DITVA: Dynamic Inter-Thread Vectorization Architecture

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

DITVA: Dynamic Inter-Thread Vectorization Architecture

Sajith Kalathingal[a], Caroline Collange[b], Bharath N. Swamy[a], André Seznec[b]

[a]*Intel Corporation*
[b]*Inria / Univ Rennes, CNRS, IRISA*

Abstract

In the Single-Program Multiple-Data (SPMD) programming model, threads of an application exhibit very similar control flows and often execute the same instructions, but on different data. In this paper, we propose the Dynamic Inter-thread Vectorization Architecture (DITVA) to leverage the implicit Data Level Parallelism that exists across threads on SPMD applications.

By assembling dynamic vector instructions at runtime, DITVA extends an in-order SMT processor with a dynamic inter-thread vector execution mode akin to the Single-Instruction, Multiple-Thread model of Graphics Processing Units. In this mode, multiple scalar threads running in lockstep share a single instruction stream and their respective instruction instances are aggregated into SIMD instructions. DITVA can leverage existing SIMD units and maintains binary compatibility with existing CPU architectures. To balance thread- and data-level parallelism, threads are statically grouped into fixed-size independently scheduled *warps*. Additionally, to maximize dynamic vectorization opportunities, we adapt the fetch steering policy to favor thread synchronization within warps and thus improve lockstep execution.

Our experimental evaluation of the DITVA architecture on the SPMD applications from the PARSEC and Rodinia OpenMP benchmarks show that a 4-warp × 4-lane 4-issue DITVA architecture with a realistic bank-interleaved cache achieves 1.55× higher performance compared to a 4-thread 4-issue SMT architecture with AVX instructions, while fetching and issuing 51% fewer instructions, and achieving an overall 24% energy reduction. DITVA also enables applications limited by memory to scale with higher bandwidth architectures. For instance, when the bandwidth is increased from 2GB/s to 16GB/s, we find that memory bound applications show an improvement in performance by 3× in comparison with the baseline SMT. Therefore, DITVA appears as a cost-effective design for achieving very high single-core performance on SPMD parallel sections.

Keywords: Simultaneous Multi-Threading, Single instruction multiple data, Single programmultiple data, Vectorization

1. 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 were introduced to leverage multi-issue superscalar processors on parallel or multi-program workloads to achieve high single-core throughput whenever the workload features parallelism or concurrency [2, 3]. 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 statically into independent so-called *warps*. We borrow the warp concept from the Graphics Processing Unit (GPU) literature, as a group of threads that are expected to share similar control and data flows. Dynamic vectorization does not require additional programmer effort or algorithm changes to the existing SPMD applications. DITVA preserves binary compatibility with existing general purpose CPU architectures 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 [4, 5] 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. For memory bound applications, DITVA achieves a speed-up of upto 3$\times$ for higher bandwidth architectures, in comparison to a baseline SMT with the same bandwidth. 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 2, and describe the DITVA architecture in Section 4. Section 5 evaluates performance and design tradeoffs, while Section 6 examines hardware complexity implications and evaluates power consumption. Section 3 reviews some related works.

2. Motivation

In this section, we first motivate our choice of building an SPMD oriented throughput processor on top of an in-order SMT processor. Then we argue that SPMD programs offer tremendous opportunities to share a significant part of the instruction execution of the different scalar threads.

2.1. SMT architectures

SMT architectures were introduced to exploit thread-level and/or multi-program level parallelism to optimize the throughput of a superscalar core [2]. Typically, on an SMT processor, instructions from different hardware threads progress concurrently in all stages of the pipeline. Depending on the precise implementation, some pipeline stages only handle instructions from a single thread at a given cycle. For instance, the instruction fetch pipeline stage may be time-multiplexed [6], while the execution stage may mix instructions from all threads.

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 [7].

SMT architectures often target both high single-thread performance and highly parallel or multi-program performance. As a consequence, most commercial designs have been implemented with 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 [8]. Therefore, in-order SMT appears as a good architecture tradeoff for implementing the cores of an SPMD oriented throughput processor.

2.2. Instruction redundancy across SPMD threads

In SPMD applications, threads usually execute very similar flows of instructions. They exhibit some control flow divergence due to branches, but generally a rapid convergence of the control flows occur. To illustrate this convergence/divergence scenario among the parallel sections, we display a control flow diagram from the *Blackscholes* workload [4] in figure 1. All the threads execute the convergent blocks while only some threads execute the divergent blocks. Moreover, in the case of divergent blocks, more than one thread often executes the divergent block.

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 favoring latency hiding and avoiding starvation. On an SMT processor, each thread manages its own control flow. In the example illustrated above, for each convergent block, instructions

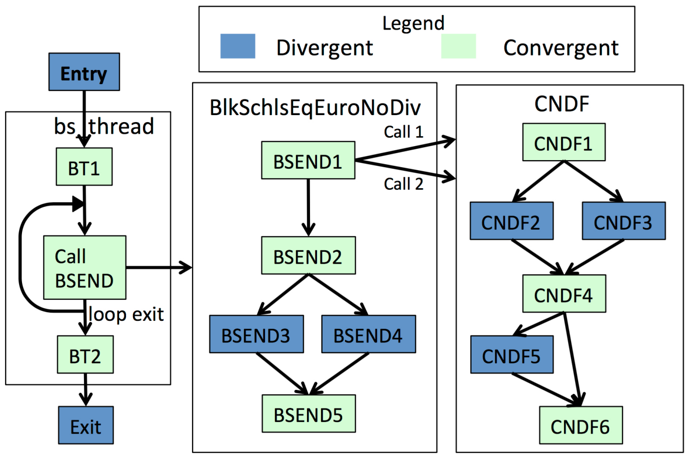


Figure 1: Control flow graph of blackscholes benchmark

are fetched, decoded, etc. by each of the threads without any sharing of this redundant effort. The same applies to the divergent blocks that are executed by more than one thread. This may cause a large waste of resources.

We study how running threads in lockstep may favor instruction sharing between threads. We synchronize threads in SPMD applications by groups of 4 whenever they follow the same control flow. We use the Min(SP:PC) scheduling policy that we will describe further in Section 4.3 to favor thread synchronization after control flow divergence. Figure 2 shows the number of instructions that can be shared among groups of 4 threads. This figure represents the DLP that could be extracted from the SPMD program.

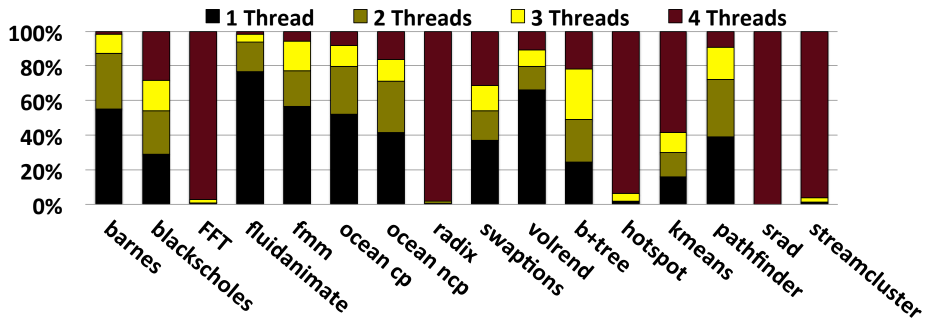


Figure 2: Breakdown of average combined instructions among groups of 4 threads

These results are in line with prior studies that 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 [9]. We leverage this instruction redundancy to mutualize the front-end pipeline of an in-order SMT processor, as a resource-efficient way to improve throughput on SPMD applications.

