Transforming TLP into DLP with the Dynamic Inter-Thread Vectorization Architecture

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

Transforming TLP into DLP with the Dynamic Inter-Thread Vectorization Architecture

Sajith Kalathingal, Sylvain Collange, Bharath N. Swamy, André Seznec

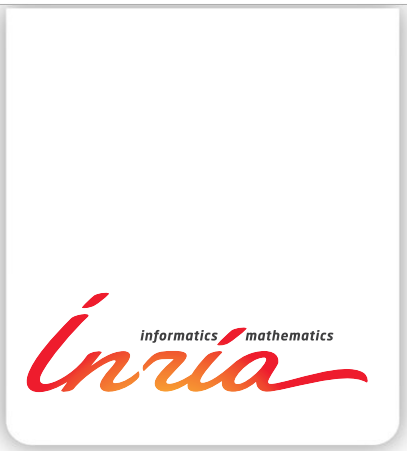
RESEARCH REPORT

N° 8830

December 2015

Project-Team ALF

ISSN 0249-6399 ISRN INRIA/RR--8830--FR+ENG



Transforming TLP into DLP with the Dynamic Inter-Thread Vectorization Architecture

Sajith Kalathingal, Sylvain Collange, Bharath N. Swamy, André Seznec

Project-Team ALF

Research Report n° 8830 — December 2015 — 25 pages

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 to create dynamic vector instructions at runtime. DITVA extends an SIMD-enabled in-order SMT processor with an inter-thread vectorization execution mode. In this mode, identical instructions of several threads running in lockstep are aggregated into a single SIMD instruction. DITVA leverages existing SIMD units, balances TLP and DLP with a warp/thread hierarchy, and maintains binary compatibility with existing CPU architectures.

Key-words: Simultaneous MultiThreading, Single instruction multiple data, Single program multiple data, Vectorization

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Transformer le TLP en DLP par la vectorisation dynamique entre threads

Résumé : Les threads des applications SPMD (single-program, multiple-data) exécutent souvent les mêmes instructions sur des données différentes. Nous proposons l'architecture de vectorisation dynamique inter-thread (DITVA) pour tirer parti de ce parallélisme de données (DLP) implicite dans les applications SPMD pour créer des instructions vectorielles dynamiques à l'exécution. DITVA étend un processeur SMT avec unité SIMD par un mode d'exécution à vectorisation inter-thread. Dans ce mode, les instructions identiques de plusieurs threads synchronisés sont agrégées en une instruction SIMD unique. DITVA tire parti des instructions SIMD existantes, équilibre le TLP et DLP par une hiérarchie warp/thread, et maintient la compatibilité binaire avec les architectures CPU existantes.

Mots-clés : SMT, SIMD, SPMD, vectorisation

1 Introduction

Single-Program Multiple-Data (SPMD) applications express parallelism by creating multiple instruction streams executed by 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, i.e., often executing 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 – thread execution can be asynchronous – nor leveraged by current processors.

Simultaneous Multi-Threaded (SMT) processors [35, 33] 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. 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 [35] with a dynamic vector execution mode supported by SIMD units. Whenever possible DITVA executes the same instruction for different threads running in lockstep on replicated functional units. At runtime, DITVA synchronizes threads within groups, or *warps*, to uncover the hidden DLP. Dynamic vectorization does not require additional programmer effort or algorithm changes to the existing SPMD applications. DITVA even preserves binary compatibility with the existing general purpose CPU architectures as it does not require any modification in the ISA.

Our experiments on SPMD applications from the PARSEC and SPLASH benchmarks [1, 36] show that the number of instructions fetched and decoded can be reduced, on average, by 40% on a 4-warp × 4-thread DITVA architecture compared with a 4-thread SMT. Coupled with a realistic memory hierarchy, this translates into a speed-up of 1.44× 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 22% average energy reduction. Therefore, DITVA appears as a very energy-effective design to execute SPMD applications.

The remainder of the paper is organized as follows. Section 2 motivates the DITVA proposition for a high throughput SPMD oriented processor architecture. Section 3 reviews some related works. Section 4 describes our proposed DITVA architecture. Section 5 presents our experimental framework and the performance and design tradeoffs of DITVA. Section 6 describes the implications of DITVA on hardware complexity and does power consumption analysis. Finally, Section 7 concludes this study and introduces directions for future 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 mutualize a significant part of the instruction execution of the different threads.

2.1 SMT architectures

SMT architectures [35] were introduced to exploit thread-level and/or multi-program level parallelism to optimize the throughput of a superscalar core. Typically, on an SMT processor, instructions from the different hardware threads progress concurrently in all stages of the pipeline. Depending on the precise implementation, some pipeline stages only treat instructions from a single thread at a given cycle. For instance the instruction fetch pipeline stage [31], while some other pipeline stages like the execution stage, 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 [9].

SMT architectures have often been targeting both high single-thread performance and high 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 have been shown to reach 85 % of the performance of an out-of-order 4-thread SMT 4-issue processor [10]. Therefore, in-order SMT appears as a good architecture tradeoff for implementing the cores of an SPMD oriented throughput processor.

2.2 Instruction redundancy in 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 reconvergence 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 [1] 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 reconvergence points.

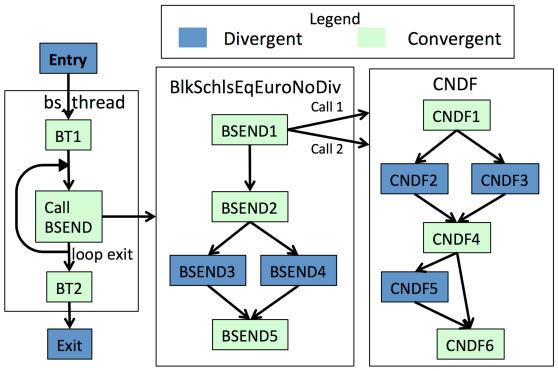


Figure 1: Control flow graph of blackscholes benchmark

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 are fetched, decoded, etc. by each of the thread without any mutualization of this redundant effort. The same applies to the divergent blocks that are executed by more than one thread. This appears as a large waste of resources. Indeed, a prior study on the PARSEC benchmarks 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 [24].

