Dynamic Inter-Thread Vectorization Architecture: extracting DLP from TLP

Sajith Kalathingal, Caroline Collange, Bharath Narasimha Swamy, André Seznec, Bharath N Swamy

Dynamic Inter-Thread Vectorization Architecture: extracting DLP from TLP

Sajith Kalathingal
Inria
sajith.kalathingal@inria.fr

Caroline Collange
Inria
caroline.collange@inria.fr

Bharath N. Swamy
Intel
bharath.n.swamy@intel.com

André Seznec
Inria
andre.seznec@inria.fr

***Abstract*—Threads of Single-Program Multiple-Data (SPMD) applications often execute the same instructions on different data. We propose the Dynamic Inter-Thread Vectorization Architecture (DITVA) to leverage this implicit data-level parallelism in SPMD applications by assembling dynamic vector instructions at runtime. DITVA extends an SIMD-enabled in-order SMT processor with an inter-thread vectorization execution mode. In this mode, multiple scalar threads running in lockstep share a single instruction stream and their respective instruction instances are aggregated into SIMD instructions. To balance thread- and data-level parallelism, threads are statically grouped into fixed-size independently scheduled warps. DITVA leverages existing SIMD units and maintains binary compatibility with existing CPU architectures.**

Our evaluation on the SPMD applications from the PARSEC and Rodinia OpenMP benchmarks shows that a 4-warp $\times$ 4-lane 4-issue DITVA architecture with a realistic bank-interleaved cache achieves $1.55\times$ higher performance than a 4-thread 4-issue SMT architecture with AVX instructions while fetching and issuing 51% fewer instructions, achieving an overall 24% energy reduction.

I. Introduction

Single-Program Multiple-Data (SPMD) applications express parallelism by creating multiple instruction streams executed by scalar threads running the same program but operating on different data. The underlying execution model for SPMD programs is the Multiple Instruction, Multiple Data execution model, i.e., threads execute independently between two synchronization points. The SPMD programming model often leads threads to execute very similar control flows: they often execute the same instructions on different data. The implicit data level parallelism (DLP) that exists across the threads of an SPMD program is neither captured by the programming model – threads execute asynchronously – nor leveraged by current processors.

Simultaneous Multi-Threaded (SMT) processors leverage multi-issue superscalar processors on parallel or multi-program workloads to achieve high single-core throughput whenever the workload features parallelism or concurrency [1], [2]. SMT cores are the building bricks of many commercial multi-cores including all the recent Intel and IBM high-end multi-cores. While SMT cores often exploit explicit DLP through Single Instruction, Multiple Data (SIMD) instructions, they do not leverage the implicit DLP present in SPMD applications.

In this paper, we propose the Dynamic Inter-Thread Vectorization Architecture (DITVA) to exploit the implicit DLP in SPMD applications dynamically at a moderate hardware cost. DITVA extends an in-order SMT architecture by dynamically aggregating instruction instances from different threads and steering them to SIMD units. To maximize dynamic vectorization opportunities, DITVA uses a fetch steering policy that favors lockstep execution of threads, while maintaining fairness guarantees to allow arbitrary thread interactions. In order to maintain the latency hiding abilities of SMT architectures, scalar threads are grouped into independent *warps*. DITVA preserves binary compatibility with existing general purpose CPU architectures and existing SPMD applications as it does not require any modification in the ISA. It even supports efficiently explicit SIMD instruction sets such as SSE and AVX on the same physical execution units, allowing programmers and compilers to freely combine explicitly-vectorized SIMD code and implicitly-vectorized SPMD code.

Our experiments on SPMD applications from the PARSEC and Rodinia benchmark suites [3], [4] show that the number of instructions fetched and decoded can be reduced, on average, by 51% on a 4-warp $\times$ 4-thread DITVA architecture compared with a 4-thread SMT. Coupled with a realistic memory hierarchy, this translates into a speed-up of $1.55\times$ over 4-thread in-order SMT, a very significant performance gain. DITVA provides these benefits at a limited hardware complexity since it relies essentially on the same control hardware as the SMT processor and the replication of the functional units by using SIMD units in place of scalar units. Since DITVA can leverage preexisting SIMD execution units, this benefit is achieved with 24% average energy reduction. Therefore, DITVA appears as a very energy-effective design to execute SPMD applications.

We motivate the DITVA proposition for a high throughput SPMD oriented processor architecture in Section II, and describe the DITVA architecture in Section III. Section IV evaluates performance and design tradeoffs. Section V reviews some related works.

II. Motivation

a) SMT architectures: SMT architectures aim at delivering throughput for any mix of threads without differentiating threads of a single parallel application from threads of a multi-program workload. Therefore, when threads from an SPMD application exhibit very similar control flows, SMT

architectures only benefit from these similarities by side-effects of sharing structures such as caches or branch predictors [5].

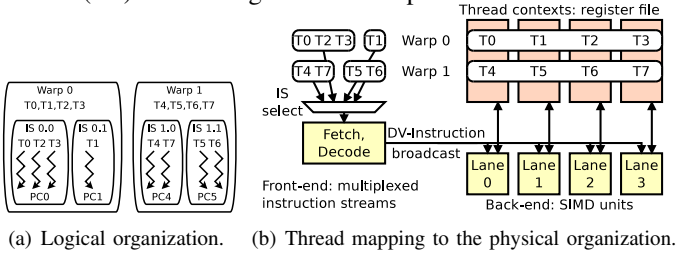
Most existing SMT architectures target both high single-thread performance and high parallel or multi-program performance. As a consequence, many commercial designs implement out-of-order execution. However, in the context of parallel applications, out-of-order execution may not be cost effective. An in-order 4-thread SMT 4-issue processor has been shown to reach 85% of the performance of an out-of-order 4-thread SMT 4-issue processor [6]. Therefore, in-order SMT appears as a good architecture tradeoff for implementing the cores of an SPMD oriented throughput processor.

b) Instruction redundancy across SPMD threads: Threads of SPMD applications usually execute very similar flows of instructions. They exhibit some control flow divergence due to different branch outcomes, but most of their instruction streams are similar. Also, SPMD applications typically resort to explicit synchronization barriers at certain execution points to enforce dependencies between tasks. Such barriers are natural control flow convergence points.

