synchronization is explicit. The problem with making a warp implicitly synchronous is that it locks in the warp size as a legacy obligation. If a programming model provides explicit synchronization primitives (e.g. `__syncthreads()` in CUDA), threads usually synchronize on the basis of a conceptual thread-block rather than an individual warp in order to preserve flexibility. Upon such a synchronization instruction, a warp or warp-slice can stall waiting for other warps in a conventional way. For vector architectures that rely on SIMD width for implicit synchronization, we suggest to implicity insert an instruction to synchronize warp-slices belonging to the same warp whenever the compiler detects that an element in a vector is comsuming another one in an unaligned vector position.

In contrasts to *warp-splits* in DWS, which are created upon branch or memory latency divergences, Robust SIMD constructs warp-slices regardless of threads' divergence histories, so that SIMD width and depth can be adjusted even when there is no memory latency divergence. In Robust SIMD, to form a warp-slice, a WPU only has to define how many threads should be grouped into the new warp-slice, without specifying *which* threads to be grouped. We heuristically group threads operating on neighboring lanes into a warp-slice. Moreover, forming and merging of warp-slices only occur at the end of each sampling interval, rather than upon individual cache hits or misses. Finally, each warp-slice has its independent re-convergence stack, allowing them to hide each other's latency in spite of the occurrences of conditional branches. In contrast, DWS warp-splits belonging to the same warp share the same re-convergence stack, and may be subject to frequent synchronization upon conditional branches, limiting the effectiveness in latency hiding. Compared to the hardware of DWS, Robust SIMD reuses the same structure of the warp-split table in DWS for the warp-slice table; in addition, it tracks the TOS for each



(a) The example program with alphabets corresponding to basic blocks and bit masks corresponding to threads in an 8-wide warp. A set bit indicates that its corresponding thread branches into its belonging control path. Gray corresponds to warp-slice 1 and black corresponds to warp-slice 2.

(c) The warp-slice table and re-convergence stacks immediately after the warp is divided. Logically, the re-convergence stack is split into two independent stacks, one for each warp-slices. So is the WST entry of the original warp.

(b) The warp-slice table and the re-convergence stack prior to reconfiguration

(d) States in the warp-slice table when warp-slice 1 is executing H while warp-slice 2 is executing D. Warp-slices handle branches independently without waiting for each other to re-converge

(e) The physical implementation of re-convergence stacks for warp-slices belong to the same warp. States correspond to those in (d)

Figure 4: An example that illustrates how a warp is split into two warp-slices using the warp-slice table (WST). Re-convergence stacks only exist in array or SIMT organizations for handling divergent branches, therefore their modification is not necessary for vector organizations. Note that in (e), the two TOSs, which correspond to two warp-slices, operate independently on different parts of the active mask; therefore there is no interference and only one TOS is accessed at a given time.

warp-slice and it introduces the ASU for adaptation.

B. Hardware Overhead

Without modifying conventional datapaths, we manage the logical grouping of datapaths using the warp-slice table (WST). Other hardware units introduced to an adaptive WPU mainly include the adaptation state unit (ASU) and counters for completed tasks within one sampling interval.

The ASU, if implemented in hardware, includes a comparator to evaluate performance measurements, an adder to offset the width and depth, a multiplier to scale the offset strides, and several multiplexors to choose the appropriate strides and directions. It also includes several registers to store current SIMD width and depth, the recorded optimal width and depth, the recorded optimal performance measure, the adaptation strides, and the current state of the ASU. Given that most of these values are below 256 and can be stored within a single byte, we estimate that the ASU requires 20 bytes of storage.

Assuming the WPU has two warps with a SIMD width of 32 and it supports up to 8 warp-slices, each entry in the WST would require 14 bytes: 32 bits for the active mask, 1 bit for the warp ID, 64 bits for the PC, 3 bits for the warp-slice ID, 2 bits for the warp status, and 8 bits for storing the top-of-stack (TOS) if a re-convergence stack is used. A maximum of 8 entries are needed, resulting in a total of 112 bytes. For SIMT or array organizations, the duplicated re-convergence stack would add an additional column of PCs for each warp-slice. Assuming the stack can grow as high as 8 layers and each warp can be divided into at most eight warp-slices, this results in additional 1024 bytes. Adding the storage requirements for the WST and the ASU, this yields a total of 1164 bytes, or approximately 1 KB of storage.

To estimate area overhead, we measure realistic sizes for 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. The area overhead of an additional lane takes into account an additional set of functional units, register files, and its intra-datapath interconnection. 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. Assuming the WPU described above has a 32 KB D-cache and a 16 KB I-cache, the additional storage requires less than 1% of a WPU's area.