The DITVA architecture leverages 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 works

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 till the 1990s in vector supercomputers [29]. Performance on these systems was highly dependent on the ability of application programmers to express computations as operations on contiguous, strided or scatter/gather vectors and on the ability of compilers to detect and use these vectors [27]. In the mid-1990s, SIMD instructions were introduced in the microprocessor ISAs, first to deal with multimedia applications [12] e.g. VIS for the SPARC ISA or MMX for x86. More recently ISAs have been augmented with SIMD instructions addressing scientific computing, e.g. SSE and AVX for x86_64.

However, SIMD instructions are not always the practical vehicle to express and exploit possible DLP in all programs. In many cases, compilers fail to vectorize loop nests with independent loop iterations that have potential control flow divergence or irregular memory accesses. One experimental study [19] shows that only 45-71% of the vectorizable code in Test Suite for Vectorizing Compilers (TSVC) and 18-30% in Petascale Application Collaboration Teams (PACT) and Media Bench II was vectorized by auto-vectorization of ICC, GCC and XLC compilers because of inaccurate analysis and transformations.

Also, the effective parallelism exploited through SIMD instructions is limited to the width (in number of data words) in the SIMD instruction. 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. An SMT core can exploit this parallelism as TLP. Our DITVA proposal further dynamically transforms this TLP into dynamic DLP.

3.2 The SIMT execution model

GPUs extract DLP from SPMD workloads by assembling vector instructions across fine-grained threads. GPUs are programmed with SPMD applications, written in low-level languages like CUDA or OpenCL, or compiled from higher-level languages like OpenACC. The SIMT execution model used on NVIDIA GPUs groups threads statically into warps, currently made of 32 threads. All threads in a warp run in lockstep, executing identical instructions. SIMD units execute these warp instructions, each execution lane being dedicated to one thread of the warp. To allow

threads to take different control flow paths through program branches, a combination of hardware and software enables differentiated execution based on per-thread predication. On NVIDIA GPU architectures, a stack-based hardware mechanism keeps track of the paths that have diverged from the currently executing path. It follows explicit divergence and reconvergence instructions inserted by the compiler [26].

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 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 [5, 23] or more flexible execution [8, 2, 11]. However, they all target applications specifically written for GPUs, rather than general-purpose parallel applications.

3.3 SMT

SMT improves the throughput of a superscalar core by enabling independent threads to share CPU resources dynamically. Resource sharing policies have [35, 34, 3, 17, 6, 7] huge impact on execution throughput. 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. Fairness among threads has been recognized as an important property that should be also tackled by resource sharing policies [18]. However, these resource sharing heuristics essentially address multi-program workloads.

DITVA targets SPMD applications, where threads run the same code. Unlike GPUs that rely on hard explicit reconvergence points, DITVA relies on resource sharing policies to restore lockstep execution of threads. In the past studies, a few policies favoring lockstep or near lockstep execution of threads execution have been proposed. A simple heuristic, which we will refer as Min-PC, prioritizes the threads based on the minimum value of their Program Counter (PC) [28, 13]. It is based 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. This heuristics will fail if the compiler heavily reorders the basic blocks. Similar heuristics has also been used in the GPU context to synchronize multiple warps together without relying on reconvergence points [8, 25]. MinSP-PC was proposed [4] to improve MinPC heuristic by first prioritizing threads based on the minimum relative value of the stack pointer. The underlying assumption here is that the size of the stack increases with each function call, resulting in a lower value of stack pointer. Therefore, the threads inside inner-most function calls are given priority.

Minimal Multi-threading [15] or MMT is the closest proposition to DITVA. 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. DITVA keeps instructions synchronized throughout the pipeline, allowing a resource-efficient in-order SIMD backend. In the same spirit, Execution Drafting [20] 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, we find that 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. To address this issue, DITVA groups threads into SIMT-style warps, which are scheduled independently and provide latency hiding capabilities.

4 The Dynamic Inter-Thread Vectorization Architecture

In this section, we present Dynamic Inter-Thread Vectorization Architecture (DITVA). Figure 2

