HAL
open science

Transforming TLP into DLP with the dynamic inter-thread vectorization architecture

Sajith Kalathingal

To cite this version:

Sajith Kalathingal. Transforming TLP into DLP with the dynamic inter-thread vectorization architecture. Hardware Architecture [cs.AR]. Université de Rennes, 2016. English. NNT : 2016REN1S133 . tel-01426915v3

HAL Id: tel-01426915

https://theses.hal.science/tel-01426915v3

Submitted on 28 Aug 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANNÉE 2016

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Sajith Kalathingal

préparée à l'unité de recherche INRIA
Institut National de Recherche en Informatique et Automatique
Université de Rennes 1

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

Thèse soutenue à Rennes
le 13 Décembre 2016

devant le jury composé de :

Bernard GOOSSENS
Professeur à l'Université de Perpignan Via Domitia / Rapporteur

Smail NIAR
Professeur à l'Université de Valenciennes / Rapporteur

Laure GONNORD
Maître de conférences à l'Université Lyon 1 / Examinatrice

Cédric TEDESCHI
Maître de conférences à l'Université Rennes 1 / Examinateur

André SEZNEC
Directeur de recherches Inria / Directeur de thése

Sylvain COLLANGE
Chargé de recherche INRIA / Co-directeur de thése

Acknowledgement

I would like to express my sincere gratitude to my thesis advisors, André SEZNEC and Sylvain COLLANGE. I would like to thank you for encouraging me and providing guidance during the course of work. Your advice on research have been priceless.

I would like to thank the jury members Bernard GOOSSENS, Smail NIAR, Laure GONNORD and Cédric TEDESCHI for providing the opportunity to defend the thesis.

I would like to thank my parents for supporting me throughout my life. Thank you for all of the sacrifices that you have made for me. Without your support I would not have reached so far.

I would like to thank the members of ALF team for their help and support and also for making my stay at Rennes really enjoyable.

I would like to thank my wonderful wife Amrutha Muralidharan for the moral support she has given me from thousands of kilometers away, in India. Thank you motivating me and for having the patience.

Contents

Table of Contents 1

Résumé en Français 5

Introduction 9

1 Background 15

1.1 Instruction-level parallelism 18

1.1.1 Instruction pipelining 20

1.1.2 Superscalar execution 21

1.1.3 Out-of-order execution 22

1.1.4 Clustering for ILP 23

1.2 Data-level parallelism 23

1.2.1 Single Instruction Multiple Data 24

1.2.2 Single Instruction Multiple Threads (SIMT) 27

1.3 Thread-level parallelism 30

1.3.1 Coarse-grained multi-threading 31

1.3.2 Fine grained multi-threading 31

1.3.3 Simultaneous multi-threading (SMT) 32

1.3.4 Clustered multi-threading (CMT) 33

1.4 TLP vs DLP 35
1.5 Chip multiprocessor 36
1.6 Conclusion 36

2 Exploiting inter-thread similarity in microprocessors 39
2.1 Inter-thread similarities in SPMD applications 39
2.1.1 Source of inter-thread similarities 40
2.1.2 Vectorization efficiency in an SMT 42
2.1.3 Dynamic Vectorization 43
2.2 Thread reconvergence for SPMD applications 44
2.2.1 Stack-based explicit reconvergence 45
2.2.2 SIMT extensions for stack-based implicit reconvergence . . 45
2.2.3 Stack-less explicit reconvergence 46
2.2.4 Stack-less implicit reconvergence 47
2.3 Vectorization of instructions across threads 49
2.4 General purpose architectures exploiting inter-thread redundancy 50
2.5 GPU architectures to exploit inter-thread redundancies 51
2.6 Conclusion 52

3 Dynamic Inter-Thread Vectorization Architecture 55
3.1 Warps in DITVA 56
3.2 Overview of the Dynamic Inter-Thread Vectorization Architecture 58
3.3 Pipeline architecture 60
3.3.1 Front-end 60
3.3.2 In-order issue enforcement and dependency check 63
3.3.3 Execution: register file and functional units 66
3.3.4 Leveraging explicit SIMD instructions 68
3.3.5 Handling misprediction, exception or divergence 69