There are several other hardware overheads but their area needs are negligible. First, dividing a warp into multiple warp-slices requires more scheduling entries. Since we use round-robin scheduling policy in all cases, such a simple scheduler's overhead is negligible for a few warps (no more than 16 in a balanced configuration according to our experiments). Second, we make the same assumption about bypassing in all cases, and it is unaffected by SIMD width and depth at all times. The area savings of eliminating bypassing on such a simple datapath are small enough not to justify the area overhead of deeper multi-threading.

V. Methodology

We model our system with the MV5 simulator [23], the only publicly available simulator that support general purpose SIMT architectures with coherent caches. MV5 is a cycle-accurate, event-driven simulator based on gem5 [6]. Because existing operating systems do not directly manage

SIMD threads, applications are simulated in system emulation mode with a set of primitives to create threads in a SIMD manner.

A. Simulation Infrastructure

The modeled WPUs handle divergent branches with reconvergence stacks. Due to the lack of compiler support, we manually instrument application code with post-dominators. Each lane is modeled with an IPC of one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory controller reordering effects). A WPU switches warps upon every cache access with no extra latency—as modern GPUs do—simply by indexing appropriately into shared scheduling and register file resources. Using our simulator, we study chip multi-processors (CMPs) with four homogeneous WPUs operating over coherent caches. Since the divergence is local to a WPU, experiments with a different WPU count produces similar results.

The memory system is a two level coherent cache hierarchy with private L1 caches and a shared L2. Each WPU has a private I-cache and D-cache. I-caches are not banked, because only one instruction is fetched every cycle for all lanes. D-caches are always banked according to the number of lanes. We assume there is a perfect crossbar connecting the lanes with the D-cache banks. If bank conflicts occur, memory requests are serialized and a small queuing overhead (one cycle) is charged. The queuing overhead can be smaller than the hit latency because requests can be pipelined. All caches are physically indexed and physically tagged with LRU replacement policy. Caches are nonblocking with miss status holding registers (MSHRs). Coherence is handled by a directory-based MESI protocol. L1 caches schedule outgoing L2 requests on a first in, first out (FIFO) basis. The L1 caches connect to L2 banks through a crossbar, which has a limited bandwidth of 57 GB/s. 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 300 cycles[2], and the memory controller is able to pipeline the requests.

Table I summarizes the main architectural parameters. Note that the aggregate L2 access latency is broken down into L1 lookup latency, crossbar latency, and the L2 lookup latency. The L2 lookup latency contains both tag and data lookup.

B. Data-Parallel Programming Model

While there are commercial software systems for data-parallel applications (*e.g.* TBB [17], OpenMP [7], Cg [21], CUDA [1], OpenCL [18], and Pthreads [3]), they either lack the capability to create SIMD threads or do not execute properly in MV5's system emulation mode. Therefore, we adopted MV5's OpenMP-style programming model and runtime threading library [22]. Each invocation of the innermost parallel `for` loop has a unique loop index and would execute on a hardware thread context as an individual task. Such an API is mainly designed to mimic the programming interface in existing parallel APIs in MV5 simulations. Applications are cross-compiled to the Alpha ISA using G++ 4.1.0. insert bytecodes in computation kernels that signal SIMD re-convergence to mimic the behavior of a SIMD compiler; this step is transparent to application developers and is not needed once full compiler support is available. We optimize the code with `O1` because higher levels of optimization sometimes misplace the inserted bytecode.

There is no need for SIMD-specific optimizations such as vector alignment; scalar instructions can be implicitly grouped in hardware to operate in lockstep, in a way similar to Cg [21]. Because implicitly managed caches are used to ensure productivity, we assume programmers would not (and they need not) explicitly optimize data allocation and movement between memory hierarchies as they would do when programming today's GPUs (e.g. using CUDA). The simulated runtime library is able to tile data-parallel applications. Symbiotic tiling [22] is employed to assign neighboring tasks to adjacent threads for locality optimization. Characterizations show that symbiotic tiling already balances the workload reasonably well across cores.