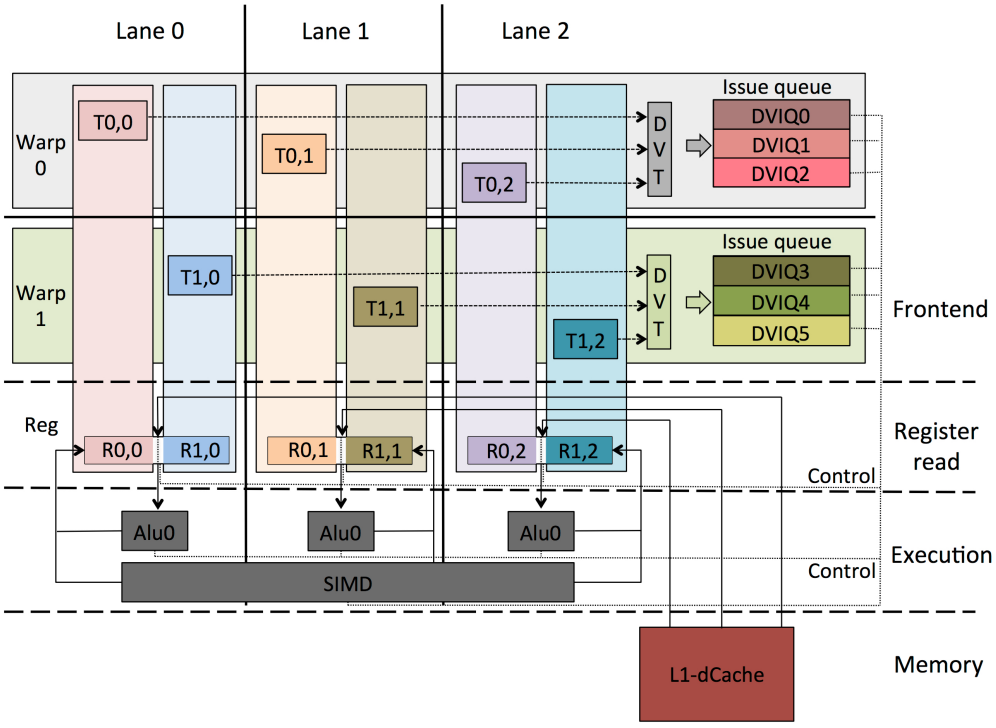


Figure 2: A logical view of the DITVA architecture

illustrates a $2W \times 3T$ DITVA processor. A $nW \times mT$ DITVA supports $n \times m$ thread contexts. The threads are executed on m execution lanes, 3 on Figure 2. Each lane replicates the register file and the functional units. The thread contexts are statically partitioned into n groups of m threads, one thread per lane. We will refer to such a group as a warp following NVIDIA GPU terminology, e.g. T1,0, T1,1 and T1,2 form Warp 1. A given thread is assigned to a fixed lane, e.g. T0,1 executes on Lane 1. We use the notation $nW \times mT$ to represent a DITVA configuration with n Warps and m Threads per warp. A $nW \times 1T$ DITVA is equivalent to a n-thread SMT, while $1W \times mT$ DITVA performs dynamic vectorization across all threads.

On DITVA, whenever the threads of a warp in an SPMD application are synchronized, i.e., have the same PC, the DITVA pipeline executes the instructions from all the threads as a single dynamic vector instruction. The instruction is fetched only once for all the threads of the warp, and a single copy of the instruction flows through the pipeline. That is, decode, dependency check, issue and validation are executed only once, but execution, including operand read, operation execution and register result write-back is performed in parallel on the m lanes.

DITVA supports dynamic vectorization of non-SIMD instructions as well as statically vectorized SIMD instructions such as SSE and AVX.

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 to be executed in lockstep mode. 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.

Notations We will refer to the group of threads from the same process running in lockstep with the same PC as a DV-thread. Likewise, a vector of instruction instances from different threads with the same PC is referred to as a DV-instruction. A DV-instruction consisting of scalar instructions are referred to as DV-scalar and those with SIMD instructions such as AVX and SSE are referred to as DV-SIMD. 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.

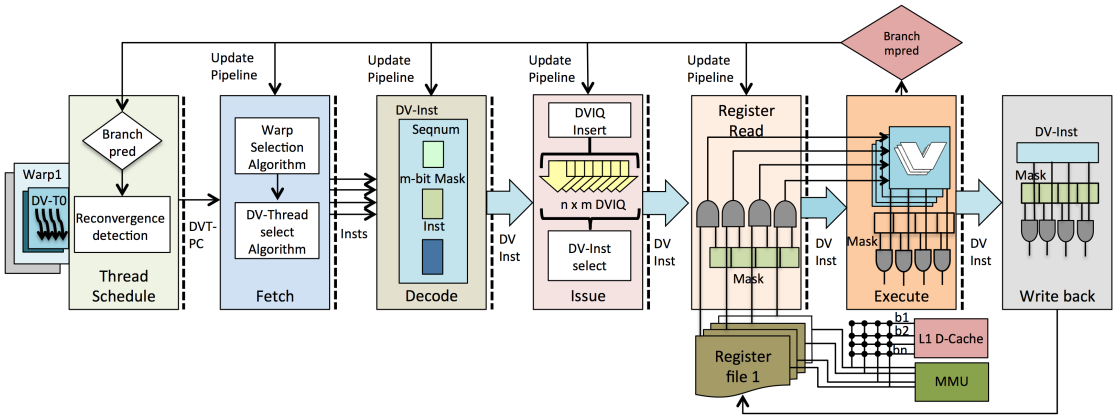


Figure 3: Overview of the DITVA pipeline

4.1 Pipeline architecture

We describe the stages of the DITVA pipeline, as illustrated in Figure 3.

4.1.1 Front end

On an SMT processor, on each cycle, a thread is selected based on an instruction fetch steering policy [35, 34, 3, 17, 6, 7]. Then a block of instructions from the selected thread is fetched from the I-cache and then forwarded to the whole pipeline. This allows all the threads to share the instruction cache without encountering any inter-thread I-cache bank conflict. At the same time, the speculative PC of the next block of instructions is predicted using the branch history of the selected thread. After the decode stage, instructions are pushed into an instruction queue associated with the selected thread.

On DITVA, threads are encapsulated in a two-level structure. Each warp features from 1 to m active DV-threads. In turn, each DV-thread tracks the state of 1 to m scalar threads (Figure 4a). Within the front-end, the basic unit of scheduling is the DV-thread. The state of

one DV-thread consists of one process identifier, one PC and an m-bit mask that tracks which threads of the warp belong to the DV-thread. Bit i of the mask is set when thread i within the warp is part of the DV-thread. Also, each DV-thread has data used by the fetch steering policy, such as the call-return nesting level (Section 4.3).

Branch prediction and re-convergence detection Both the PC and mask of each DV-thread are speculative. An instruction address generator initially produces a PC prediction for one DV-thread. After the instruction address generation, the PC and process identifier of the predicted DV-thread T are compared with the ones of the other DV-threads. A match indicates that the two DV-threads have reached a point of re-convergence and may be merged. When merging happens, the mask of T is updated to the logical OR of its former mask and the mask of the other DV-thread. The second DV-thread is aborted. Figures 4a and 4b illustrates the re-convergence for Warp 0 where threads DV-T2 and DV-T0 re-converge at PC 142. When no re-convergence is detected, the mask of the DV-thread is unaffected. All the threads of the DV-thread share the same instruction address generation, by speculating that they will all follow the same branch direction. Accordingly, thread divergence within a DV-thread will be handled as a branch misprediction (Section 4.1.4).