On a multi-threaded machine, e.g. an SMT processor, threads execute independently between barriers without any instruction level synchronization. However, prior studies have shown that the instruction fetch of 10 threads out of 16 on average could be mutualized if the threads were synchronized to progress in lockstep, on the PARSEC benchmarks [7]. We leverage this instruction redundancy to mutualize the front-end pipeline of an in-order SMT processor and create vector instructions dynamically, as a resource-efficient way to improve throughput on SPMD applications.

III. The Dynamic Inter-Thread Vectorization Architecture

Flynn's taxonomy classically breaks parallel architectures into Single Instruction stream, Single Data stream (SISD), Single Instruction stream, Multiple Data streams (SIMD) and Multiple Instruction streams, Multiple Data streams (MIMD) [8]. An *instruction stream* is generally assumed to mean a hardware thread in modern terms. However, we can decouple the notion of the instruction stream from the notion of the thread. In particular, multiple threads can share a single instruction stream, as long as they have the same program counter (PC) and belong to the same process.



(a) Logical organization. (b) Thread mapping to the physical organization.

Fig. 1. Overview of a 2-warp× 4-thread DITVA configuration

a) Logical thread organization: DITVA supports a number of hardware thread contexts, which we will refer to as (scalar) threads. Scalar threads are partitioned statically into n *warps* of m threads each, borrowing NVIDIA GPU terminology. In Figure 1(a), scalar threads T0 through T3 form Warp 0, while T4 to T7 form Warp 1.

Inside each warp, threads that have the *same PC* and process identifier share an Instruction Stream (IS). The concept of IS corresponds to warp-split [9] in the GPU architecture literature. While thread-to-warp assignment is static, a thread-to-IS assignment is dynamic: the number of IS per warp may vary from 1 to m during execution, as does the number of threads per IS. In Figure 1(a), scalar threads T0, T2 and T3 in Warp 0 have the same PC PC0 and share Instruction Stream 0.0, while thread T1 with PC PC1 follow IS 0.1.

The state of one Instruction Stream consists of one process identifier, one PC and an m-bit inclusion mask that tracks which threads of the warp belong to the IS. Bit i of the inclusion mask is set when thread i within the warp is part of the IS. Also, each IS has data used by the fetch steering policy, such as the call-return nesting level (Section III-C).

b) Mapping to physical resources: DITVA consists in a front-end that processes Instruction Streams and a SIMD back-end (Figure 1(b)). An instruction is fetched only once for all the threads of a given IS, and a single copy of the instruction flows through the pipeline. That is, decode, dependency check, issue and validation are executed only once. Each of the m lanes of the back-end replicates the register file and the functional units. A given thread is assigned to a fixed lane, e.g. T5 executes on Lane 1 in our example. Execution, including operand read, operation execution and register result write-back is performed in parallel on the m lanes. A notable exception is instructions that already operate on vectors, such as SSE and AVX, that are executed in multiple waves over the whole SIMD width.

c) Notations: We use the notation $nW \times mT$ to represent a DITVA configuration with n Warps and m Threads per warp. An $nW \times 1T$ DITVA has 1 thread and 1 IS per warp, and is equivalent to an n-thread SMT. At the other end of the spectrum, a $1W \times mT$ DITVA has all threads share a single pool of IS without restriction. A vector of instruction instances from different threads of the same IS is referred to as a DV-instruction.

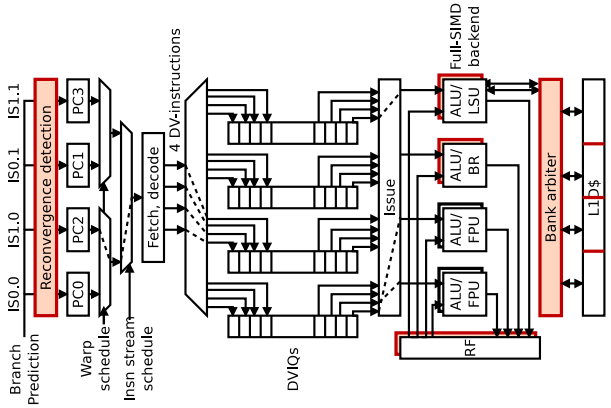


Fig. 2. Overview of a $2W \times 2T$, 4-issue DITVA pipeline. Main changes from SMT are highlighted.

In the remainder of the section, we first describe the modifications required in the pipeline of an in-order SMT processor to implement DITVA and particularly in the front-

end engine to group instructions of the same IS. Then we address the specific issue of data memory accesses. Finally, as maintaining/acquiring lockstep execution mode is the key enabler to DITVA efficiency, we describe the fetch policies that could favor such acquisition after a control flow divergence.

A. Pipeline architecture

We describe the stages of the DITVA pipeline, as illustrated in Figure 2.

1) Front-end: The DITVA front-end is essentially similar to an SMT front-end, except it operates at the granularity of Instruction Streams rather than scalar threads.