3. Related work

DITVA aims at exploiting data-level parallelism in SPMD applications by extending an in-order SMT processor. Therefore, DITVA is strongly related to the three domains, SIMD/vector architecture, SIMT also known as GPU architectures and SMT architectures.

3.1. SIMD and/or vectors

Static DLP, detected on the source code, has been exploited by hardware and compilers for decades. SIMD and/or vector execution have been considered as early as the 1970s in vector supercomputers [10]. 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 [11, 12]. However, the data parallelism is constrained by the width of vector or SIMD instructions. 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 [13]. 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.

3.2. 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 [14]. 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. SIMT architectures are highly sensitive to divergence as they serialize divergent branch paths. DITVA maintains SMT-level performance on divergent code by leveraging the TLP between multiple divergent threads in the same warp. Various works extend the SIMT model to support more generic code [15, 16] or more flexible execution [17, 18, 19]. However, they all target applications specifically written for GPUs, rather than general-purpose parallel applications.

3.3. Instruction redundancy in SMT

SMT improves the throughput of a superscalar core by enabling independent threads to share CPU resources dynamically. Resource sharing policies have huge impact on execution throughput [2, 20, 21, 22, 23, 24]. Many studies have focused on optimizing the instruction fetch policy and leaving the instruction core unchanged while other studies have pointed out the ability to benefit from memory level parallelism through resource sharing policies. However, these resource sharing heuristics essentially address multi-program workloads.

3.4. Thread reconvergence for SPMD applications

Similarity exploitation in SPMD applications requires the threads to execute similar instructions at around the same time. Thread synchronization is not difficult to achieve in a regular application when the threads are on a convergent path. With divergence, threads start to follow different execution paths. Divergence is the source of performance degradation in SIMD and SIMT architectures. An optimal solution to maximize thread synchronization should support early reconvergence. It should perfectly identify the best reconvergence point. For certain programs, identifying reconvergence point is difficult. An example of such a progam is shown in listing 1. When the program is executed, the threads may execute either function A() or function B(). After the divergence, it is difficult to know the point at which the threads will reconverge.

Listing 1: A program with divergence

```
for(i = 0 to n)
  if((i+tid) % c))
    A();
  else
    B();
```

Thread reconvergence is a challenging task because of its dynamic nature. In general, there are two classes of reconvergence mechanisms. The first one is stack-based and the second one is stack-less. A bulk of SIMT architectures uses an explicit reconvergence mechanism based on explicit annotations from ISA and a hardware stack. A stack-based approach uses a stack to keep track of divergences within a thread group (warp). There are several proposals for implicit reconvergence in SIMT architectures that are still stack-based. DITVA uses a stack-less implicit mechanism, which prioritizes threads, to maximize convergent execution of an SPMD application. In this section, we will discuss some of the methods used for thread synchronization.

A simple stack-less heuristics prioritizes the threads based on *textually earliest program point* in the source code [25]. Reconvergence happens when the threads share the same PC. It builds on the idea that compilers typically lay out basic blocks in memory in a way that preserves dominance relations: reconvergence points have a higher address than the matching divergence points. The earliest-first policy can be applied directly to the binary using instruction addresses to determine instruction order. We name this heuristic *MinPC*. However, MinPC may fail if the compiler heavily re-orders the basic blocks. As the MinPC heuristic may rely on program counters only, it does not require compiler hints.

MinSP-PC [26] heuristic improved MinPC by giving priority to the block with minimum relative stack pointer (SP) value. On a match, the priority is given for MinPC. The underlying assumption is that the stack size increases with the function call depth, resulting in a lower value of stack pointer in a downward growing stack. Therefore, the threads with highest call depth are given top priority. MinSP-PC was shown to double

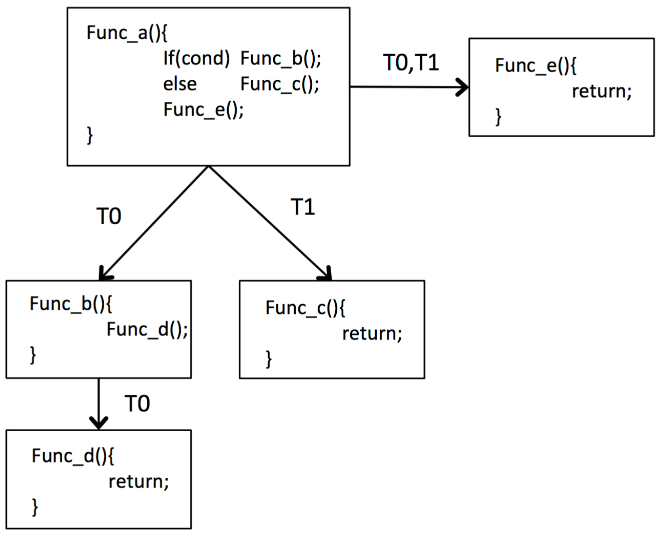


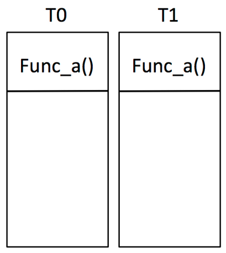
Figure 3: Call graph example

the number of instruction combining opportunities on average compared to MinPC, on HPC-oriented SPMD applications [9].

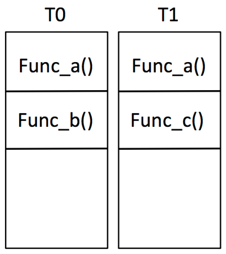
Let us consider threads T0 and T1 executing the code blocks shown in figure 3. In the example, T0 and T1 starts at Func_a and diverges at the end of the function call. T0 proceeds to execute Func_b and Func_d. Later T0 start the execution of Func_e. Similarly, after divergence, T1 executes Func_c and later execute Func_e. Figure 4 shows the growth of stack with MinSP-PC heuristics. Initially, both threads start at Func_a, and a stack entry is created as shown in 4(a). T0 and T1 share the same stack offset and hence the heuristic uses MinPC policy, which keeps them in lockstep mode until the end of the function call. After divergence, an entry for Func_b and Func_c is created in the stack of T0 and T1 respectively. After divergence, the threads are prioritized with MinSP-PC policy. Assuming that T0 had the highest priority, T0 will start the execution of Func_d. After the execution of Func_b, Func_d and Func_c by T0 and T1 the stack entries are popped out. The next synchronization point for T0 and T1 is at the beginning of the function call Func_e, where both the threads have same SP offset and MinPC policy will ensure lockstep execution in Func_e. A drawback of the heuristic is its inability to synchronize the same functions arrived through a different path. Figure 5 shows a control flow of a program with synchronization point at Func_d. Since T0 and T1 were in a divergent path after Func_a, MinSP-PC heuristic will miss the synchronization point at Func_d and will synchronize on Func_e instead.