Fetch and decode Reflecting the two-level organization in warps and DV-threads, instruction fetch obeys two fetch steering policies. First, a warp is selected following a similar policy as in the SMT case. Then, an intra-warp instruction fetch steering policy (that will be described in Section 4.3) selects one DV-thread within the selected warp. From the selected DV-thread 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 DV-thread 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. In a conventional SMT, instruction queues are typically associated with individual threads. DITVA follows a similar model, but applies it at the DV-thread granularity: each DVIQ tail is associated with one DV-thread. Unlike in SMT, instructions that are further ahead in the DVIQ may not necessarily belong to the DV-thread currently associated with the DVIQ, due to potential DV-thread divergence and reconvergence. The threads that own a given instruction are explicitly identified by the DV-instruction mask, rather than implicitly by their home queue.

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 correspond 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 DV-threads, if the consumer instruction lies after a reconvergence point. Therefore, the DITVA scoreboard mechanism must take into account the masks of all the previous in-flight DV-instructions of the warp to ensure operand availability, including instructions from other DVIQs. A straightforward scoreboard implementation uses a $nr \times m$ table of masks with $8m$ ports. However, 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. Each register is accessed individually. 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 register file.

In contrast, DITVA implements a split 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 uses the existing SIMD units of a superscalar processor for the execution of statically vectorized SIMD instructions. However, DITVA factorizes the control.

Figure 5a shows the execution of DV-scalar instruction in a 4×4 DITVA. A DV-scalar reads DV-registers (SRF), i.e. reads from the same registers in the m register files for all the threads of a warp, executes on a DV-unit, i.e. on m similar functional units and writes the m results to the same register in the m register files. All these actions are conditioned by the mask of the DV-instruction. Thus, for DV-scalar instructions, the DITVA back-end is essentially equivalent to an SIMD processor with per-lane predication.

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. Whenever possible, DITVA keeps explicit vector instructions as contiguous vectors when executing them on SIMD units. That is, data-level parallelism comes in priority from explicit vector instructions, then from dynamic vectorization across threads. This allows to maintain the contiguous memory access patterns when reading and writing vectors in memory. However, in order to efficiently support partial access to vector registers, 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 slide $i \oplus j$, $\oplus$ 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 banking enables both contiguous execution of full 256-bit AVX instructions, and partial dynamic vectorization of 128-bit vector and 64-bit scalar SSE instructions to fill the 256-bit datapath. Figure 5b 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 5c shows the pipelined execution of a SSE DV-instruction with mask 1011 in a 4×4 DITVA. In figure 5c, T0 and T2 are issued in the first cycle and T1 is issued in the subsequent cycle. Finally, full-width AVX instructions within a DV-instructions are issued in successive cycles to a pipelined functional unit. Figure 5d shows the execution of a AVX DV-instruction with mask 1101 in a 4×4 DITVA. Division operations are not pipelined, and their execution are completely serialized.

As the DITVA processor is p-issue superscalar, on each cycle up to p independent DV-instructions are picked from the heads of the DVIQs.

4.1.4 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, mispredictions or exceptions may be either partial or total, depending on whether or not all the threads within the DV-thread encounter the event. A partial branch misprediction corresponds to DV-thread divergence when some threads of the DV-thread follow a different direction than the correctly-predicted threads. Handling an exception would be similar to handling a branch misprediction.

On DITVA, on a mispredicted branch, partial or total, for every mispredicted thread T, the following actions must be done.

1. Reset the speculative thread PC of T to its effective PC.
2. Remove the Instructions from T from the pipeline.

As the DV-instructions encapsulates the thread instructions, the point 1 can be managed as follows. First, mispredicted threads are grouped into sets of same-PC threads, to form a new DV-thread on each mispredicted path. In the case of a two-sided conditional branch, there is only one set containing all mispredicted threads. Then, the initial DV-thread is split between the correctly-predicted path and each mispredicted path. In this scenario, the bits corresponding to mispredicted threads are cleared in the mask of the initial DV-thread and spawns a new DV-thread for each mispredicted path. The sequence numbers of the threads in the new DV-thread are reset to follow the sequence numbers of the branch instruction in each thread.

To implement point 2, the bits corresponding to the mispredicted threads are cleared in all the masks of the DV-instructions in progress in the pipeline and in the DVIQ. This corresponds to disabling all the instructions fetched from the wrong path for each thread T. Figure 4b shows the state of the pipeline when the instruction PC-11d of thread T5 illustrated in figure 4a triggers a branch misprediction. In the given example, a new DV-thread is spawned for thread T5, which starts fetching from the other branch after misprediction.

In addition, some bookkeeping is needed. In the case of a full misprediction, the masks of some DV-instructions become null, i.e. no valid thread remains. 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, a DV-thread with an empty mask is aborted.

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

Full 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, partial misprediction has no significant performance impact as both branch paths are eventually taken.

4.2 Data memory accesses

A DV-load (resp. DV-store) of a full 256-bit AVX DV-SIMD instruction is pipelined. Each data access request corresponding to the participant thread is serviced in the successive cycles. For a 128-bit SSE DV-SIMD 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. A data access operation in a DV-scalar 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 very high. In particular when the data are truly shared, and handling multiple writes requires implementing effective multiple ports (and not just simple replication of the data cache). Therefore, the proposed implementation of DITVA relies on a banked data cache. Banking is performed at cache line granularity. 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. The load data path must be able to support the special cases of several threads accessing the same element, for both regular and atomic memory operations.

It is possible to 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, the threads do not systematically use the same data pages. That is, Thread I only references the pages directly accessed by it in the data TLB associated with itself. Our simulations in Section 5 show that this optimization significantly decrease the total number of TLB misses or allows to use smaller TLBs.

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 may lose synchronization as soon as there is a control flow divergence among the threads. Apart the synchronization points inserted by the application developer or the compiler, the instruction fetch policy and the execution priority policy are the two possible vehicles to restore lockstep execution.

One of the most simple yet fairly efficient fetch policies to reinitiate lockstep execution is MinSP-PC [4]. 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 DV-threads with free DVIQ slots, if any DV-thread has not got the instruction fetch priority for $(m+1) \times n$ cycles, then it gets the priority. Otherwise, the MinSP-PC DV-thread is scheduled.