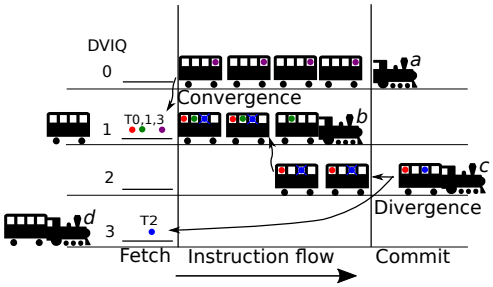


Fig. 3. Instruction stream tracking in DITVA. Instruction Streams (a, b, c, d) are illustrated as trains, DVIQs as tracks, DV-instructions as train cars, and scalar instructions as passengers.

a) Branch prediction and reconvergence detection: Within the front-end, both the PC and inclusion mask of each IS are speculative. An instruction address generator initially produces a PC prediction for one IS based on the branch history of the first active scalar thread of the IS. After instruction address generation, the PC and process identifier of the predicted IS_i, are compared with the ones of the other ISs of the same warp. A match between IS_i and IS_j indicates they have converged and may be merged. In such case, the mask of IS_i is updated to the logical OR of its former mask and the mask of IS_j, while IS_j is aborted. Figure 3 illustrates convergence happening between threads 0 and 1 in IS b with thread 3 in IS a. IS b contains threads 0, 1 and 3 after convergence, so its inclusion mask is now 1101. IS a is aborted. Earlier in time, convergence did also happen between threads 0 and 2 in IS c and thread 1 in IS b. All the threads of an IS share the same instruction address generation, by speculating that they will all follow the same branch direction. Unlike convergence, thread divergence within an IS is handled at instruction retirement time by leveraging branch misprediction mechanisms, as will be described in Section III-A5.

b) Fetch and decode: Reflecting the two-level organization in warps and ISs, instruction fetch obeys a mixed fetch steering policy. First, a warp is selected following a similar policy as in SMT [1], [10]. Then, an intra-warp instruction fetch steering policy selects one IS within the selected warp. The specific policy will be described in Section III-C. From the selected IS PC, a block of instructions is fetched.

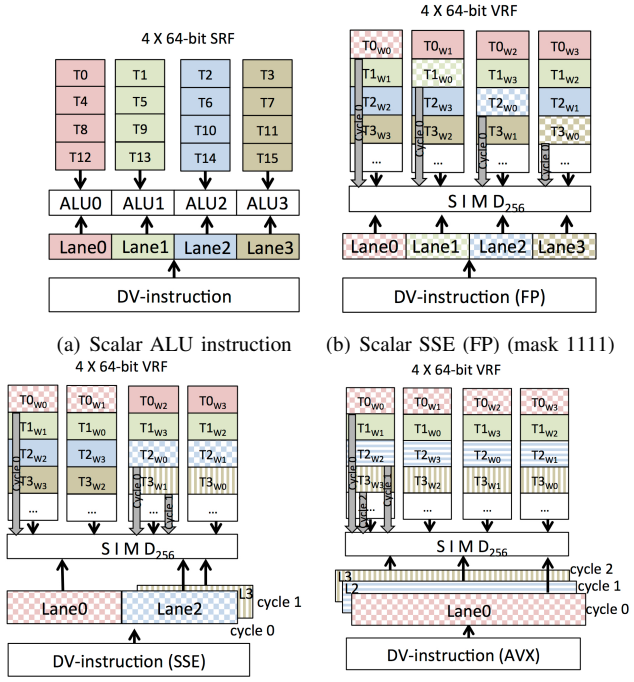
Instructions are decoded and turned into DV-instructions by assigning them an m-bit speculative mask. The DV-instruction then progresses in the pipeline as a single unit. The DV-instruction mask indicates which threads are expected to execute the instruction. Initially, the mask is set to the IS mask. However, as the DV-instruction flows through the pipeline, its mask can be narrowed by having some bits set to zero whenever an older branch is mispredicted, or an exception is encountered for one of its active threads.

After the decode stage, DV-instructions are pushed in a DV-instruction queue (DVIQ) associated with the IS. In a conventional SMT, instruction queues are typically associated with individual threads. DITVA applies this approach at the IS granularity: each DVIQ tail is associated with one IS. Unlike in SMT, instructions that are further ahead in the DVIQ may not necessarily belong to the IS currently associated with the DVIQ, due to potential IS divergence and convergence. For instance in Figure 3, DVIQ 2 contains instructions of threads T0 and T2, while the IS associated with the tail of DVIQ 2 has no active threads. The DV-instruction mask avoids this ambiguity.

2) In-order issue enforcement and dependency check: On a 4-issue superscalar SMT processor, up to 4 instructions are picked from the head of the instruction queues on each cycle. In each queue, the instructions are picked in-order. In a conventional in-order superscalar microprocessor, the issue queue ensures that the instructions are issued in-order. In DITVA, instructions from a given thread T may exist in one or more DVIQs. To ensure in-order issue in DITVA, we maintain a sequence number for each thread. Sequence numbers track the progress of each thread. On each instruction fetch, the sequence numbers of the affected threads are incremented. Each DV-instruction is assigned an m-wide vector of sequence numbers upon fetch, that corresponds to the progress of each thread fetching the instruction. The instruction issue logic checks that sequence numbers are consecutive for successively issued instructions of the same warp. As DVIQs maintain the order, there will always be one such instruction at the head of one queue for each warp.

To avoid any ambiguity, we use more sequence numbers than the maximum number of instructions belonging to a given thread in all DVIQs, which is bounded by the total number of DVIQ entries assigned to a warp. For instance, if the size of DVIQs is 16 and $m = 4$, 6-bit sequence numbers are sufficient, and each DV-instruction receives a 24-bit sequence vector.