[2]Compared to our previous work on dynamic warp subdivision [24], we increased the memory latency to 300 cycles which is common for current GPUs

Tech. Node	65 nm
WPU	1 GHz, 0.9 V Vdd, Alpha ISA up to 256 hardware thread contexts and a SIMD width of up to 64
I-Cache	16 KB, 4-way associative, 128 B line size 1 cycle hit latency, 4 MSHRs, LRU, write-back
D-Cache	32 KB, 8-way associative, 128 B line size MESI directory-based coherence Requests are sent to L2 on a FIFO basis 3 cycle hit latency, LRU, write-back 256 MSHRs each hosts up to 8 requests
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, store-and-forward contention is modeled, token-based flow control
Memory	300 cycles access latency

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

C. Benchmarks

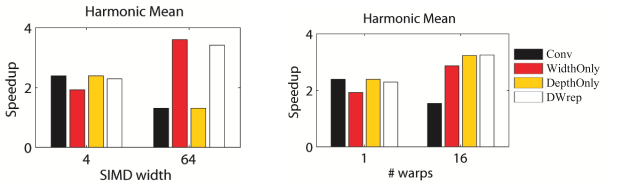
The throughput-oriented WPUs target data-parallel applications with large input data sets. We adopt the set of SIMD benchmarks provided by the MV5 [23] simulator. The same set of benchmarks have also been used to study dynamic warp subdivision [24]; it is therefore important to note that these were not selected to illustrate or emphasize any particular aspects of this study. The benchmarks are data-parallel kernels ported from Minebench [25], Splash2 [37], and Rodinia [8]. These benchmarks exhibit diverse data access and communication patterns [4], and cover the application domains of scientific computing, image processing, physics simulation, and data mining. The input size is set in such a way that we have sufficient parallelism and still have reasonable simulation times. Details are listed in Table II.

VI. An Investigation of Adaptation Mechanisms

An adaptation strategy can be regarded as a simple finite state machine implemented by the per-WPU adaptation state unit (ASU) in hardware. Instead of searching the entire SIMD design space by brute force, the ASU attempts to learn the preferred configuration based on gradients in the performance feedback. First, the ASU picks a dimension (*i.e.* either

	Benchmark Description
FFT	Fast Fourier Transform (Splash2 [37]). Spectral methods. Butterfly computation Input: a 1-D array of 262,144 (2^{18}) 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 [8]). Iterative partial differential equation solver Input: a 300×300 2-D grid, 100 iterations
LU	LU Decomposition (Splash2 [37]). 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 [25]). Distance aggregation using Map-Reduce. Input: 30,000 points in a 8-D space
SVM	Supervised Learning (MineBench [25]). Support vector machine's kernel computation. Input: 100,000 vectors with a 20-D space

Table II: Simulated benchmarks with descriptions and input sizes.



(a) Various SIMD widths with a single warp

(b) Various multi-threading depths with a SIMD width of 4

Figure 5: Comparing different adaptation strategies. Speedup is normalized to that of a system with single-threaded in-order cores. With narrow SIMD and few warps, all adaptation strategies perform similarly. (a) When SIMD width increases from 4 to 64, *WidthOnly* and *DWrep* outperform *Conv* and *DepthOnly* since they trade SIMD width for depth. (b) When the number of warps increases from 1 to 16, *DepthOnly* and *DWrep* perform best because they are able to deactivate some warps to reduce cache contention.

SIMD width or multi-threading depth). It then attempts to increase and decrease the value along that dimension in the subsequent two intervals. The amount to increase or decrease is discussed in Section VI-B. Section VI-C discusses the length of a sampling interval and how performance is evaluated after one interval. The ASU then chooses the direction which yields better performance. In the following intervals, it suggests subsequent SIMD organizations in that direction until it no longer gains performance. At this point, it may choose another dimension to adapt.

A. Exploring Various Strategies

We investigate several adaptation strategies including:

- *WidthOnly*, which only adjusts SIMD width. The number of active warps remains constant. The preferred SIMD width is identified by turning off more lanes in subsequent sampling intervals until it begins to hurt performance.
- *DepthOnly*, which only suspends or resumes warps to adjust depth, using an approach similar to *WidthOnly* to identify the preferred depth.
- *DWrep*, which adjusts depth until it converges, and then adjusts width until it converges; it repeats until neither depth nor width needs to be adjusted. Note that reducing SIMD width may generate more warp-slices, which increases the upper limit of multi-threading depth. We have also experimented with the same strategy with SIMD width adapted first, which yields similar performance.

We compare these strategies over various SIMD widths and multi-threading depths. The baseline system with conventional WPUs is referred to as *Conv*. Because of the space limitation, we only illustrate two scenarios in Figure 5. Our results show that *DWrep* performs consistently well on various organizations, therefore *DWrep* is used in Robust SIMD in the rest of this paper. Note that when the initial configuration is already the optimal configuration, adaptive strategies may perform slightly worse than *Conv* due to adaptation phases with suboptimal configurations.

We have also experimented with adapting depth and width simultaneously and with allowing multiple WPUs to collaboratively explore and evaluate various SIMD organizations. These strategies are susceptible to noise and are outperformed by*DWrep*.