This hybrid fetch policy is biased toward the DV-thread 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.

5 Evaluation

We simulate DITVA to evaluate its performance and design tradeoffs.

5.1 Experimental Framework

We model DITVA using an in-house trace-driven x86_64 simulator. A Pin tool [16] 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 basic 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

[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

Configuration	
L1 data cache	32 KB, 16 ways LRU, lat 2 cycles
L2 cache	4MB, 16 ways LRU, lat 15 cycles
L2 miss latency	215 cycles
Cache banks	16
Branch predictor	64-Kbit TAGE [30]
DVIQs	$n \times m$ 16-entry queues
DV-thread 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 $n \times$ 64-bit ALUs, 2 256-bit AVX/FPUs, 1 $n \times$ 64-bit mul/div, 1 256-bit load/store

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 2 $n \times$ 64-bit ALUs, achieving the equivalent of $4n \times$ 64-bit ALUs.

We evaluate DITVA on SPMD benchmarks from the PARSEC [1] and SPLASH 2 [36] suites compiled for $corei7-avx$ architecture. We simulate the following benchmarks: *Barnes*, *Blackscholes*, *Fluidanimate*, *FFT*, *Fmm*, *Swaptions*, *Radix*, *Volrend*, *Ocean CP* and *Ocean NCP*. Since accurate simulation of the relative progress of threads is fundamental for our study, we simulate entire benchmarks rather than rely on sampling techniques. We obtained the results in this section with the *simsmall* input dataset.

Figure 6 shows the performance scaling of single-thread and SMT configurations with 4, 8 and 16 threads. Application exhibit diverse scaling behavior with thread count. *FFT*, *Ocean* and *Radix* 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 and $4W \times 4T$ DITVA, i.e., 4-way SMT with 4 lanes.

5.2 Bank conflict reduction

Bank interleaving using the low order bits on the L1 data cache leads to a noticeable number of bank conflicts, as illustrated in Figure 7. We find that many conflicts are caused by concurrent accesses to the stack. When the base addresses of the thread call stacks are aligned on page boundaries, concurrent accesses at the same offset in different stacks result in bank conflicts. This observation matches the findings of prior studies [21, 24].

To reduce such bank conflicts for DV-loads and DV-stores, we use a hashed set index as suggested 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 7 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*. In

the remainder of the evaluation section, this hashed set index is used.

5.3 Vectorization efficiency

Figure 8 illustrates the DV functional unit occupancy with a breakdown of individual instructions count per DV-instruction on DITVA $4W \times 4T$. The *k thread* bar represents the normalized frequency of occurrence of DV-instructions with k active threads. This figure represents the DLP that could be extracted from the SPMD program. *Radix* and *FFT* have vast amounts of DLP. Hence, most of their instructions are perfectly combined to form DV-instructions. On the other hand, the threads of benchmarks that have low exploitable DLP like *Fluidanimate* tend to diverge. Only 23% of DV-instructions contain more than one instruction on *Fluidanimate*.

Dynamic vectorization reduces the number of DV-instructions over original instructions. Figure 9 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 74% for $4W \times 2T$ DITVA and 60% for $4W \times 4T$ DITVA. DV-instruction count is low for applications for *Radix* and *FFT*, which has nearly perfect dynamic vectorization. However, the DV-instruction count reduction in *Volrend*, *Fluidanimate* and *Ocean* is compensated by the parallelization overhead caused by the thread count increase.

5.4 Throughput

Figure 10 shows the speed-up achieved for $4W \times 2T$ DITVA and $4W \times 4T$ DITVA over 4-thread SMT with AVX instructions. For reference, we illustrate the performance of 16-thread and 8-thread SMT ($16W \times 1T$ and $8W \times 1T$ respectively). On average, $4W \times 2T$ DITVA achieves 31% higher performance than 4-thread SMT and $4W \times 4T$ DITVA achieves 44% performance improvement. The $4W \times 4T$ DITVA also achieves 20% speedup over 16-thread SMT. The application speedup is due to replicated ALU and efficient utilization of AVX units for the execution of SSE and scalar floating-point instructions.

Due to memory hierarchy related factors, the actual speed-up is not proportional to DV-instruction occupancy. For instance, although *Radix* has high dynamic vectorization potential, its performance is rather disappointing with four lanes performing worse than two lanes. Indeed, with four lanes and so 16 threads *Radix* experiences many cache misses that exhausts the memory bandwidth. With two lanes and so only 8 threads, the demand on memory bandwidth is much smaller and some speed-up is encountered. For applications with low DLP, like *Fluidanimate*, the performance of DITVA is on par with 16-thread SMT. The relative speed up of *Fluidanimate*, *Ocean CP*, *Ocean NCP* and *Volrend* is impacted by the increase in the number of instructions with the increase in the number of threads.

5.5 Impact of split data TLB

Each execution lane of DITVA must feature a data TLB or have an access port to the data TLB, Not only the TLB must be duplicated, but one should also increase its number of entries in order to limit the TLB miss rate, since the number of threads executed on DITVA is larger than the one executed on the 4-thread SMT.

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 11 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.

5.6 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 which is a feasible alternative in the modern multi-core microprocessors. The performance scaling of 16 GB/s relative to 4-thread SMT with 2 GB/s DRAM bandwidth is illustrated in Figure 12.

For many benchmarks, 2 GB/s bandwidth is sufficient. However, as discussed in section 5.1, the performance of *Ocean*, *Radix* and *FFT* is bound by memory throughput. DITVA enables these applications to benefit from the extra memory bandwidth, 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 DV-thread reconvergence, the logic to select the DV-thread 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 DV-threads 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 40% of instruction fetches for $4W \times 4T$ DITVA.

The 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 On the 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 [32] and McPAT 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.

DV-units The replication of the two scalar functional units to form DV-units introduces the most significant hardware area overhead. The DV-units have a higher leakage and require higher instantaneous power supply than the functional units of the in-order execution SMT core. 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 roughly the same for DITVA and the in-order SMT core on a given workload, and so does the dynamic energy.