A DV-instruction can only be issued once all its operands are available. A scoreboard tracks instruction dependencies. In an SMT having n threads with r architectural registers each, the scoreboard consists of a nr data dependency table with 8 ports indexed by the source register IDs of the 4 pre-issued 2-input instructions. In DITVA, unlike in SMT, an operand may be produced by several DV-instructions from different ISs, if the consumer instruction lies after a convergence point. Therefore, the DITVA scoreboard mechanism must take into account all older in-flight DV-instructions of the warp to ensure operand availability, including instructions from other DVIQs. As sequence numbers ensure that each thread issues at most 4 instructions per cycle, the scoreboard can be partitioned between threads as m tables of nr entries with 8 ports.



(a) Scalar ALU instruction (b) Scalar SSE (FP) (mask 1111)

(c) Packed 128-bit SSE (mask 1011) (d) Packed 256-bit AVX (mask 1011)

Fig. 4. Operand collection on $4W \times 4T$ DITVA depending on DV-instruction type and execution mask. 'w' represents a 64-bit word

3) Execution: register file and functional units: On an in-order SMT processor, the register file features n instances of each architectural register, one per thread. The functional units are not strictly associated with a particular group of registers and an instruction can read its operands or write its result to a single monolithic register file.

In contrast, DITVA implements a partitioned register file; each of the m sub-files implements a register context for one thread of each warp. DITVA also replicates the scalar functional units m times and leverages the existing SIMD units of a superscalar processor for the execution of statically vectorized SIMD instructions.

Figure 4(a) shows the execution of a scalar DV-instruction (i.e. dynamically vectorized instruction from multi-thread scalar code) in a 4W×$4T$ DITVA. A scalar DV-instruction reads different thread instances of the same registers in each of the m register files. It executes on m similar functional units and writes the m results to the same register in the m register files, in a typical SIMD fashion. All these actions are conditioned by the mask of the DV-instruction. Thus, the DITVA back-end is equivalent to an SIMD processor with per-lane predication.

4) Leveraging explicit SIMD instructions: Instruction sets with SIMD extensions often support operations with different vector lengths on the same registers. Taking the x86_64 instruction set as an example, AVX instructions operate on 256-wide registers, while packed SSE instructions support 128-bit operations on the lower halves of AVX architectural registers. Scalar floating-point operations are performed on the low-order 64 or 32 bits of SSE/AVX registers. We assume AVX registers may be split into four 64-bit slices.

Whenever possible, DITVA keeps explicit vector instructions as contiguous vectors when executing them on SIMD units. This maintains the contiguous memory access patterns of vector loads and stores. In order to support both explicit SIMD instructions and dynamically vectorized DV-instructions on the same units without cross-lane communication, the vector register file of DITVA is banked using a hash function. Rather than making each execution lane responsible for a fixed slice of vectors, slices are distributed across lanes in a different order for each thread. For a given thread i, the lane j is responsible for the slice $i \oplus j$, $\oplus$ being the exclusive or operator. All registers within a given lane of a given thread are allocated on the same bank, so the bank index does not depend on the register index.

This essentially free banking enables contiguous execution of full 256-bit AVX instructions, as well as partial dynamic vectorization of 128-bit vector and 64-bit scalar SSE instructions to fill the 256-bit datapath. Figure 4(b) shows the execution of a scalar floating-point DV-instruction operating on the low-order 64-bit of AVX registers. The DV-instruction can be issued to all lanes in parallel, each lane reading a different instance of the vector register low-order bits. For a 128-bit SSE DV-instruction, lanes 0,2 or 1,3 can be executed in the same cycle. Figure 4(c) shows the pipelined execution of a SSE DV-instruction with mask 1011 in a 4W×$4T$ DITVA. In figure 4(c), T0 and T2 are issued in the first cycle and T1 is issued in the subsequent cycle. Finally, the full-width AVX instructions within a DV-instructions are issued in up to m successive waves to the pipelined functional units. Time-compaction skips SIMD instructions of inactive threads, as in vector processors. Figure 4(d) shows the execution of a AVX DV-instruction with mask 1011 in a $4W \times 4T$ DITVA.

5) Handling misprediction, exception or divergence: Branch mispredictions or exceptions require repairing the pipeline. On an in-order SMT architecture, the pipeline can be repaired through simply flushing the subsequent thread instructions from the pipeline and resetting the speculative PC to the effective PC.

In DITVA, we generalize branch divergence, misprediction and exception handling through a unified mechanism. Branch divergence is detected at branch resolution time, when some threads of the current IS, IS_i, actually follow a different control flow direction than the direction the front-end predicted. IS_i is split into two instruction streams: IS_i continues with the scalar threads that were correctly predicted, and a new stream IS_j is spawned in the front-end for the scalar threads that do not follow the predicted path. The inclusion masks of both IS are adjusted accordingly: bits corresponding to non-following threads are cleared in IS_i mask and set in IS_j mask. For instance, in Figure 3, ISc with threads T0 and T2 is split to form the new ISd with thread T2. Instructions of thread T2 are invalidated within the older DV-instructions of ISc as well as ISb. Handling a scalar exception would be similar to handling a divergence. The bits corresponding to the mispredicted scalar threads are also cleared in all the masks of the DV-instructions in progress in the pipeline and in the DVIQs. In Figure 3, they correspond to disabling thread T2 in the DV-instructions from IS_2 and IS_1.