3.4 Data memory accesses 70
3.5 Maintaining lockstep execution 71
3.6 Clustered Multi-threading in DITVA 72
3.7 Conclusion 73

4 Evaluation 77
4.1 Experimental Framework 77
4.1.1 DITVA-SIM 77
4.1.2 Evaluations using DITVA-SIM 79
4.2 Performance evaluation 81
4.2.1 Throughput 81
4.2.2 Divergence and mispredictions 82
4.2.3 Impact of split data TLB 83
4.2.4 L1 cache bank conflict reduction 83
4.2.5 Impact of memory bandwidth on memory intensive applications 84
4.2.6 Impact of Warp size 86
4.2.7 Impact of banked DV-SIMD register banking 86
4.3 Hardware Overhead, Power and Energy 87
4.3.1 Qualitative evaluation 87
4.3.2 Quantitative evaluation 89
4.4 Conclusion 90

5 Future work 93
5.1 Out-of-order execution 93
5.1.1 Register renaming 94
5.1.2 Reservation station / Issue queue 95
5.1.3 Reorder buffer 95

5.1.4 Branch misprediction 95
5.2 Out-of-order extension for DITVA architecture 96
5.2.1 Register management 97
5.2.1.1 Register allocation 98
5.2.1.2 Register deallocation 99
5.2.1.3 Handling divergence and reconvergence 99
5.2.2 Handling branch misprediction 100
5.3 Conclusion 100

Bibliography 119

List of figures 121

Résumé en Français

Dans cette thèse, nous proposons l'architecture Dynamic Inter-Thread Vectorization (DITVA), une technique pour améliorer les performances des applications multi-thread SPMD dans un microprocesseur généraliste. Les threads d'applications SPMD exécutent souvent les mêmes instructions sur des données différentes. Pour tirer parti de la redondance de contrôle dans les applications SPMD, DITVA assemble dynamiquement des instructions identiques de plusieurs threads en cours d'exécution en une seule instruction vectorielle au moment de l'exécution. La vectorisation réduit le nombre d'opérations dans le pipeline, car l'instruction vectorisée factorise le contrôle entre les threads en travaillant sur des données différentes. DITVA étend un processeur SMT disposant d'instructions SIMD avec un mode d'exécution de vectorisation inter-threads. Dans ce mode, DITVA exploite les unités vectorielles existantes, améliorant ainsi l'utilisation des capacités vectorielles des microprocesseurs existants. DITVA maintient la compatibilité binaire avec les architectures CPU existantes. Grâce à l'utilisation des unités vectorielles et à la réduction des opérations de pipeline, DITVA vise à améliorer le débit d'exécution d'une puce microprocesseur x86_64 tout en réduisant sa consommation globale d'énergie.

Limitations au niveau transistor d'un microprocesseur La technologie des microprocesseurs a beaucoup évolué depuis l'introduction du premier microprocesseur Intel 4004 en 1971 par Intel. L'Intel 4004 était réalisé avec une finesse de gravure de 10 μm avec seulement 2300 transistors. Le processeur Intel Skylake, mis sur le marché en 2015, emploie un process de 14nm avec près de 1,9 milliards de transistors. Gordon E. Moore a observé que le nombre de transistors dans un circuit intégré double chaque année. [M$^+$98]

Robert H. Dennard a observé qu'avec la réduction de taille des transistors,

la puissance totale requise reste constante [DRBL74]. La loi de Moore associée au passage à l'échelle de Dennard se traduit par une croissance exponentielle de la performance par watt. La loi de Dennard a pris fin avec l'augmentation de la finesse de gravure. Les courants de fuite et la dissipation de chaleur ont commencé à devenir un réel problème pour le passage à l'échelle de la performance au travers le l'augmentation des fréquences d'horloge. Par conséquent, les fréquences d'horloge ont commencé à se stabiliser après le milieu des années 2000, malgré l'adoption de différentes technologies de transistors.