Pipeline Front End The modifications in the pipeline front-end induce essentially extra logic, e.g. comparators and logic to detect DV-thread reconvergence, the logic to select the DV-thread 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 DV-threads per warp. However, this scoreboard is also banked since there are no inter-thread register dependencies.

On the other hand, DITVA eliminates significant dynamic energy consumption in the front-end. Our experiments show a reduction of 40% of instruction fetches for $4W \times 4T$ DITVA.

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-threaded 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 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

We modeled a baseline SMT processor and DITVA within McPAT [14]. 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 13 while modeled silicon area and static energy are reported in Table 2. 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 [37]. 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

13 and Table 2 show that DITVA appears as an energy-effective solution for SPMD applications with respectively 20% and 22% average energy reduction for $4W \times 2T$ and $4W \times 4T$ DITVA.

Table 2: 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

The energy reduction is the result of both a decrease in run-time (Figure 10) 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 uses dynamic vector execution to allow lockstep parallel execution as in SIMT GPUs, but retaining binary compatibility with existing general-purpose CPUs. Applications require no source modification nor re-compilation.

On DITVA, threads are statically grouped 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 whenever their PC are equal. Then the group of instructions (the DV-instruction) progresses in the pipeline in lockstep mode. This allows to mutualize the instruction front-end as well the overall instruction control. The instructions from the different threads in a DV-instruction are executed on replicated execution lanes. DITVA maintains a 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 reconvergence 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 reconvergence 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 $4W \times 4T$ lanes DITVA processors are cost-effective design points. For instance, a $4W \times 4T$ DITVA architecture reduces instruction count by 40% and improving performance by 44% over a 4-thread 4-way issue SMT on the SPMD applications from PARSEC. 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, some micro-architecture improvements proposed for SIMT could also apply to DITVA. For instance, more flexibility could be obtained using Dynamic Warp Formation [8] or Simultaneous Branch Interweaving [2], while Dynamic Warp Subdivision [22] could improve latency tolerance by allowing threads to diverge on partial cache misses.

References

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[2] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained GPU performance. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 49–60. IEEE Computer Society, 2012.

[3] Francisco J. Cazorla, Alex Ramírez, Mateo Valero, and Enrique 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*, pages 171–182, 2004.

[4] Sylvain Collange. Stack-less simt reconvergence at low cost. Technical report, HAL, 2011.

[5] Gregory Diamos, Andrew Kerr, Haicheng Wu, Sudhakar Yalamanchili, Benjamin Ashbaugh, and Subramaniam Maiyuran. SIMD re-convergence at thread frontiers. In *MICRO 44: Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, December 2011.

[6] Ali El-Moursy and David 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*, pages 31–40, 2003.

[7] Stijn Eyerman and Lieven 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*, pages 240–249, 2007.

[8] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Trans. Archit. Code Optim.*, 6:7:1–7:37, July 2009.

[9] Sébastien Hily and André 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*, pages 169–173, 1996.

[10] Sébastien Hily and André 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*, pages 64–67, 1999.

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

[12] Ruby Lee. Multimedia extensions for general-purpose processors. In *Proc. IEEE Workshop on Signal Processing Systems*, pages 9–23, 1997.

[13] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. *ACM Transactions on Computer Systems (TOCS)*, 31(3):6, 2013.

[14] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman 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.*, pages 469–480. IEEE, 2009.

[15] Guoping Long, Diana Franklin, Susmit Biswas, Pablo Ortiz, Jason Oberg, Dongrui Fan, and Frederic 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*, pages 337–348. IEEE Computer Society, 2010.

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[17] Kun Luo, Manoj Franklin, Shubhendu S. Mukherjee, and André 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*, page 2, 2001.

[18] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing thoughput and fairness in SMT processors. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software, November 4 - 6,2001, Tucson, Arizona, USA, Proceedings*, pages 164–171, 2001.

[19] Saeed Maleki, Yaoqing Gao, María Jesús Garzaran, Tommy Wong, and David A Padua. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382. IEEE, 2011.

[20] Michael Mckeown, Jonathan Balkind, and David Wentzlaff. Execution drafting: Energy efficiency through computation deduplication. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 432–444. IEEE, 2014.

[21] Jiayuan Meng and Kevin 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*, pages 282–288. IEEE, 2009.

[22] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3):235–246, 2010.

[23] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. iGPU: exception support and speculative execution on GPUs. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 72–83. IEEE Computer Society, 2012.

[24] Teo Milanez, Sylvain Collange, Fernando Magno Quintão Pereira, Wagner Meira, and Renato Ferreira. Thread scheduling and memory coalescing for dynamic vectorization of SPMD workloads. *Parallel Computing*, 40(9):548–558, 2014.

[25] Veynu Narasiman, Chang Joo Lee, Michael Shebanow, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU performance via large warps and two-level warp scheduling. In *MICRO 44: Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, December 2011.

[26] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, March 2010.

[27] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[28] Michael J Quinn, Philip J Hatcher, and Karen C Jourdenais. Compiling c* programs for a hypercube multicomputer. In *ACM SIGPLAN Notices*, volume 23, pages 57–65. ACM, 1988.

[29] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978.

[30] André Seznec. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–127. ACM, 2011.

[31] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis 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*, pages 295–306, 2002.

[32] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. Cacti 5.1. Technical report, HP Laboratories, 2008.