3.5. General purpose architectures exploiting inter-thread redundancy

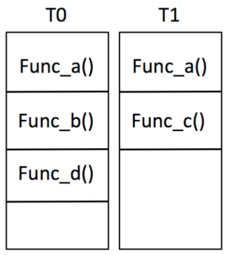
In the past, several attempts were made to leverage this redundancy to optimize the performance of SPMD applications. Most of the previous work focuses on instruction



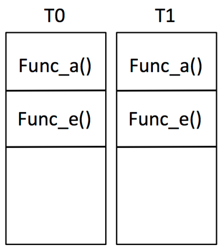
(a) Initial call at func_a()



(b) Divergence - T0 takes Func_b() and Func_c() path



(c) T0 calls Func_d()



(d) T0 and T1 calls Func_e()

Figure 4: Growth of stack in time for threads T0 and T1

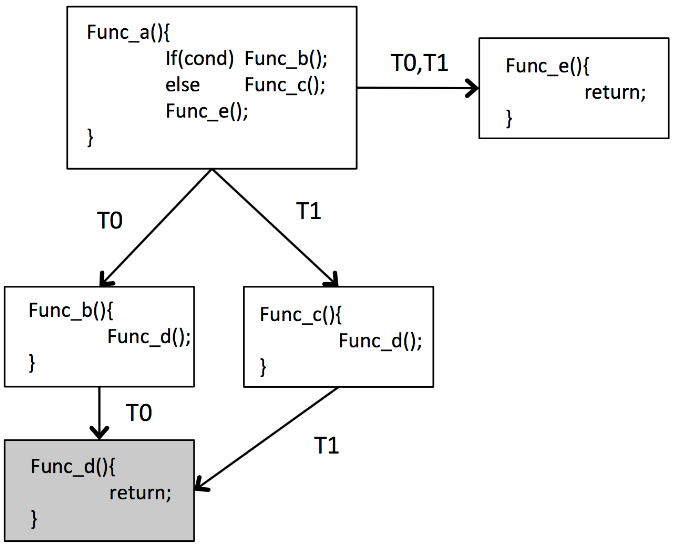

```
Func_a(){
        If(cond)  Func_b();
        else      Func_c();
        Func_e();
}
```
```
Func_e(){
        return;
}
```
```
Func_b(){
        Func_d();
}
```
```
Func_c(){
        Func_d();
}
```
```
Func_d(){
        return;
}
```

Figure 5: Call graph with synchronization point at Func_d

redundancy. Some works propose to re-use the result from a previous execution of instruction with identical data to eliminate data/value redundancy.

Kumar et al [27] proposed *Fetch combining* for their Conjoined-core Chip Multiprocessing. A Conjoined-core shares resources between adjacent cores. *Fetch combining* improves the fetch bandwidth when two threads running on conjoined-core pair have the same PC. In this case, a fetch is consumed by both the cores.

Thread Fusion [28] fuses instructions across threads in a 2-way SMT when both threads are executing identical instructions. A fused instruction only has one instance in the pipeline front-end. Within the pipeline back-end, the fused instruction is split back into two separate instructions that execute independently. Thread fusion require both threads to run in lock-step (i.e. Threads execute the same instruction at the same time). When the threads are not in lock-step, they are executed in *Normal mode*. Thread fusion switches to *Fused mode* with the help of synchronization points inserted by the compiler. Synchronization points are defined as the first PC (Program Counter) of an instruction that is frequently visited by the threads executing a parallel section. Thread fusion is an optimization technique focused on reducing the energy consumption (it does not focus on improving the performance).

Minimal Multi-threading [29] improves it further by achieving this without compiler support. MMT does this by using a Fetch History Buffer(FHB), which keeps track of the fetch history(PC) at a branch for each thread. For a branch instruction, it also checks if a PC is found in other thread's history then the thread will be transitioning to the *CATCHUP* mode, and it is given higher priority until it is re-synchronized. MMT tries to favor thread synchronization in the front-end (instruction fetch and decode) of an SMT core. It further tries to eliminate redundant computation on the threads. However, MMT assumes a conventional out-of-order execution superscalar core and does not attempt to synchronize instructions within the backend.

Multi-threaded Instruction Sharing(MIS) [30] uses the instruction similarity and retires identical instructions without executing them. MIS uses a *match table*, which holds the results of a previous instruction. MIS performs *match test* on other threads in parallel. If there is a hit in the *match table*, the instruction from the current thread is retired without execution.

Execution Drafting [31] focuses on the energy efficiency of an in-order core processor by *drafting* duplicate instructions. *Drafting* is a technique in which subsequent duplicate instructions follow the first instruction in the pipeline. Execution drafting support instructions from multiple programs as well, unlike other techniques which mostly focus on a multi-threaded program. A duplicate instruction can be either partial or full duplicate. A partial duplicate instruction is one which has the same opcode but has different machine code. Execution drafting uses a Hybrid Thread Synchronization Method(HTSM) which is a combination of MinPC [25] and random thread synchronization method. The use of heuristics may result in increased latency. Execution drafting primarily focuses on applications in data centres, which are often multiple instances of the same program or the applications that have latency tolerance. Execution Drafting [31] seeks to synchronize threads running the same code and shares the instruction control logic to improve energy efficiency. It targets both multi-thread and multi-process applications by allowing lockstep execution at arbitrary addresses.

Both MMT and Execution Drafting attempt to run all threads together in lockstep as much as possible. However, full lockstep execution is not always desirable as it defeats the latency tolerance purpose of SMT. The threads running in lockstep will all stall at the same time upon encountering a pipeline hazard like a cache miss, causing inefficient resource utilization.

3.6. GPU architectures to exploit inter-thread redundancies

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 reconvergence information to the hardware. GPU compilers have to emit explicit instructions to mark reconvergence points in the binary program. These mechanisms are designed to handle user-level code with a limited range of control-flow constructs. The stack-based divergence tracking mechanism does not support exceptions or interruptions, which prevents its use with a general-purpose system software stack. Various works extend the SIMT model to support more generic code [15, 16] or more flexible execution [17, 18, 19]. However, they all target applications specifically written for GPUs, rather than general-purpose parallel applications.

4. The Dynamic Inter-Thread Vectorization Architecture

In this section, we present the Dynamic Inter-Thread Vectorization Architecture (DITVA).

The classic Flynn's taxonomy classifies parallel architectures among Single Instruction stream, Single Data stream (SISD), Single Instruction steam, Multiple Data streams (SIMD) and Multiple Instruction streams, Multiple Data streams (MIMD) [32]. Transposed in modern terms, an *instruction stream* is generally assumed to be a hardware thread. However, such strict 1-to-1 mapping between threads and instruction streams is not necessary, and we can decouple the notion of instruction stream from the notion of 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.

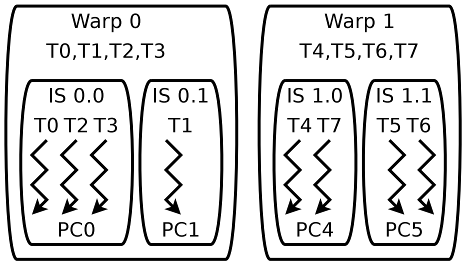


Figure 6: Logical organization of hardware threads on 2-warp× 4-thread DITVA. Scalar threads are grouped into 2 warps of 4 threads. Scalar threads sharing the same PC within a warp form an Instruction Stream (IS).

Logical thread organization. DITVA supports a number of hardware thread contexts, that 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 6, 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 [33] in the GPU architecture literature. While thread-to-warp assignment is static, 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 6, 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 4.3).

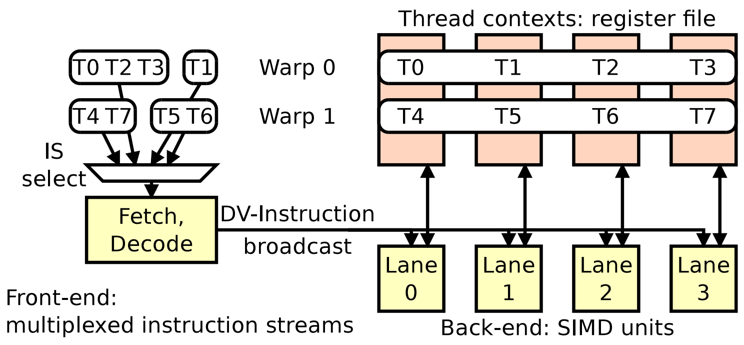


Figure 7: Thread mapping to the physical organization.