B. Convergence Robustness

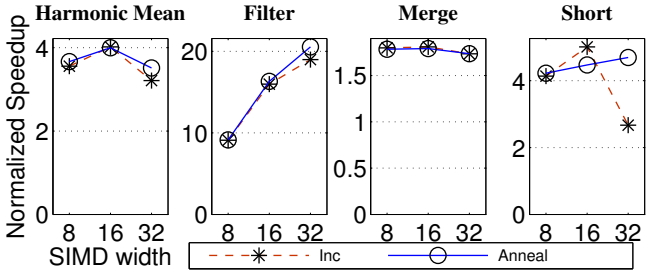
At the end of each sampling interval, the ASU has to offset the current SIMD organization with a certain stride to suggest a new configuration. A naïve implementation increments or decrements either SIMD width or multi-threading depth after each sampling interval. Such a convergence mechanism with a stride of one is named *Inc*.

However, our experiments show that *Inc* converges slowly and is susceptible to noise. To address these issues, we propose to first use a larger stride at beginning, and then gradually decrease the stride until it reaches one (*i.e.* incremental adjustment). We name this method *quasi-annealing* (*Anneal*), and it can be combined with any adaptation strategies described in Section VI-A. In our evaluation, the initial stride for SIMD width is set to half of number of available lanes. The initial stride for multi-threading depth is set to half the number of available warps. The stride is reduced by half once a preferred configuration is identified using the coarser stride. The adaptation then continues with the finer stride until the stride reaches one. Figure 6a compares *Inc* and *Anneal* and shows that the latter performs better with wide SIMD.

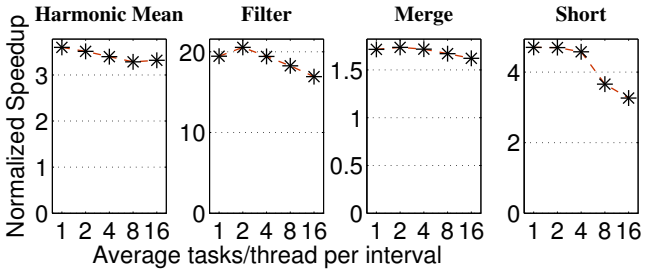
C. Evaluating a Particular SIMD Configuration

We assume that data parallel tasks within a parallel `for` loop are homogeneous and propose to use the average cycles per task (CPT) as the performance measure to be used by the ASU. While cycles per instruction (CPI) can also reflect overall performance, it is difficult to determine the right sampling interval for a CPI calculation; short sampling intervals may fail to capture a complete data-parallel task and generate a biased, noisy performance measure, while long sampling intervals introduce unnecessary overhead. Because the computational workload within a data parallel task differs across applications and application phases, we propose to adjust the sampling interval according to task lengths.

First, we predefine a desired number of data-parallel tasks to be completed by an average thread in one sampling interval. This number is set to two in the rest of the



(a) Comparing Robust SIMD performance with a converging stride of one (*Inc*) and that with annealed strides (*Anneal*). WPUs each have two warps and we vary the SIMD width.



(b) Comparing various sampling intervals on WPUs with two 32-wide warps.

Figure 6: Sensitivity study for convergence rate and interval length. Speedup is normalized to a system with four single-threaded, in-order cores. We show three benchmarks that exhibit different responses. The overall trend is shown by taking the harmonic mean of performance across all benchmarks. In (b), performance decreases with more tasks per interval due to sampling overhead; however, this overhead does not increase with larger workload.

paper based on results of the sensitivity study illustrated in Figure 6b. Multiplying this number with the number of threads per WPU, we derive the minimum number of tasks to be completed by a WPU within one sampling interval. The WPU then uses a hardware counter to count the number of completed tasks until it exceeds the minimum threshold. Once such a threshold is reached, the WPU will suspend any active threads when they finish their current tasks to prepare warp-slices for SIMD reconfiguration. According to our experiments, the synchronization overhead accounts for less than 9% of the adaptation phase, or less than 2% of the entire execution time. At the end of a sampling interval, the length of the interval in cycles is then divided by the number of completed tasks to estimate the average cycles per task (CPT). The lower the CPT is, the more preferred the SIMD organization is. Note that different WPUs monitor their sampling intervals independently.

There are cases where a parallel `for` loop contains only a few long-running tasks, resulting in insufficient tasks for adaptation. Merge, LU, and KMeans all have a portion of their execution time that exhibits such behavior, often in their latter phases of reduction. In these cases, we apply the heuristic that if a parallel `for` loop has too few tasks to warrant 30 sampling intervals, an empirically picked SIMD organization with two eight-wide warps will be applied instead; we found this to be the best default.