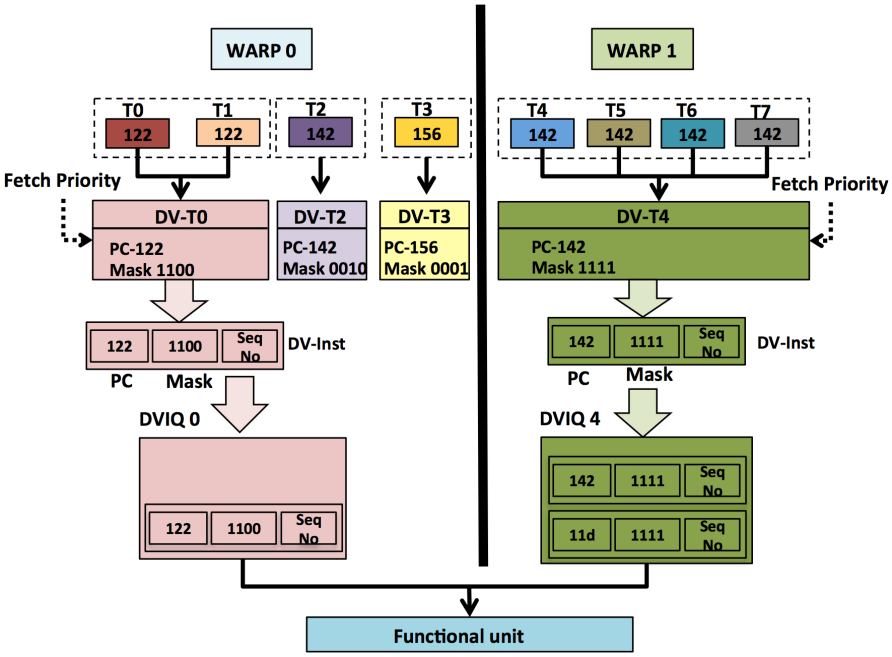
[33] Dean M Tullsen, Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, and Rebecca L Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ACM SIGARCH Computer Architecture News*, volume 24, pages 191–202. ACM, 1996.

[34] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca 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*, pages 191–202, 1996.

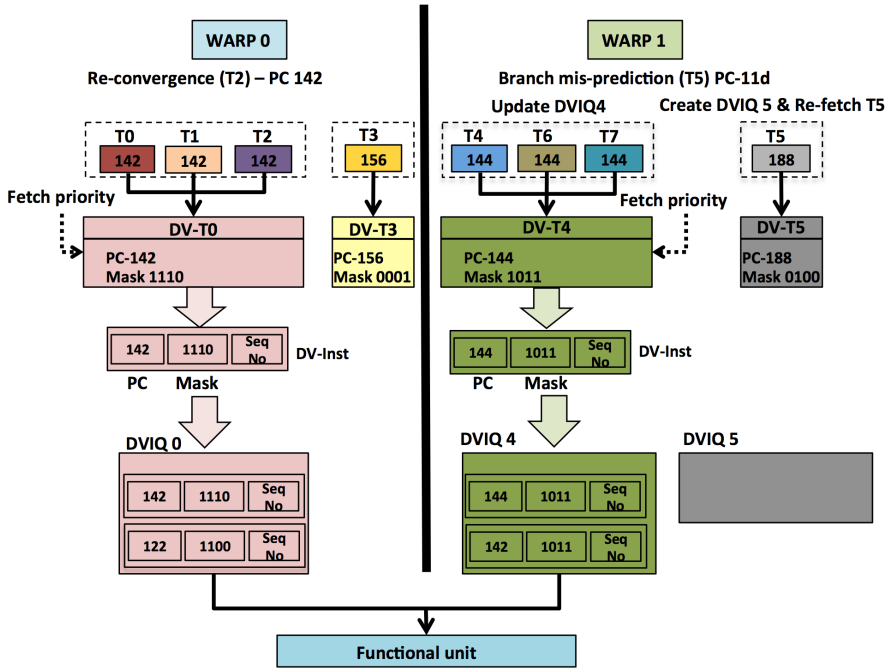
[35] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*, pages 392–403, 1995.

[36] University of Washington. Splash-2, 2006. Benchmark package.

[37] Sam Likun Xi, Hans M. Jacobson, Pradip Bose, Gu-Yeon Wei, and David 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*, pages 577–589, 2015.

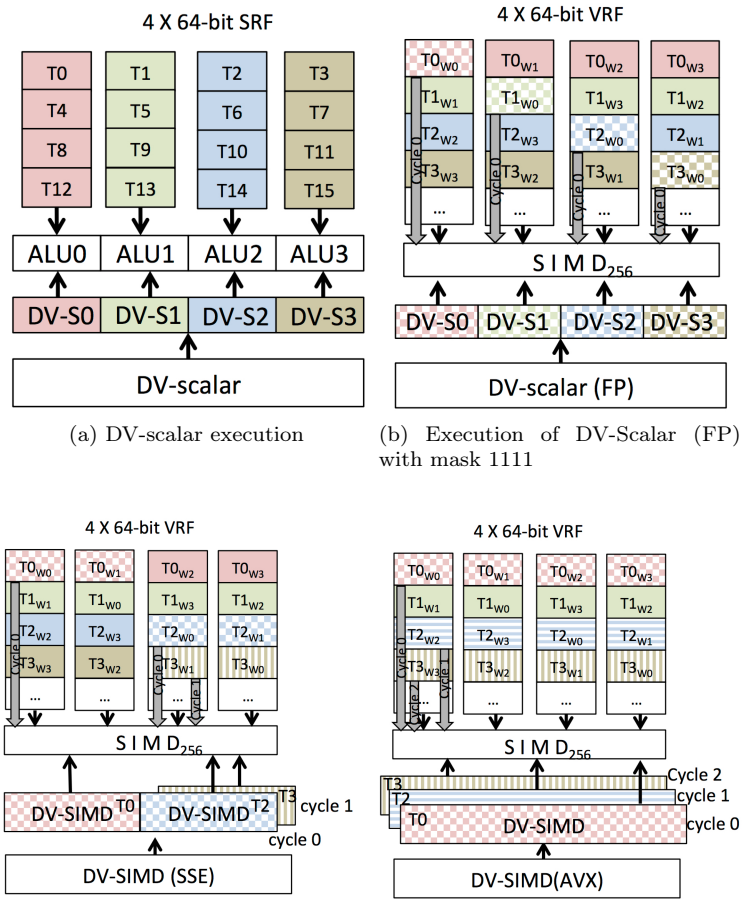


(a) Initial state



(b) After reconvergence in Warp 0 and misprediction in Warp 1

Figure 4: Evolution of the DV-thread and DVIQ states upon thread divergence and reconvergence