Mapping to physical resources. DITVA consists in a front-end that processes Instruction Streams and a SIMD back-end (Figure 7).

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 are instructions that already operate on vectors, such as SSE and AVX, that are executed in multiple waves over the whole SIMD width.

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. We will refer to the group of registers Ri from the set of hardware contexts in a DITVA warp as the DV-register DRi, and the group of a replicated functional unit as a DV functional unit.

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

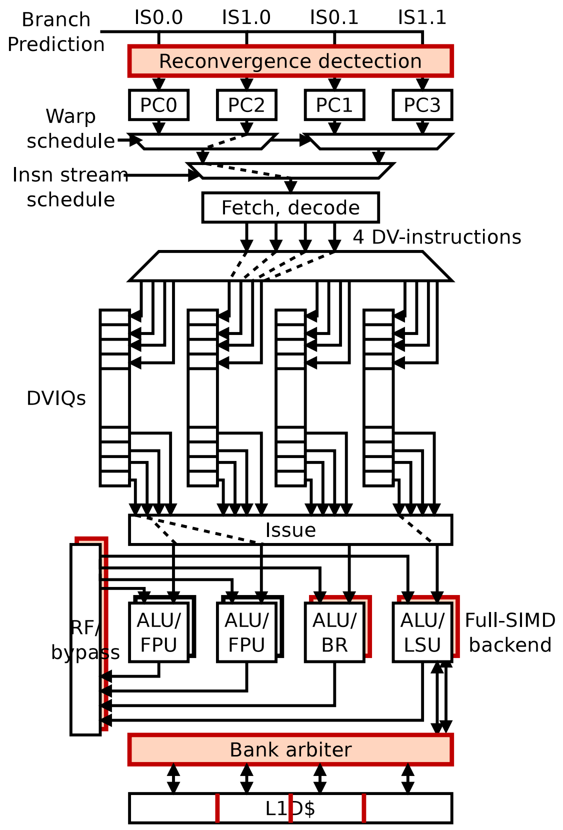


Figure 8: Overview of a $2W \times 2T$, 4-issue DITVA pipeline. Main changes from SMT are highlighted.

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.

4.1. Pipeline architecture

We describe the stages of the DITVA pipeline, as illustrated in Figure 8.

4.1.1. 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.

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

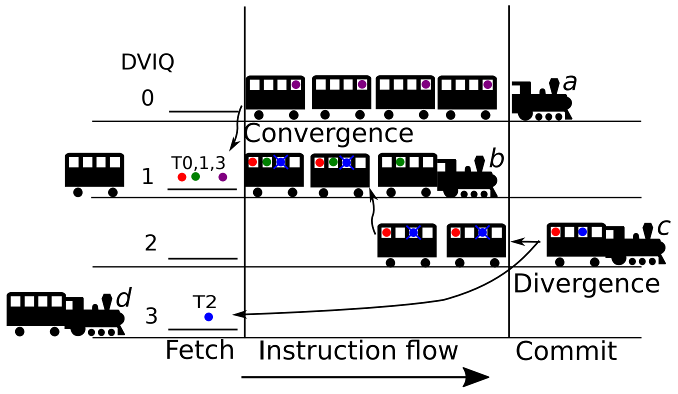


Figure 9: Instruction stream tracking in DITVA. Instruction Streams (a, b, c, d) are illustrated as trains, DVIQs as tracks, DV-instructions as train cars, and scalar instructions as passengers.

match between IS$_i$ and IS$_j$ indicates they have reached a point of convergence 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 9 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 is also shown 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. Figure 9 illustrates IS *c* diverging at commit time to spawn the new IS *d* containing thread 2. As divergence is only detected late in the pipeline, thread 2 initially follows speculatively the IS *c* in DVIQ 2. After divergence is detected and thread 2 is determined to belong to IS *d*, the speculative operations corresponding to thread 2 in IS *c* are masked out so they have no architectural effect. Divergence will be described in further details in Section 4.1.5.

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 [2, 24]. Then, an intra-warp instruction fetch steering policy selects one IS within the selected warp. The specific policy will be described in Section 4.3. 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, the 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 9, DVIQ 2 contains instructions of threads T0 and T2, while IS 2 has no active threads. The DV-instruction mask avoids this ambiguity.

4.1.2. 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.

The length of sequence numbers should be dimensioned in such a way that there is no possible ambiguity in comparing two sequence numbers. The ambiguity is avoided by using more sequence numbers than the maximum number of instructions belonging to a given thread in all DVIQs, which are 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 cannot be launched before 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.

4.1.3. 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 10(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 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.

4.1.4. 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 10(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 10(c) shows the pipelined execution of a SSE DV-instruction with mask 1011 in a 4W×4T DITVA. In figure 10(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 over SIMD instructions of inactive threads, as in vector processors. Figure 10(d) shows the execution of a AVX DV-instruction with mask 1101 in a $4W \times 4T$ DITVA.

4.1.5. 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 9, IS*c* with threads T0 and T2 is split to form the new IS*d* with thread T2. Instructions of thread T2 are invalidated within the older DV-instructions of IS*c* as well as IS*b*. 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 9, they correspond to disabling thread T2 in the DV-instructions from IS$_2$ and IS$_1$.

In addition, some bookkeeping is needed. In the case of a true branch misprediction, 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 kept simple as null DV-instructions 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 a misprediction. The only case when all IS$_s$ slots are occupied is when each IS has only one thread. In that case, a misprediction 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.

4.2. Data memory accesses

A data access operation in a DV-instruction may have to 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 use a fully hashed set index to reduce bank conflicts, assuming a virtually indexed L1 data cache. Our experiments in Section 5 illustrate the reduction in the number of data access conflicts due to the alignment of the bottom of thread stacks on page boundaries.

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. Our simulations in Section 5 show that this optimization significantly decreases the total number of TLB misses or allows to use smaller TLBs.

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.

4.3. 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 [26]. 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. However, while experience shows that MinSP-PC tends to synchronize SPMD threads in many cases, there is no guarantee that each thread will make continuous forward progress. MinSP-PC could even lead to deadlocks, e.g. in the event of an active waiting loop. Besides, when going through non-SPMD code sections involving independent threads, applying the MinSP-PC policy would essentially result in executing a single thread, while stalling all other threads.

Therefore, for this study on DITVA, we use 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[1], 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.

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

[1]RR/MinSP-PC is not completely fair among independent threads, e.g. multiple program workloads as it may favor some threads. However, fairness on this type of workloads is out of the scope of this paper.

Table 1: 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 [35]
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

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.

5. Evaluation

We simulate DITVA to evaluate its performance and design tradeoffs.

5.1. 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 [34] 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

Table 2: Rodinia Applications

Application	Problem size
B+tree	1 million keys
Hotspot	4096×4096 data points
Kmeans	10000 datapoints, 34 features
Pathfinder	100000 width, 100 steps
SRAD	2048×2048 datapoints
Streamcluster	4096 points, 32 dimensions

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 1 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 [4] and Rodinia [5] 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. The simulation parameters that we used for Rodinia benchmarks are shown in Table 2.

Figure 11 shows the speedup 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 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.