VII. Evaluation

The goal of Robust SIMD is to come up with a SIMD configuration that performs reasonably well across various SIMD applications *and* despite runtime variation in available cache capacity. In addition, the effort in space exploration should be reasonably low compared to previous techniques. We evaluate Robust SIMD (denoted with *RobustSIMD*) by demonstrating its performance over the design space and comparing the results against those of fixed SIMD organizations (denoted with *StaticSIMD*) as well as SIMD organizations with dynamic warp subdivision (denoted with *DWS*). The comparison is made by starting the experiments with the same initial configuration with regard to width and depth. We examine how different techniques respond to various applications and identify the overall best configuration for each technique. These preferred configurations are further compared under various degrees of D-cache pollution.

In this section, a SIMD organization with a width of W and a depth of D is denoted $W \times D$. Averaged performance is calculated using the harmonic mean. Performance measurements include both adaptation phase and execution phase. The adaptation phase accounts for about 6% of the total execution time on average, and it never exceeds 15% of the total execution time. Note that larger input size does not prolong adaptation time, because the number of tasks required in learning the optimal SIMD configuration is independent of the total number of data-parallel tasks.

A. Robustness Across Applications

We evaluate each technique with eight benchmarks, described in Section V-C. In each experiment, the workload of an application is evenly partitioned across four WPUs which execute in parallel. Performance is measured as the overall parallel execution time. Starting with WPUs with private 8-way associative, 32 KB D-caches, we conduct space exploration of various SIMD widths and depths at powers of two. Results shown are limited to a maximum of 64 hardware thread contexts per WPU. Our experiments show that having more threads per WPU actually degrades performance for all benchmarks due to cache thrashing.

StaticSIMD organizations yield the best mean performance when WPUs have a fixed width of eight and a fixed depth of four. However, for most benchmarks this "best on average" configuration is not the preferred (see Row 4 in Table III). Comparing Row 3 to Row 5 in Table III, we show that the "best on average" *StaticSIMD* organization loses 16% of performance when compared with the *static optimal case* defined as running each benchmark on its preferred *StaticSIMD* organization.

For *RobustSIMD* organizations, WPUs originating from a single, 32-wide warp perform best (Row 6 in Table III). Using *RobustSIMD*, WPUs with a small number of wide warps are likely to perform well uniformly across all benchmarks. This is because such configurations maximize throughput when data-parallel tasks are computationally intensive, and can be converted to multiple narrower warps or fewer threads to hide latency or reduce contention during memory intensive phases. As a result, *RobustSIMD* yields similar performance to the static optimal case and 17% better than the best *StaticSIMD* organization (compare Rows 5 and Row 6 in Table III). In fact, the "best-on-average" *RobustSIMD* performs better than the static optimal in some cases. The reasons are two-fold. First, *RobustSIMD* can converge to

1	Benchmarks	**Harmonic Mean**	FFT	Filter	HotSpot	LU	Merge	Short	KMeans	SVM
2	***32KB, 8-way assoc. D-caches***									
3	Best *StaticSIMD* (8×4)	3.47	3.23	9.47	6.22	4.03	1.26	4.28	4.56	4.30
4	App-specific best fixed config.		8×4	32×1	8×4	8×4	4×2	16×1	8×2	2×8
5	Static optimal speedup	4.15	3.23	20.18	6.22	4.03	1.76	4.96	5.20	5.02
6	Best *RobustSIMD* (32×1)	4.07	3.36	20.32	6.31	3.18	1.79	4.88	7.62	4.00
7	Best *DWS* (16×2)	4.14	3.51	11.68	5.49	3.80	1.95	3.97	6.92	4.79
8	Best *RobustDWS* (16×2)	4.27	3.33	16.62	6.16	3.66	1.96	4.96	6.90	4.59
9	***The same SIMD configuration with D-caches reduced to 16KB, 4-way***									
10	*StaticSIMD* (8×4)	1.24	2.03	2.75	1.11	0.64	0.85	0.88	2.14	2.70
11	App-specific best fixed config.		4×4	8×1	4×2	4×2	2×4	8×1	4×2	2×8
12	Static optimal speedup	2.92	2.94	6.99	3.14	2.17	1.57	3.29	3.51	4.01
13	*RobustSIMD* (32×1)	2.63	2.84	9.07	3.09	2.03	1.19	3.19	4.82	2.44
14	*DWS* (16×2)	1.62	2.23	2.76	1.29	0.84	1.70	0.91	4.90	3.66
15	*RobustDWS* (16×2)	2.98	2.93	9.83	2.71	2.03	1.72	3.14	4.91	3.66

Table III: Performance comparison between *StaticSIMD*, *RobustSIMD*, *DWS*, and *RobustDWS* across various benchmarks and D-cache settings. Performance numbers are shown as the speedup over single-threaded in-order cores.

nonstandard organizations where SIMD widths and depths are no longer powers of two. Second, *RobustSIMD* can also adapt to phase changes where applications prefer different SIMD configurations at different parallel code sections.