(a) DV-scalar execution

(b) Execution of DV-Scalar (FP) with mask 1111

(c) Execution of DV-SIMD (SSE) with mask 1011

(d) Execution of DV-SIMD (AVX) with mask 1011

Figure 5: DV-instruction execution for $4W \times 4T$ DITVA

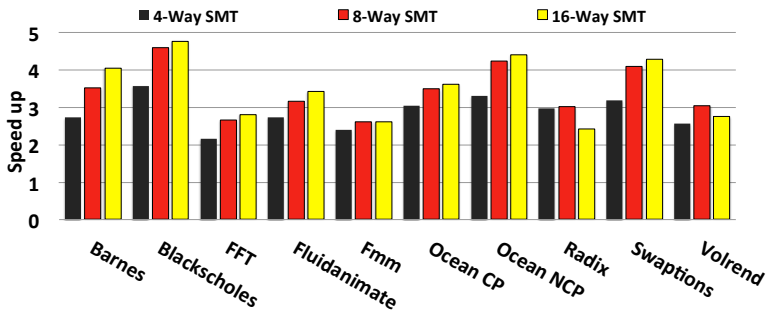


Figure 6: Speedup with thread count in the baseline SMT configuration, normalized to single-thread performance

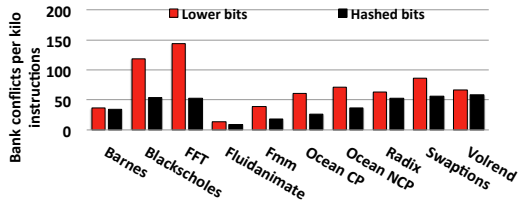


Figure 7: Bank conflicts for $4W \times 4T$ DITVA

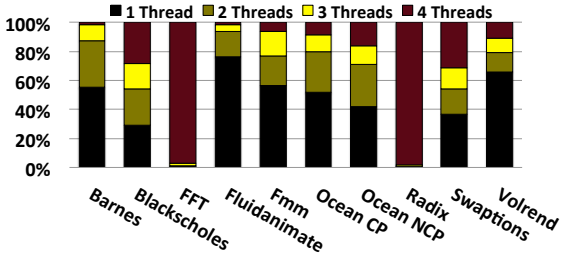


Figure 8: Breakdown of average DV-instruction occupancy for the $4W \times 4T$ configuration

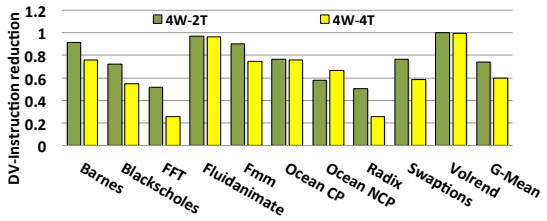


Figure 9: DV-instruction count reduction over 4-thread SMT as a function of warp size

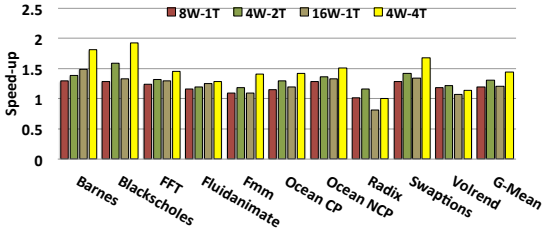


Figure 10: Speed-up over 4-thread SMT as a function of warp size

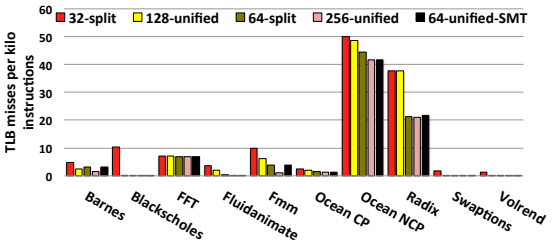


Figure 11: TLB misses per thousand instructions for split or unified TLBs on $4W \times 4T$ DITVA

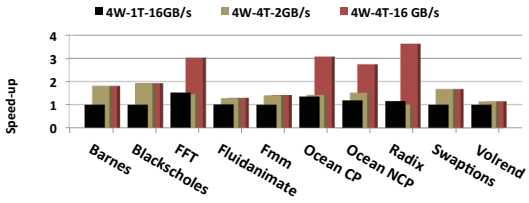


Figure 12: Performance scaling with memory bandwidth, relative to 4-thread SMT with 2 GB/s DRAM bandwidth

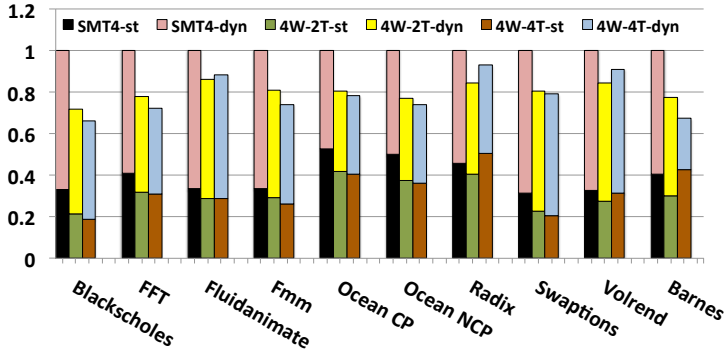
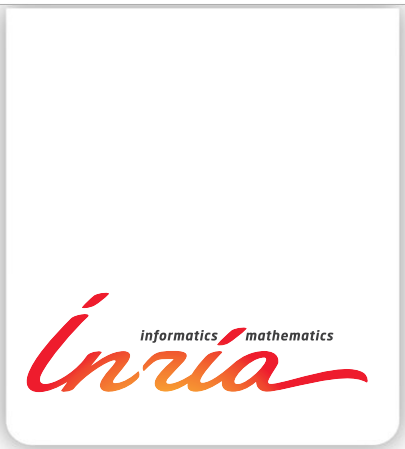


Figure 13: Relative energy consumption: base 4-thread SMT.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399