5.2. Throughput

Figure 12 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. The $4W \times 4T$ DITVA also achieves 34% speedup over 16-thread SMT. The $2W \times 8T$ DITVA achieves 46% speedup 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.

Due to memory hierarchy related factors, the actual speed-up is not proportional to DV-instruction occupancy. For instance, the performance of *Radix* drops with higher thread counts due to reduced cache locality. The speedup of DITVA over SMT just compensates this performance loss. The scaling of *Hotspot* and *Srad* is likewise limited by the memory related factors. The performance of DITVA on applications with low DLP, like *Fluidanimate*, is on par with 16-thread SMT. *Fluidanimate*, *Ocean CP*, *Ocean NCP* and *Volrend* show sub-linear parallel scaling: the total instruction count increases with thread count, due to extra initialization, bookkeeping and control logic. Still, DITVA enables extra performance gains for a given thread count.

5.3. Divergence and mispredictions

Figure 13 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.

5.4. Impact of split data TLB

However, as pointed out in Section 4.2, there is no need to maintain equal contents for the TLBs of the distinct lanes. Assuming a 4KB page size, Figure 14 illustrates the TLB miss rates for different configurations: 4-lanes DITVA, i.e., a total of 16 threads, with 128-entry unified TLB, 256-entry unified TLB and 64-entry split TLB, and a 64-entry TLB for the SMT configuration.

On our set of benchmarks, 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.

5.5. L1 cache bank conflict reduction

Straightforward bank interleaving using the low order bits on the L1 data cache leads to mild to severe bank conflicts, as illustrated in Figure 15. We find that many 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 [36, 9].

To reduce such bank conflicts for DV-loads and DV-stores, we use a hashed set index as introduced in Section 4.2. 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. Figure 15 illustrates that such a hashing mechanism is effective in reducing bank conflicts on applications where threads make independent sequential memory accesses, such as *Blackscholes* and *FFT*. Most other applications also benefit from hashing. Bank conflicts increase with hashing on *B+tree* and *Kmeans*. However, these applications have few conflicts in either configuration. In the remainder of the evaluation section, this hashed set index is used.

5.6. Impact of memory bandwidth on memory intensive applications

In the multi-core era, memory bandwidth is a bottleneck for the overall core performance. Our simulations assume 2 GB/s DRAM bandwidth per core. To analyze the impact of DRAM bandwidth on memory intensive applications running on DITVA, we simulate configurations with 16 GB/s DRAM bandwidth per core which is a feasible alternative using high-end memory technologies like HBM [37]. The performance scaling of 16 GB/s relative to 4-thread SMT with 2 GB/s DRAM bandwidth is illustrated in Figure 16.

For many benchmarks, 2 GB/s bandwidth is sufficient. However, as discussed in Section 5.1, the performance of *Srad*, *Hotspot*, *Ocean*, *Radix* and *FFT* is bound by memory throughput. DITVA enables these applications to benefit from the extra memory bandwidth to scale further, widening the gap with the baseline SMT configuration.

6. Hardware Overhead, Power and Energy

DITVA induces extra hardware complexity and area as well as extra power supply demand over an in-order SMT core. On the other hand, DITVA achieves higher performance on SPMD code. This can lead to reduced total energy consumption on such code.

We analyze qualitatively and quantitatively the sources of hardware complexity, power demand and energy consumption throughout the DITVA pipeline compared with the ones of the corresponding in-order SMT core.

6.1. Qualitative evaluation

Pipeline Front End. The modifications in the pipeline front-end induce essentially extra logic, e.g. comparators and logic to detect IS convergence, the logic to select the IS within the warp, and the DVIQ mask unsetting logic for managing branch mispredictions and exceptions. The extra complexity and power consumption should remain relatively limited. The most power hungry logic piece introduced by the DITVA architecture is the scoreboard that must track the dependencies among registers of up to *m* ISs per warp. However, this scoreboard is also banked since there are no inter-thread register dependencies.

On the other hand, DITVA significantly cuts down dynamic energy consumption in the front-end. Our experiments show a reduction of 51% of instruction fetches for $4W \times 4T$ DITVA.

Memory unit. The DITVA memory unit requires extra hardware. First, bank conflict handling logic is needed, as we consider an interleaved cache. Then, replicated data TLBs add an overhead in area and static energy. Moreover as DITVA executes more threads in parallel than an SMT core, the overall capacity of the TLB must be increased to support these threads. However, as TLB contents do not have to be maintained equal, we have shown that lane TLBs with the same number of entries as a conventional 4-way SMT core would be performance effective. Therefore, on DITVA, the TLB silicon area as well as its static energy consumption is proportional to the number of lanes.

Register file. An in-order n-thread SMT core features $n \times$ NbISA scalar registers of width B bits while a $nW \times mT$ DITVA features $n \times$ NbISA DV-registers of width $m \times B$ bits. Estimations using CACTI [38] and McPAT [39] indicate that the access time and the dynamic energy per accessed word are in the same range for DITVA and the SMT. The register file silicon area is nearly proportional to m, the number of lanes, and so is its static leakage.

Execution units. The widening of the two scalar functional units into DV-units constitutes the most significant hardware area overhead. The SIMD DV-units have a higher leakage and require higher instantaneous power supply than their scalar counterparts. However, DITVA also leverages the existing AVX SIMD units by reusing them as DV-units. Additionally, since DV-units are activated through the DV-instruction mask, the number of dynamic activations of each functional type is about the same for DITVA and the in-order SMT core on a given workload, and so is the dynamic energy.

Non-SPMD workloads. DITVA only benefits shared memory SPMD applications that have intrinsic DLP. On single-threaded workloads or highly divergent SPMD workloads, DITVA performs on par with the baseline 4-way in-order SMT processor. Workloads that do not benefit from DITVA will mostly suffer from the static power overhead of unused units. Moreover, on single-thread workloads or on multiprogrammed workloads, a smart runtime system could be used to power down the extra execution lanes thus bringing the energy consumption close to the one of the baseline SMT processor.

Non-SPMD multi-threaded workloads may suffer scheduling unbalance (unfairness) due to the RR/MinSP-PC fetch policy. However, this unbalance is limited by the hybrid fetch policy design. When all threads run independently, a single thread will get a priority boost and progress twice as fast as each of the other threads. e.g. with 4 threads, the MinSP-PC thread gets 2/5th of the fetch bandwidth, each other thread gets 1/5th.

6.2. Quantitative evaluation

Dynamic vectorization reduces the number of DV-instructions over original instructions. Figure 17 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. DV-instruction count is low for applications *Radix*, *FFT*, *Hotspot*, *Srad* and *Streamcluster*, which have nearly perfect dynamic vectorization. However, the DV-instruction count reduction

Table 3: Area and static power McPAT estimates.

Component	4T SMT		4W×2T DITVA		4W×4T DITVA	
	Area (mm^2)	Static P. (W)	Area (mm^2)	Static P. (W)	Area (mm^2)	Static P. (W)
Front-end	3.46	0.140	3.63	0.149	4.14	0.175
LSU	1.32	0.050	2.21	0.041	2.33	0.054
MMU	0.22	0.009	0.32	0.012	0.50	0.018
Execute	20.98	0.842	21.51	0.868	22.40	0.920
Core total	35.50	1.815	37.30	1.868	39.09	2.001

in *Volrend*, *Fluidanimate* and *Ocean* is compensated by the parallelization overhead caused by the thread count increase.