We also observe a couple of benchmarks often prefer *DWS*. Further analysis reveals two reasons. First, workloads such as LU and Merge spend significant time in `for` loops with too few parallel tasks to warrant effective adaptation. In these cases, *RobustSIMD* would use predefined heuristics described in Section VI-C, which leads to suboptimal performance. Second, workloads such as SVM have fast-changing phases that exhibit different divergence behaviors and memory intensities; therefore a stable SIMD configuration that performs consistently well may not even exist. As Section VII-C shows, Robust SIMD can be easily integrated into DWS, and the resulting technique, named *RobustDWS*, achieves the best performance among all (Row 8 in Table III).

B. Robustness under D-cache Pollution

The above results are obtained when each workload possesses the entire cache capacity. However, this may not be true in reality; not all cache lines are available to a running workload due to faults, power saving, or runtime pollution. For NVIDIA's Fermi [2] and future architectures, multiple kernels may be co-scheduled over the same WPU, leading to contention over the L1 storage. Such kernel co-scheduling can occur for HPC applications as well. Kernel co-scheduling can improve occupancy of hardware thread contexts when kernels have limited data-level parallelism[3]; it improves memory efficiency by co-executing computation-intensive and memory-intensive kernels [16]; it also enables software pipelining by co-locating producer kernels and consumer kernels for locality. However, all the above scenarios lead to "jittering" of the cache capacity available to a particular kernel. Hence, it is questionable whether the preferred SIMD configurations resulting from space exploration under an optimistic condition would continually perform well. With Robust SIMD, the WPUs can restart the adaptation phase runtime variation and re-adapt to the new environment.

Therefore, we study the "best-on-average" SIMD configurations in Section VII-A and compare their performance when D-caches are polluted. To quantify the intensity of pollution and measure its effect on performance, we manually inject pollution into D-caches. Such an approach is an approximation of the actual runtime dynamics and is similar to error injection methods used in evaluating error-resilience techniques [5], [39]. In this set of experiments, D-cache lines are randomly marked as polluted and cannot be used by the running workload. The assumption is that runtime environment's disruptiveness in cache accesses is in a steady state so that its intensity can be quantified in our evaluation. The percentage of polluted cache lines is varied from 12% to 75% of all cache lines. For each degree of pollution, we run every workload 10 times and report their average performance to account for variations.

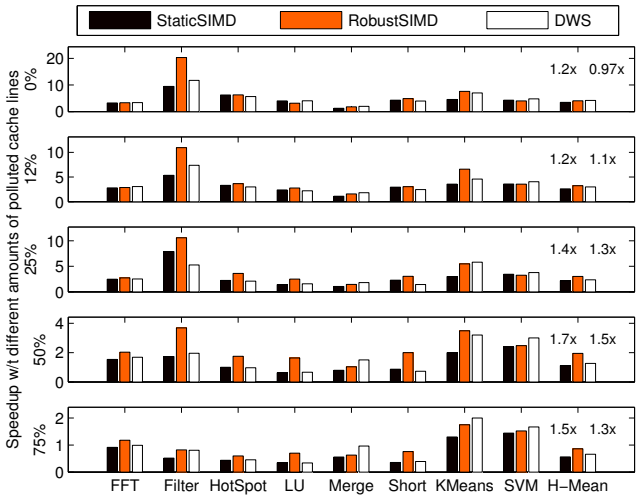


Figure 7: Comparing performance of the "best-on-average" *StaticSIMD* (8×4) (left), *RobustSIMD* (32×1) (middle), and *DWS* (16×2) (right) when 12%, 25%, 50%, and 75% of the D-cache lines are polluted. D-caches are 8-way associative and are sized 32 KB. Numbers are reported by averaging 10 independent runs to account for variation. Performance is normalized to that of *StaticSIMD* configured with (1×1) without any cache pollution. The harmonic mean across all applications are shown as "H-Mean". The left and right columns of text show the speedup of *RobustSIMD* compared to *StaticSIMD* and *DWS*, respectively.