The same mechanism handles both branch misprediction and divergence. Divergence occurs when a subset of the threads follows the predicted path, and another subset follows an alternate path. A full branch misprediction is a special case of divergence when the set of threads that follow the predicted path is empty, and the set of threads that follow the alternate path is the full IS.

Some bookkeeping is needed in the case of a full branch misprediction. In that case, the masks of some DV-instructions become null, i.e. no valid thread remains in IS_1. These DV-instructions have to be flushed out from the pipeline to avoid consuming bandwidth at execution time. This bookkeeping is easy to implement as all DV-instructions with null masks are at the head of the DVIQ. Likewise, an IS with an empty mask is aborted.

As DITVA provisions m IS slots and DVIQs per warp, and the masks of ISs do not overlap, resources are always available to spawn the new ISs upon divergence. The only case when all IS_s slots are occupied is when each IS has only one thread. In that case, a divergence can only be a full misprediction, and the new IS can be spawned in the slot left by the former one.

True branch mispredictions in DITVA have the same performance impact as a misprediction in SMT, i.e. the overall pipeline must be flushed for the considered DV-warp. On the other hand, simple divergence has no significant performance impact as it does not involve any "wrong path": both branch paths are eventually taken.

B. Data memory accesses

A data access operation in a DV-instruction (*DV-load* or *DV-store*) may access up to m data words in the cache. These m words may belong to m distinct cache lines and/or to m distinct virtual pages. Servicing these m data accesses on the same cycle would require a fully multiported data cache and a fully multiported data TLB. The hardware cost of a multiported cache is prohibitively high. Truly shared data demands implementing multiple effective ports, rather than simply replicating the data cache. Instead, DITVA relies on a banked data cache. Banking is performed at cache line granularity. The load data path supports concurrent access to different banks, as well as the special case of several threads accessing the same element, for both regular and atomic memory operations. In case of conflicts, the execution of a DV-load or a DV-store stays atomic and spans over several cycles, thus stalling the pipeline for all its participating threads.

We found that straightforward bank interleaving using the low order bits on the L1 data cache leads bank conflicts ranging from mild (20 conflicts per 1000 instructions) to severe (400 conflicts per 1000 instructions). Many such conflicts are caused by concurrent accesses to the call stacks of different threads. When the stack base addresses are aligned on page boundaries, concurrent accesses at the same offset in different stacks result in bank conflicts. Our observation confirms the findings of prior studies [11], [7]. To reduce such bank conflicts for DV-loads and DV-stores, we use a fully hashed set index. For a 16-bank cache interleaved at 32-bit word granularity, we use lower bits from 12 to 15 and higher bits from 24 to 27 and hash them for banking. We find this hashing mechanism is effective in reducing bank conflicts by 21% on average.

Maintaining equal contents for the m copies of the TLB is not as important as it is for the data cache: there are no write operations on the TLB. Hence, the data TLB could be implemented just as m copies of a single-ported data TLB. However, all threads do not systematically use the same data pages. That is, a given thread only references the pages it directly accesses in its own data TLB. On the set of benchmarks presented in Section IV, we find the miss rate of the 64-entry split TLB for four lanes DITVA is in the same range as the one of the 64-entry for SMT. If the TLB is unified, 256-entry is needed to reach the same level of performance. Thus, using split TLBs appears as a sensible option to avoid the implementation complexity of a unified TLB.

A DV-load (resp. DV-store) of a full 256-bit AVX DV-instruction is pipelined. Each data access request corresponding to the participant thread is serviced in the successive cycles. For a 128-bit SSE DV-instruction, data access operation from lane 0,2 or 1,3 are serviced in the same cycle. Any other combination of two or more threads are pipelined. For example, a DV-load with threads 0,1 or 0,1,2 would be serviced in 2 cycles. DITVA executes DV-instructions in-order. Hence, a cache miss on one of the active threads in a DV-load stalls the instruction issue of all the threads in the DV instruction.

C. Maintaining lockstep execution

DITVA has the potential to provide high execution bandwidth on SPMD applications when the threads execute very similar control flows on different data sets. Unfortunately, threads lose synchronization as soon as their control flow diverges. Apart from the synchronization points inserted by the application developer or the compiler, the instruction fetch policy and the execution priority policy are two possible vehicles to restore lockstep execution.

One of the most simple yet fairly efficient fetch policies to reinitiate lockstep execution is MinSP-PC [12], [7]. The highest priority is given to the thread with the deepest call stack, based on the relative stack pointer address or call/return count. On a tie, the thread that has the minimum PC is selected. Assuming a downward growing stack, MinSP gives priority for the deepest function call nesting level. When there is a tie the priority is based on the minimum value of PC which gives a more fine grained synchronization.

As general-purpose parallel applications may have active waiting loops that lead to deadlock with MinSP-PC, DITVA uses a hybrid Round-Robin/MinSP-PC instruction fetch policy. The MinSP-PC policy helps restore lockstep execution and Round-Robin guarantees forward progress for each thread. To guarantee that any thread T will get the instruction fetch priority periodically, the RR/MinSP-PC policy acts as follows. Among all the ISs with free DVIQ slots, if any IS has not got the instruction fetch priority for $(m+1) \times n$ cycles, then it gets the priority. Otherwise, the MinSP-PC IS is scheduled.