We modeled a baseline SMT processor and DITVA within McPAT [39]. It assumes a 2 GHz clock in 45nm technology with power gating. We modeled two alternative designs. The first one is the configuration depicted on Table 1, except the cache that was modeled as 64 KB 8-way as we could not model the banked 32 KB 16-way configuration in McPAT. The dynamic energy consumption modeling is illustrated on Figure 18 while modeled silicon area and static energy are reported in Table 3. As in Section 5, 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.

Note that McPAT models execution units as independent ALUs and FPUs, rather than as SIMD blocks as implemented on current architectures. Also, estimations may tend to underestimate front-end energy [40]. Thus, the front-end energy savings are conservative, while the overhead of the back-end is a worst-case estimate. Despite these conservative assumptions, Figure 18 and Table 3 show that DITVA appears as an energy-effective solution for SPMD applications with 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 run-time (Figure 12) and a reduction in the number of fetched instructions, mitigated by an increase in static power from the wider execution units.

7. Conclusion

In this paper, we have proposed the DITVA architecture that aims at partially filling the gap between massively threaded machines, e.g. SIMT GPUs, and SMT general-purpose CPUs. Compared with an in-order SMT core architecture, DITVA achieves high throughput on the parallel sections of the SPMD applications by extracting dynamic data-level parallelism at runtime. DITVA provides a design tradeoff between an in-order SMT core and an SIMT GPU core. It vectorizes instructions dynamically across threads like SIMT GPUs, but retains binary compatibility with general-purpose CPUs. Applications require no source modification nor re-compilation.

DITVA group threads statically into fixed-size warps. SPMD threads from a warp are dynamically vectorized at instruction fetch time. The instructions from the different threads are grouped together to share an instruction stream whenever their PC are equal. Then the group of instructions (the DV-instruction) progresses in the pipeline as a unit. This allows to mutualize the instruction front-end as well as the overall instruction control. The instructions from the different threads in a DV-instruction are executed on replicated execution lanes. 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 a DITVA architecture 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 to execute SPMD code. Therefore, DITVA appears as a cost-effective design for achieving very high single-core performance on SPMD parallel sections. A DITVA-based multi-core or many-core would achieve very high parallel performance.

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 [17] or Simultaneous Branch Interweaving [18], Dynamic Warp Subdivision [33] could improve latency tolerance by allowing threads to diverge on partial cache misses, and Dynamic Scalarization [41] could further unify redundant data-flow across threads. The design of fetch steering policies that best balance SIMD utilization with fairness is still an open research problem. Adaptive hybrid policies could enable both high SIMD utilization on data-parallel code and good fairness when running independent threads.

Acknowledgements

This work was supported by the ERC DAL project of the EU FP7 program.

References

References

[1] S. Kalathingal, C. Collange, B. Narasimha Swamy, A. Seznec, Dynamic inter-thread vectorization architecture: extracting DLP from TLP, in: International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD), Los Angeles, United States, 2016.
URL `https://hal.inria.fr/hal-01356202`

[2] D. M. Tullsen, S. J. Eggers, H. M. Levy, Simultaneous multithreading: Maximizing on-chip parallelism, in: Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995, 1995, pp. 392–403. `doi:10.1145/223982.224449`.
URL `http://doi.acm.org/10.1145/223982.224449`

[3] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor, in: ACM SIGARCH Computer Architecture News, Vol. 24, ACM, 1996, pp. 191–202.
URL `http://dl.acm.org/citation.cfm?id=232993`

[4] C. Bienia, S. Kumar, J. P. Singh, K. Li, The parsec benchmark suite: Characterization and architectural implications, in: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, ACM, 2008, pp. 72–81.
URL `http://dl.acm.org/citation.cfm?id=1454128`

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, IEEE, 2009, pp. 44–54.

[6] A. Seznec, S. Felix, V. Krishnan, Y. Sazeides, Design tradeoffs for the alpha EV8 conditional branch predictor, in: 29th International Symposium on Computer Architecture (ISCA 2002), 25-29 May 2002, Anchorage, AK, USA, 2002, pp. 295–306.
URL `http://computer.org/proceedings/isca/1605/16050295abs.htm`

[7] S. Hily, A. Seznec, Branch prediction and simultaneous multithreading, in: Proceedings of the Fifth International Conference on Parallel Architectures and Compilation Techniques, PACT'96, Boston, MA, USA, October 20-23, 1996, 1996, pp. 169–173. `doi:10.1109/PACT.1996.552664`.
URL `http://dx.doi.org/10.1109/PACT.1996.552664`

[8] S. Hily, A. Seznec, Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading, in: Proceedings of the Fifth International Symposium on High-Performance Computer Architecture, Orlando, FL, USA, January 9-12, 1999, 1999, pp. 64–67. `doi:10.1109/HPCA.1999.744331`.
URL `http://dx.doi.org/10.1109/HPCA.1999.744331`

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

[10] R. M. Russell, The cray-1 computer system, Commun. ACM 21 (1) (1978) 63–72. `doi:10.1145/359327.359336`.
URL `http://doi.acm.org/10.1145/359327.359336`

[11] R. Karrenberg, S. Hack, Whole-function vectorization, in: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, IEEE Computer Society, 2011, pp. 141–150.

[12] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, K. Asanovic, Exploring the design space of SPMD divergence management on data-parallel architectures, in: 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2014, IEEE, 2014, pp. 101–113.

[13] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, K. Asanovic, A 45nm 1.3 ghz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators, in: ESSCIRC 2014 – 40th European Solid State Circuits Conference, IEEE, 2014, pp. 199–202.

[14] J. Nickolls, W. J. Dally, The GPU computing era, IEEE Micro 30 (2010) 56–69.
URL `http://dx.doi.org/10.1109/MM.2010.41`

[15] G. Diamos, A. Kerr, H. Wu, S. Yalamanchili, B. Ashbaugh, S. Maiyuran, SIMD re-convergence at thread frontiers, in: MICRO 44: Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture, 2011.

[16] J. Menon, M. De Kruijf, K. Sankaralingam, iGPU: exception support and speculative execution on GPUs, in: ACM SIGARCH Computer Architecture News, Vol. 40, IEEE Computer Society, 2012, pp. 72–83.

[17] W. W. L. Fung, I. Sham, G. Yuan, T. M. Aamodt, Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware, ACM Trans. Archit. Code Optim. 6 (2009) 7:1–7:37.

[18] N. Brunie, C. Collange, G. Diamos, Simultaneous branch and warp interweaving for sustained GPU performance, in: ACM SIGARCH Computer Architecture News, Vol. 40, IEEE Computer Society, 2012, pp. 49–60.

[19] A. Lashgar, A. Khonsari, A. Baniasadi, HARP: Harnessing inactive threads in many-core processors, ACM Transactions on Embedded Computing Systems (TECS) 13 (3s) (2014) 114.