[3]We have encountered such scenarios in an HPC application, GFMC [28]. It is a quantum physics application that performs Monte Carlo calculation for light nuclei. It has tremendous thread-level parallelism, but each thread has too few data-parallel tasks to benefit from GPU acceleration.

As Figure 7 shows, performance of *RobustSIMD* degrades much more gracefully than *StaticSIMD* or *DWS*. Although the "best-on-average" *DWS* performs similarly to *RobustSIMD* without cache pollution, its performance can easily fall victim to runtime variations of cache capacity. With only 25% of cache lines polluted, *RobustSIMD* performs 1.3× better than *DWS* in terms of execution time. The only benchmark that persistently prefers *DWS* is Merge; it spends significant time in loops with too few parallel tasks for *RobustSIMD* adaptation, and it is not sensitive to cache pollution.

C. Robust SIMD Combined with Dynamic Warp Subdivision

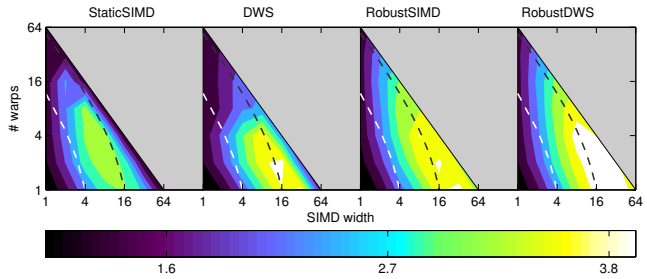


Figure 8: Comparing the speedup of *StaticSIMD*, *DWS*, *RobustSIMD*, and *RobustDWS* over various SIMD organizations. Speedup is normalized to the organization with four single-threaded, in-order cores. Lighter color indicates more speedup. D-caches are 8-way associative and are sized 32 KB. We experiment with SIMD widths and depths at powers of two, with up to 64 thread contexts per WPU. Dashed lines are contour lines connecting area-equivalent configurations.

Robust SIMD is complementary to DWS: the former adapts to runtime dynamics across different workloads and jittering cache capacity, but it requires a larger number or parallel tasks to ensure effective adaptation; the latter divides warps instantaneously based on branch or memory latency divergence, but cannot reduce the number of active threads nor re-adjust according to actual performance feedback. We therefore propose *RobustDWS* as a combination of Robust SIMD and DWS to show that Robust SIMD can benefit DWS as well. *RobustDWS* first calculates the total number of tasks from the parallel `for` loop sizes so that it abandons the use of Robust SIMD whenever there are too few tasks; the calculation time is negligible compared to the overall workload. If there are adequate tasks, *RobustDWS* chooses between Robust SIMD and DWS judiciously. For each data-parallel `for` loop, Robust SIMD is used at first without DWS to converge to a preferred organization. At the end of the adaptation phase and before the execution phase of Robust SIMD, the WPU runs another interval that uses DWS only. Eventually, the WPU chooses the better performer of the two. Note that occasionally *RobustDWS* can choose the worse of the two because the evaluation period for DWS may not be long enough. Nevertheless, it usually makes the right choice especially when the application has a strong preference of the two techniques.

Integrating Robust SIMD and DWS in the above manner can be achieved in hardware. The *warp-slice table* in Robust SIMD is structurally the same as the *warp-split table* in DWS. In addition, DWS can function well with the warp-slices' re-convergence stacks in Robust SIMD. The only additional logic needed by DWS is a register that records which threads hit in the cache and which missed during a SIMD cache access. The value of this register is used by DWS to subdivide a warp.

Figure 8 illustrates that *RobustDWS* performs best among all techniques. Not only does it achieve better performance than *DWS* and *RobustSIMD*, it also allows a wide selection of starting SIMD configurations to achieve near-optimal performance, thereby reducing design risk. The best *RobustDWS* organization has two 16-wide warps for each WPU, and it improves the performance of *RobustSIMD* by another 5% with a 32 KB D-cache. With 50% polluted cache lines, it improves the performance of *RobustSIMD* by 13%.

Figure 8 also reveals that *DWS* may perform worse than *StaticSIMD* with a modest SIMD width and depth (*e.g.*, in the 4×8 case). In contrast, *RobustSIMD* and *RobustDWS* perform consistently better than their corresponding *StaticSIMD* configuration. Finally, area analysis demonstrates that the best *RobustSIMD* and *RobustDWS* configurations perform better than any area-equivalent *StaticSIMD* configurations.

We have also explored other ways of integrating the two techniques. For example, DWS can be used during the adaptation phase as well as the execution phase of Robust SIMD. However, due to interference among the two, such close coupling of DWS and Robust SIMD does not provide robust performance.