TABLE I
SIMULATOR PARAMETERS

L1 data cache	32 KB, 16 ways LRU, 16 banks, 2 cycles
L2 cache	4MB, 16 ways LRU, 15 cycles
L2 miss latency	215 cycles
Branch predictor	64-Kbit TAGE [14]
DVIQs	$n \times m$ 16-entry queues
IS select	MinSP-PC + RR every $n(m+1)$ cycles
Fetch and decode	4 instructions per cycle
Issue width	4 DV-instructions per cycle
Functional units (SMT)	4 64-bit ALUs, 2 256-bit AVX/FPUs, 1 mul/div, 1 256-bit load/store, 1 branch
Functional units (DITVA)	2 $m \times$ 64-bit ALUs, 2 256-bit AVX/FPUs, 1 $m \times$ 64-bit mul/div, 1 256-bit load/store

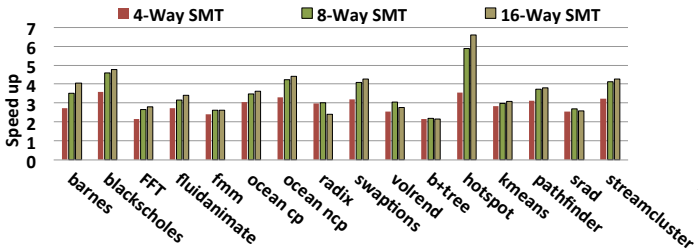


Fig. 5. Speed-up with thread count in the baseline SMT configuration, normalized to single-thread performance

This hybrid fetch policy is biased toward the IS with minimum stack pointer or minimum PC to favor thread synchronization, but still guarantees that each thread will make progress. In particular, when all threads within a warp are divergent, the MinSP-PC thread will be scheduled twice every $m+1$ scheduling cycles for the warp, while each other thread will be scheduled once every $m+1$ cycles.

Since warps are static, convergent execution does not depend on the prioritization heuristics of the warps. The warp selection is done with round robin priority to ensure fairness for each of the independent thread groups.

IV. EVALUATION

A. Experimental Framework

Simulating DITVA involves a few technical challenges. First, we need to compare application performance for different thread counts. Second, the efficiency of DITVA is crucially dependent on the relative execution order of threads. Consequently, instructions per cycle cannot be used as a proxy for performance, and common sampling techniques are inapplicable. Instead, we simulate full application kernels, which demands a fast simulator.

We model DITVA using an in-house trace-driven x86_64 simulator. A Pin tool [13] records one execution trace per thread of one SPMD application. The trace-driven DITVA simulator consumes the traces of all threads concurrently, scheduling their instructions in the order dictated by the fetch steering and resource arbitration policies.

Thread synchronization primitives such as locks need a special handling in this multi-thread trace-driven approach since they affect thread scheduling. We record all calls to synchronization primitives and enforce their behavior in the simulator to guarantee that the order in which traces are replayed results in a valid scheduling. In other words, the simulation of synchronization instructions is execution-driven, while it is trace-driven for all other instructions.

Just like SMT, DITVA can be used as a building block in a multi-core processor. However, to prevent multi-core scalability issues from affecting the analysis, we focus on the micro-architecture comparison of a single core in this study. To account for memory bandwidth contention effects in a multi-core environment, we simulate a throughput-limited memory with 2 GB/s of DRAM bandwidth per core. This corresponds to a compute/bandwidth ratio of 32 FLOPS per byte in the $4W \times 4T$ DITVA configuration, which is representative of current multi-core architectures. We compare two DITVA core configurations against a baseline SMT processor core with AVX units. Table I lists the simulation parameters of both micro-architectures. DITVA leverages the 256-bit AVX/FPU unit to execute scalar DV-instructions in addition to the two $m \times 64$-bit ALUs, achieving the equivalent of four $m \times 64$-bit ALUs.

We evaluate DITVA on SPMD benchmarks from the PARSEC [3] and Rodinia [4] suites. We use PARSEC benchmark applications that have been parallelized with pthread library. We considered the OpenMP version of the Rodinia benchmarks. All are compiled with AVX vectorization enabled. We simulate the following benchmarks: *Barnes*, *Blackscholes*, *Fluidanimate*, *FFT*, *Fmm*, *Swaptions*, *Radix*, *Volrend*, *Ocean CP*, *Ocean NCP*, *B+tree*, *Hotspot*, *Kmeans*, *Pathfinder*, *Srad* and *Streamcluster*. PARSEC benchmarks use the *simsmall* input dataset.

Figure 5 shows the speed-up of SMT configurations with 4, 8 and 16 threads over single threaded applications. Applications exhibit diverse scaling behavior with thread count. *FFT*, *Ocean*, *Radix*, *B+tree* and *Srad* tend to be bound by memory bandwidth, and their performance plateaus or decreases after 8 threads. *Volrend* and *Fluidanimate* also have a notable parallelization overhead due to thread state management and synchronization. In the rest of the evaluation, we will consider the 4-thread SMT configuration ($4W \times 1T$) with AVX as our baseline. We will consider $4W \times 2T$ DITVA, i.e., 4-way SMT with two dynamic vector lanes, $2W \times 8T$ DITVA, i.e., 2-way SMT with eight dynamic vector lanes and $4W \times 4T$ DITVA, i.e., 4-way SMT with 4 lanes.

B. Throughput

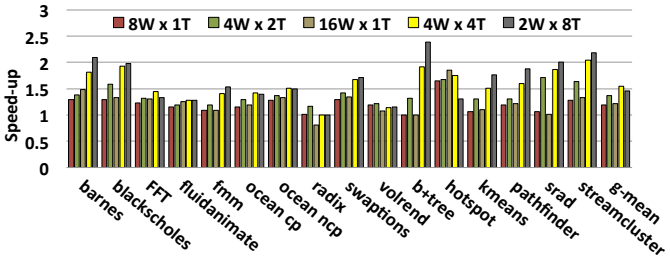


Fig. 6. Speed-up over 4-thread SMT as a function of warp size

Figure 6 shows the speed-up achieved for $4W \times 2T$ DITVA, $4W \times 4T$ DITVA and $2W \times 8T$ DITVA over 4-thread SMT with AVX instructions. For reference, we illustrate the performance of SMT configurations with the same scalar thread count ($16W \times 1T$ and $8W \times 1T$). On average, $4W \times 2T$ DITVA achieves 37% higher performance than 4-thread SMT and $4W \times 4T$ DITVA achieves 55% performance improvement.