[20] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor, in: Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, USA, May 22-24, 1996, 1996, pp. 191–202. `doi:10.1145/232973.232993`.
URL `http://doi.acm.org/10.1145/232973.232993`

[21] F. J. Cazorla, A. Ramírez, M. Valero, E. Fernández, Dynamically controlled resource allocation in SMT processors, in: 37th Annual International Symposium on Microarchitecture (MICRO-37 2004), 4-8 December 2004, Portland, OR, USA, 2004, pp. 171–182. `doi:10.1109/MICRO.2004.17`.
URL `http://dx.doi.org/10.1109/MICRO.2004.17`

[22] K. Luo, M. Franklin, S. S. Mukherjee, A. Seznec, Boosting SMT performance by speculation control, in: Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, April 23-27, 2001, 2001, p. 2. doi:10.1109/IPDPS.2001.924929.
URL http://dx.doi.org/10.1109/IPDPS.2001.924929

[23] A. El-Moursy, D. H. Albonesi, Front-end policies for improved issue efficiency in SMT processors, in: Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA'03), Anaheim, California, USA, February 8-12, 2003, 2003, pp. 31–40. doi:10.1109/HPCA.2003.1183522.
URL http://dx.doi.org/10.1109/HPCA.2003.1183522

[24] S. Eyerman, L. Eeckhout, A memory-level parallelism aware fetch policy for SMT processors, in: 13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA, 2007, pp. 240–249. doi:10.1109/HPCA.2007.346201.
URL http://dx.doi.org/10.1109/HPCA.2007.346201

[25] M. J. Quinn, P. J. Hatcher, K. C. Jourdenais, Compiling c* programs for a hypercube multicomputer, in: ACM SIGPLAN Notices, Vol. 23, ACM, 1988, pp. 57–65.
URL http://dl.acm.org/citation.cfm?id=62122

[26] C. Collange, Stack-less simt reconvergence at low cost, Tech. rep., HAL (2011).
URL http://hal.archives-ouvertes.fr/hal-00622654/

[27] R. Kumar, N. P. Jouppi, D. M. Tullsen, Conjoined-core chip multiprocessing, in: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2004, pp. 195–206.

[28] J. González, Q. Cai, P. Chaparro, G. Magklis, R. Rakvic, A. González, Thread fusion, in: Proceedings of the 2008 international symposium on Low Power Electronics & Design, ACM, 2008, pp. 363–368.

[29] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, F. T. Chong, Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors, in: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2010, pp. 337–348.
URL http://dl.acm.org/citation.cfm?id=1934980

[30] M. Dechene, E. Forbes, E. Rotenberg, Multithreaded instruction sharing, Department of Electrical and Computer Engineering, North Carolina State University, Tech. Rep (2010).

[31] M. Mckeown, J. Balkind, D. Wentzlaff, Execution drafting: Energy efficiency through computation deduplication, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2014, pp. 432–444.

[32] M. J. Flynn, Some computer organizations and their effectiveness, Computers, IEEE Transactions on 100 (9) (1972) 948–960.

[33] J. Meng, D. Tarjan, K. Skadron, Dynamic warp subdivision for integrated branch and memory divergence tolerance, SIGARCH Comput. Archit. News 38 (3) (2010) 235–246. `doi:http://doi.acm.org/10.1145/1816038.1815992`.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, ACM Sigplan Notices 40 (6) (2005) 190–200.

[35] A. Seznec, A new case for the TAGE branch predictor, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2011, pp. 117–127.

[36] J. Meng, K. Skadron, Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling, in: IEEE International Conference on Computer Design (ICCD) 2009, IEEE, 2009, pp. 282–288.

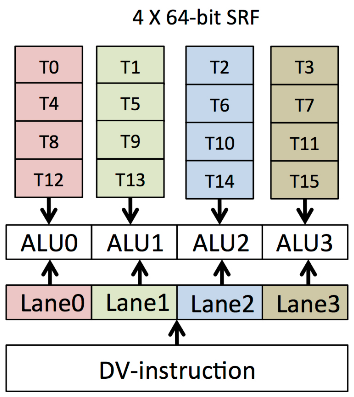
[37] M. O'Connor, Highlights of the High-Bandwidth Memory (HBM) standard, in: Memory Forum Workshop, 2014.

[38] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, N. P. Jouppi, Cacti 5.1, Tech. rep., HP Laboratories (2008).

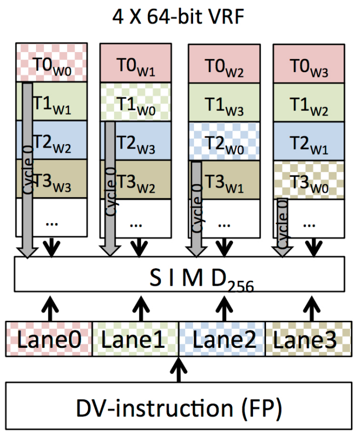
[39] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, N. P. Jouppi, Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures, in: 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009. MICRO-42., IEEE, 2009, pp. 469–480.

[40] S. L. Xi, H. M. Jacobson, P. Bose, G. Wei, D. M. Brooks, Quantifying sources of error in mcpat and potential impacts on architectural studies, in: 21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015, 2015, pp. 577–589. `doi:10.1109/HPCA.2015.7056064`.
URL `http://dx.doi.org/10.1109/HPCA.2015.7056064`

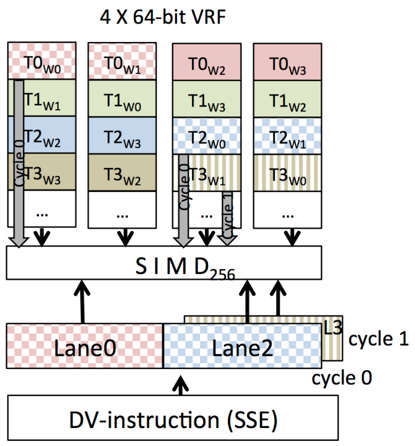
[41] C. Collange, D. Defour, Y. Zhang, Dynamic detection of uniform and affine vectors in GPGPU computations, in: Europar 3rd Workshop on Highly Parallel Processing on a Chip (HPPC), Vol. LNCS 6043, 2009, pp. 46–55.
URL `http://hal.archives-ouvertes.fr/hal-00396719/en/`



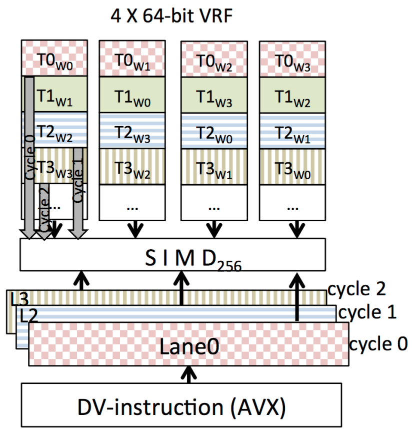
(a) Scalar ALU instruction



(b) Scalar SSE (floating-point) with mask 1111



(c) Packed 128-bit SSE with mask 1011



(d) Packed 256-bit AVX with mask 1011

Figure 10: Operand collection on $4W \times 4T$ DITVA depending on DV-instruction type and execution mask. 'w' represents a 64-bit word

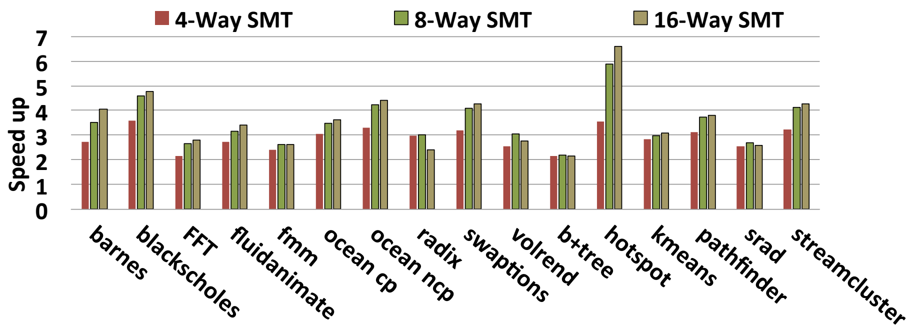


Figure 11: Speedup with thread count in the baseline SMT configuration, normalized to single-thread performance