VIII. Conclusions and Future Work

This paper studies SIMD over cache hierarchies for general purpose applications. We show that cache hierarchies affect the choices of SIMD width and multi-threading depth in different ways than streaming memories. Due to more obtrusive memory latency divergence, lower latency in non-L1 data accesses, and relatively unpredictable L1 contention, deep multi-threading with modest SIMD width no longer works well consistently. The preferred SIMD width and depth depends heavily on runtime dynamics and can vary due to different applications and jittering cache capacities. As a result, we propose Robust SIMD which provides wide SIMD and offers the flexibility to re-configure itself to narrower or deeper SIMD. Our experiments show that Robust SIMD achieves performance gains of 17% when compared to the best fixed SIMD organization. When available D-cache capacity is reduced by 25% due to runtime dynamics, performance degrades. However, in terms of execution time, Robust SIMD performs 1.4× better compared to a conventional SIMD architecture, and 1.3× better compared to dynamic warp subdivision.

To further reduce adaptation time, and to improve adaptation when there are only a few data parallel tasks, we can employ persistent adaptation: the preferred organization for an application can be generated using multiple executions and stored in a file. The system can then load the learned SIMD organization directly for future executions. Moreover, homogeneous, data-parallel tasks may prefer different SIMD configurations if their data accesses are irregular; in such circumstances, the adaptation phase can be restarted periodically even within the same parallel section.

IX. Acknowledgements

This work was supported in part by SRC grant No. 1607 and task 1972, 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). This research is also supported by the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] NVIDIA CUDA compute unified device architecture programming guide. *NVIDIA Corporation*, 2007.

[2] NVIDIA's next generation CUDA compute architecture: Fermi. *NVIDIA Corporation*, 2009.

[3] R. A. Alfieri. An efficient kernel-based implementation of POSIX threads. In *USTC*, 1994.

[4] K. Asanovic, R. Bodik, B. Christopher C., J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[5] A. Benso and P. Prinetto. *Fault injection techniques and tools for embedded systems reliability evaluation*.

[6] 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.

[7] OpenMP Architecture Review Board. OpenMP application program interface. http://www.openmp.org/mp-documents/spec30.pdf, 2008.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphisc processors using CUDA. *JPDC*, 2008.

[9] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *HPCA*, 2007.

[10] Intel Corporation. Intel news release: Intel unveils new product plans for high-performance computing. 2010.

[11] NVIDIA Corporation. GeForce GTX 280 specifications. 2008.

[12] N. Eisley, L.-S. Peh, and L. Shang. Leveraging on-chip networks for data cache migration in chip multiprocessors. In *PACT*, 2008.

[13] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ISCA*, 2002.

[14] 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.

[15] M. Gschwind. Chip multiprocessing and the Cell Broadband Engine. In *CF*, 2006.

[16] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling task parallelism in the CUDA scheduler. In *PMEA Workshop*, 2009.

[17] Intel Corporation. Intel Threading Building Blocks. http://www.threadingbuildingblocks.org.

[18] Khronos Group Std. The OpenCL Specification, Version 1.0. http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf, April 2009.

[19] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. *IEEE Micro*, 24:84–90, 2004.

[20] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. In *ISCA*, 2000.

[21] W. R. Mark, R. Steven, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *SIGGRAPH*, 22:896–907, 2003.

[22] J. Meng, J. W. Sheaffer, and K. Skadron. Exploiting inter-thread temporal locality for chip multithreading. In *IPDPS*, page 117, 2010.

[23] J. Meng and K. Skadron. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In *ICCD*, 2007.

[24] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA*, 2010.

[25] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. *IISWC*, 2006.

[26] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, , and A. Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *CGO*, 2011.

[27] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, 2000.

[28] S. C. Pieper, K. Varga, and R. B. Wiringa. Quantum Monte Carlo calculations of A=9,10 nuclei. In *Phys. Rev. C 66, 044310-1:14*, 2002.

[29] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[30] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39*, 2006.

[31] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector lane threading. *ICPP*, 0:55–64, 2006.

[32] 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.

[33] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, pages 234–244, New York, NY, USA, 2000.

[34] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, page 117, 2002.

[35] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for SIMD cores. In *SC*, 2009.

[36] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu. Trading off cache capacity for reliability to enable low voltage operation. In *ISCA*, 2008.

[37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *ISCA*, 1995.

[38] S.-H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA*, 2002.

[39] W. Zhang, S. Gurumurthi, M. Kandemir, and A. Sivasubramaniam. ICR: In-cache replication for enhancing data cache reliability. *ICDSN*, 2003.