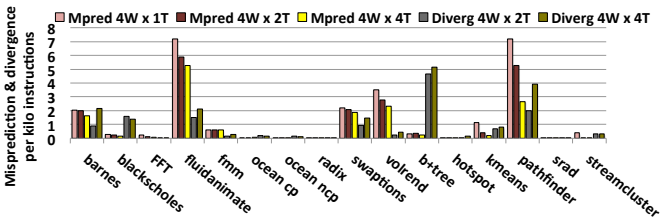


Fig. 7. Divergence and mispredictions per thousand instructions

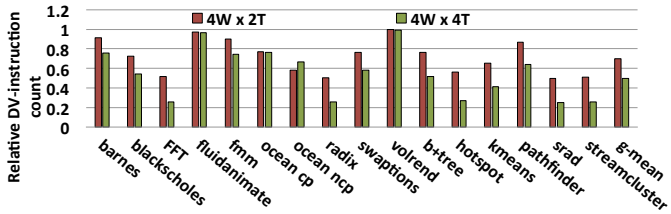


Fig. 8. DV-instruction count reduction over 4-thread SMT as a function of warp size

The $4W \times 4T$ DITVA also achieves 34% speed-up over 16-thread SMT. The $2W \times 8T$ DITVA achieves 46% speed-up over 4-thread SMT. Widened datapaths and efficient utilization of AVX units to execute dynamically vectorized instructions enable these performance improvements. Although $2W \times 8T$ DITVA has twice the SIMD width of $4W \times 4T$ DITVA, it has half as many independent warps. This TLP reduction aggravates stalls during long latency operations. We find that the best performance-cost tradeoffs are obtained by balancing homogeneous DLP and heterogeneous TLP.

C. Divergence and mispredictions

Figure 7 illustrates the divergence and misprediction rates for respectively single-lane (i.e. SMT), two-lane and four-lane DITVA configurations. Mispredictions in DITVA have the same performance impact as mispredictions in SMT. Divergences can impact time to re-convergence, but have no significant performance impact as both branch paths are eventually taken. As expected, we observe the highest misprediction rate on divergent applications. Indeed, we found that most mispredictions happen within the IS that are less populated, typically with one or two threads only.

D. Front-end utilization

Dynamic vectorization reduces the number of instructions going through the front-end. Figure 8 shows the ratio of the DV-instruction count over the individual instruction count for $4W \times 2T$ DITVA and $4W \times 4T$ DITVA. In average on our benchmark set, this ratio is 69% for $4W \times 2T$ DITVA and 49% for $4W \times 4T$ DITVA. Applications *Radix*, *FFT*, *Hotspot*, *Srad* and *Streamcluster* have nearly perfect dynamic vectorization, while the DV-instruction count reduction in *Volrend*, *Fluidanimate* and *Ocean* is compensated by the parallelization overhead caused by the thread count increase.

E. Power and energy

We modeled a baseline SMT processor and DITVA within McPAT [15], assuming a 2 GHz clock in 45nm technology with power gating. It follows the configuration depicted on Table I, except the cache was modeled as 64 KB 8-way as we could not model the banked 32 KB 16-way configuration in McPAT. As in Section IV, we assume that DITVA is built on top of an SMT processor with 256-bit wide AVX SIMD execution units and that these SIMD execution units are reused in DITVA.

McPAT estimates report an average energy reduction of 22% and 24% for $4W \times 2T$ and $4W \times 4T$ DITVA respectively. The energy reduction is the result of both a decrease in runtime (Figure 6) and a reduction in the number of fetched instructions, mitigated by an increase in static power from the wider execution units.

V. Related work

1) SIMD and vectors: SIMD and/or vector execution have been considered as early as the 1970s in vector supercomputers [16]. Short-vector SIMD instruction-set extensions are commonplace in general-purpose processors. Recent work enables the compilation of SPMD applications to SIMD or vector instruction sets [17], [18]. However, a change in the vector length of SIMD instructions requires recompiling or even rewriting programs. In contrast, SPMD applications typically spawn a runtime-configurable number of worker threads and can scale on different platforms without recompilation. Vector processors typically support variable-size vectors, but they require advanced prefetching or memory decoupling in order to overlap memory latency with computations [19]. DITVA runs multiple independent warps that cover each-other's long-latency operations. By translating TLP into DLP dynamically, DITVA offers the flexibility to select the vector length (warp size) that best suits each micro-architecture while exploiting the remaining parallelism as TLP, without compiler or programmer involvement.

2) The SIMT execution model: Like DITVA, SIMT architectures can vectorize the execution of multi-threaded applications at warp granularity, but they require a specific instruction set to convey branch divergence and convergence information to the hardware [20]. GPU compilers emit explicit instructions to mark convergence points in the binary program. The SIMT stack-based divergence tracking mechanisms handle user-level code with a limited range of control-flow constructs. They do not support exceptions or interruptions, which prevents their use with a general-purpose system software stack. Various works extend the SIMT model to support more generic code [21], [22] or more flexible execution [23], [24], [25]. However, they all target applications specifically written for GPUs, rather than general-purpose parallel applications.