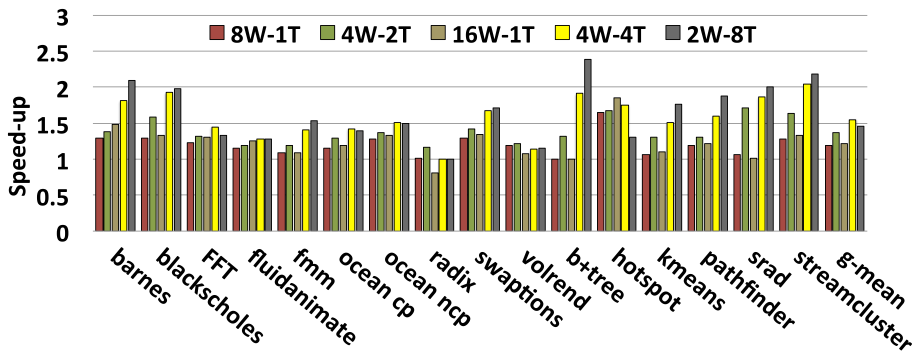


Figure 12: Speed-up over 4-thread SMT as a function of warp size

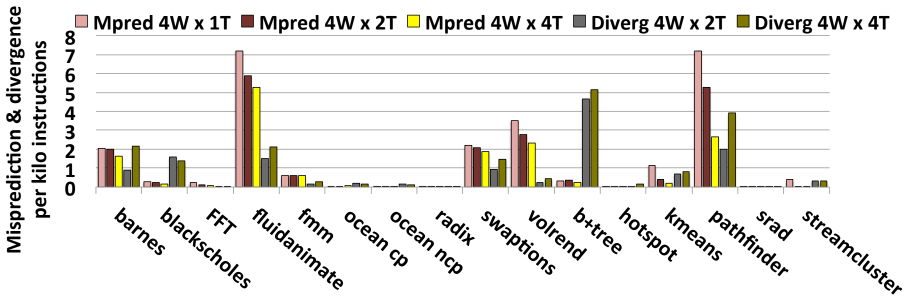


Figure 13: Divergence and mispredictions per thousand instructions

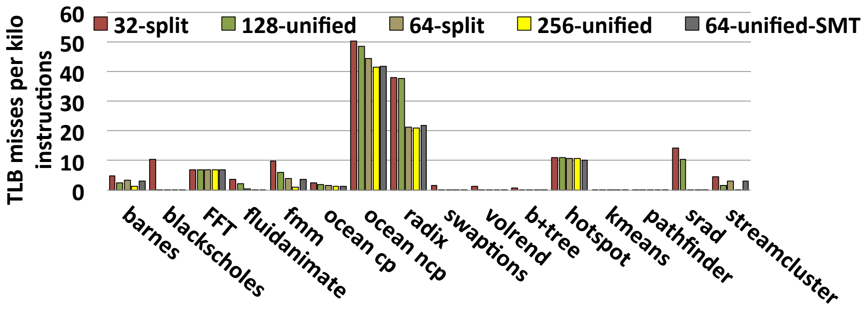


Figure 14: TLB misses per thousand instructions for split or unified TLBs on $4W \times 4T$ DITVA

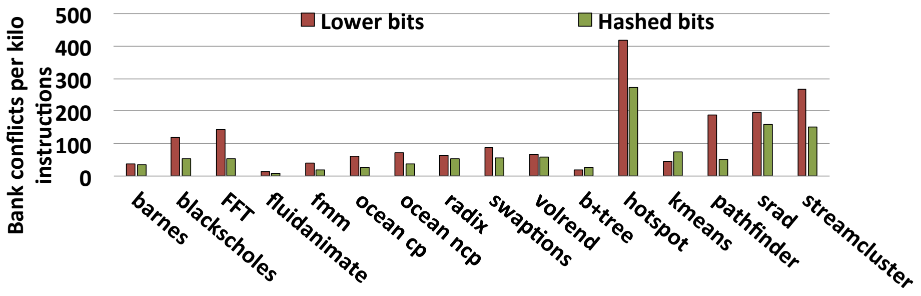


Figure 15: Bank conflicts for $4W \times 4T$ DITVA

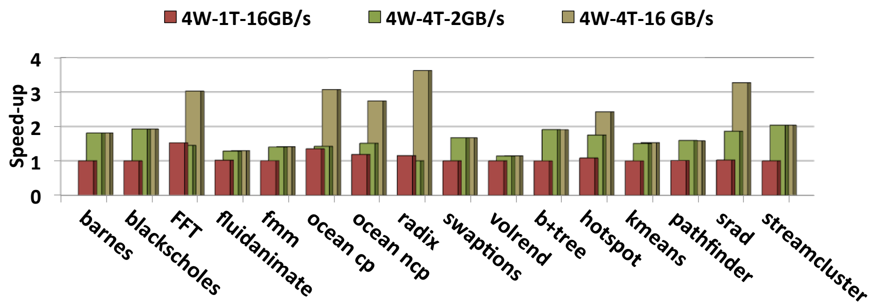


Figure 16: Performance scaling with memory bandwidth, relative to 4-thread SMT with 2 GB/s DRAM bandwidth

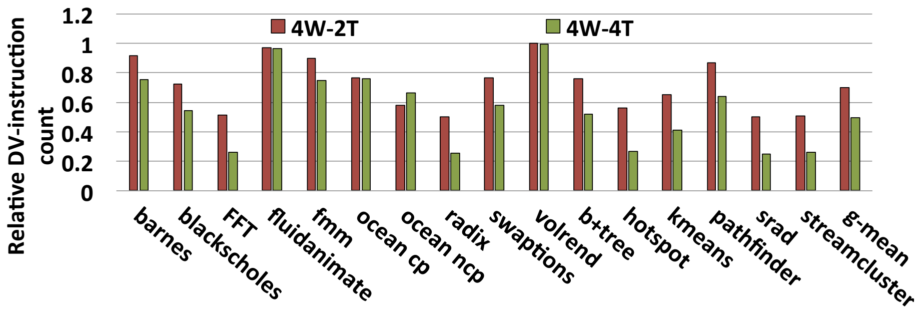


Figure 17: DV-instruction count reduction over 4-thread SMT as a function of warp size

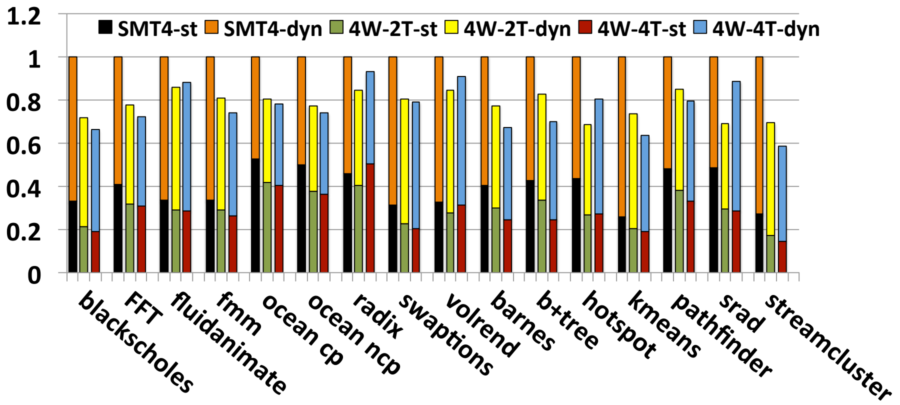


Figure 18: Relative energy consumption over 4-thread SMT. *-dyn* and *-st* are dynamic and static energy, respectively.