3) Instruction redundancy in SMT: Minimal Multi-threading or MMT favors thread synchronization in the front-end of an SMT core to combine the instruction fetch and decode, and avoid redundant computation between threads [26]. Instructions that operate on different data are broken back into independent scalar instructions executed independently in a conventional out-of-order engine. Execution Drafting synchronizes threads running the same code and shares the instruction control

logic to improve energy efficiency [27]. It targets both multi-thread and multi-process applications by allowing lockstep execution at arbitrary addresses. MMT and Execution Drafting primarily target data-flow redundancy. DITVA targets control-flow redundancy, although it could be extended to exploit data-flow redundancy through dynamic scalarization techniques proposed for SIMT [28]. Both MMT and Execution Drafting seek to run all threads together in lockstep as much as possible. However, we find that full lockstep execution is not always desirable as it defeats the latency tolerance purpose of SMT. To address this issue, DITVA groups threads into SIMT-style warps, which are scheduled independently and provide latency hiding capabilities.

VI. Conclusion

We proposed the DITVA architecture to generalize the successful SIMT GPU execution model to general-purpose SMT CPUs. DITVA vectorizes instructions dynamically across threads like SIMT GPUs, but retains binary compatibility with general-purpose CPUs. Compared with an in-order SMT core architecture, it achieves high throughput on the parallel sections of the SPMD applications by extracting dynamic data-level parallelism at runtime.

DITVA maintains competitive single-thread and divergent multi-thread performance by using branch prediction and speculative predicated execution. By relying on a simple thread scheduling policy favoring convergence and by handling branch divergence at the execute stage as a partial branch misprediction, most of the complexity associated with tracking and predicting thread divergence and convergence can be avoided. To support concurrent memory accesses, DITVA implements a bank-interleaved cache with a fully hashed set index to mitigate bank conflicts. DITVA leverages the possibility to use TLBs with different contents for the different threads. It uses a split TLB much smaller than the TLB of an in-order SMT core.

Our simulation shows that $4W \times 2T$ and $4W \times 4T$ DITVA processors are cost-effective design points. For instance, a $4W \times 4T$ DITVA architecture reduces instruction count by 51% and improving performance by 55% over a 4-thread 4-way issue SMT on the SPMD applications from PARSEC and OpenMP Rodinia. While DITVA induces some silicon area and static energy overheads over an in-order SMT, by leveraging the preexisting SIMD execution units to execute the DV-instructions, DITVA can be very energy effective on SPMD code. A DITVA-based multi-core or many-core would achieve very high parallel performance on SPMD parallel sections.

As DITVA shares some of its key features with the SIMT execution model, many micro-architecture improvements proposed for SIMT could also apply to DITVA. For instance, more flexibility could be obtained using Dynamic Warp Formation [23] or Simultaneous Branch Interweaving [24], Dynamic Warp Subdivision [9] could improve latency tolerance by allowing threads to diverge on partial cache misses, and Dynamic Scalarization [28] could further unify redundant data-flow across threads.

Acknowledgement This work was supported by the Inria PROSPIEL Associate Team.

References

[1] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ISCA*, 1995.
[2] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *ISCA*, 1996.
[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, 2008.
[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[5] S. Hily and A. Seznec, "Branch prediction and simultaneous multithreading," in *PACT*, 1996.
[6] ——, "Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading," in *HPCA*, 1999.
[7] T. Milanez, C. Collange, F. M. Q. Pereira, W. Meira, and R. Ferreira, "Thread scheduling and memory coalescing for dynamic vectorization of SPMD workloads," *Parallel Computing*, vol. 40, no. 9, 2014.
[8] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 948–960, 1972.
[9] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ISCA*, 2010.
[10] S. Eyerman and L. Eeckhout, "A memory-level parallelism aware fetch policy for SMT processors," in *HPCA*, 2007.
[11] J. Meng and K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling," in *ICCD*, 2009.
[12] C. Collange, "Stack-less simt reconvergence at low cost," HAL, Tech. Rep. hal-00622654, 2011.
[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 190–200, 2005.
[14] A. Seznec, "A new case for the TAGE branch predictor," in *Micro*, 2011.
[15] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.
[16] R. M. Russell, "The cray-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.
[17] R. Karrenberg and S. Hack, "Whole-function vectorization," in *CGO*, 2011.
[18] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanovic, "Exploring the design space of SPMD divergence management on data-parallel architectures," in *MICRO*, 2014.
[19] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanovic, "A 45nm 1.3 ghz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *ESSCIRC*, 2014.
[20] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, pp. 56–69, March 2010.
[21] G. Diamos, A. Kerr, H. Wu, S. Yalamanchili, B. Ashbaugh, and S. Maiyuran, "SIMD re-convergence at thread frontiers," in *MICRO*, 2011.
[22] J. Menon, M. De Kruijf, and K. Sankaralingam, "iGPU: exception support and speculative execution on GPUs," in *ISCA*, 2012.
[23] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware," *ACM TACO*, vol. 6, pp. 7:1–7:37, July 2009.
[24] N. Brunie, C. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained GPU performance," in *ISCA*, 2012.
[25] A. Lashgar, A. Khonsari, and A. Baniasadi, "HARP: Harnessing inactive threads in many-core processors," *ACM TECS*, vol. 13, no. 3s, 2014.
[26] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong, "Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors," in *MICRO*, 2010.
[27] M. Mckeown, J. Balkind, and D. Wentzlaff, "Execution drafting: Energy efficiency through computation deduplication," in *MICRO*, 2014.
[28] C. Collange, D. Defour, and Y. Zhang, "Dynamic detection of uniform and affine vectors in GPGPU computations," in *Europar HPPC*, 2009.