Techniques micro-architecturales pour améliorer la performance Alors que l'augmentation des fréquences d'horloge est une façon d'améliorer les performances des microprocesseurs, la micro-architecture elle-même joue un rôle important dans l'optimisation des microprocesseurs pour le débit, la latence et l'efficacité énergétique. Suivant ces exigences, les fabricants de processeurs doivent souvent assurer des compromis lors de la conception d'un processeur à usage généraliste. Par exemple, un processeur mobile, qui utilise des petits cœurs est optimisé pour l'énergie aux dépens du débit, tandis qu'un GPU est optimisé pour le débit au détriment de la latence.

Avec l'évolutivité limitée des fréquences d'horloge, le multiprocesseur sur puce (CMP), c'est-à-dire un multi-cœur [ONH+96], a été largement adopté au cours des dix dernières années. Au lieu d'un grand cœur monolithique, un CMP se compose de plusieurs cœurs dans une seule puce qui partagent certaines des ressources matérielles telles que des caches. Une application mono-thread est exécutée dans un cœur unique du CMP tandis qu'une application parallèle peut utiliser plusieurs cœurs grâce à un faible coût de communication entre les processeurs. Les processeurs multi-core hétérogènes [KTJR05] sont également de plus en plus répandus. Pour conserver un meilleur compromis entre consommation d'énergie et performance, une puce peut être constituée de plusieurs cœurs grands et petits pouvant être allumés et éteints, la charge de travail pouvant être migrée en fonction des exigences de l'application [PG13].

Les grands cœurs utilisent des techniques telles que l'exécution spéculative, l'exécution superscalaire, l'exécution dans le désordre, etc., pour améliorer la performance d'un thread unique. La performance mono-thread est limitée par le parallélisme d'instructions (ILP) d'une application. Les caractéristiques micro-architecturales peuvent améliorer l'ILP mais sont encore limitées par les propriétés inhérentes de l'application. Avec un ILP limité, l'utilisation des ressources

d'un cœur diminue. Le multi-threading a été mis en œuvre pour améliorer l'utilisation des ressources d'un cœur avec un impact minimal sur la latence mono-thread [TEL95, Upt, SBB+07, BEKK00]. Les petits cœurs évitent la plupart de ces optimisations pour améliorer l'efficacité énergétique.

Microarchitecture pour application extrêmement parallèle Il existe de nombreuses catégories d'applications extrêmement parallèles, orientées sur le débit et tolérantes à la latence. Les GPU, introduits en 1999, sont plus efficaces dans l'exécution de ces applications [NBGS08, KWm12, LNOM08, ND10a]. Avec un grand nombre d'applications de jeu et multimédia, les GPU sont devenus populaires dans les années 2000. L'architecture Tesla de NVIDIA a été introduite en 2006 et permet notamment l'exécution d'applications de calcul parallèle haute performance écrites en langage C à l'aide de l'architecture CUDA pour le calcul généraliste sur GPU. Les GPU ne sont pas des dispositifs de calcul autonomes. Ils sont conçus pour compléter le CPU existant pour améliorer l'efficacité de l'exécution d'application hautement parallèles. Le CPU se décharge de ce travail vers un GPU. D'autre part, un GPU ne peut pas exécuter les processus critiques nécessaires pour exécuter un système d'exploitation. Les CPU et GPU ont différentes philosophies de conception architecturale, le problème qu'ils essaient de résoudre étant différent. Ils mettent en œuvre différents jeux d'instructions (ISA).

DITVA - une optimisation pour les architectures SMT généralistes Dans cette thèse, nous proposons l'architecture de vectorisation dynamique inter-thread (DITVA), un point de conception intermédiaire entre un CPU et un GPU. DITVA est une optimisation pour les applications parallèles s'exécutant sur un CPU tout en maintenant la performance mono-thread. DITVA optimise les threads d'application en vectorisant dynamiquement les instructions entre threads. DITVA est basé sur l'architecture x86_64 et supporte les applications régulières exécutées dans un CPU. DITVA peut être réalisé comme un cœur spécial dans une puce CMP (multi-core) hybride pour optimiser l'exécution des applications parallèles. DITVA n'a pas besoin de modifications dans l'ISA x86_64 existante. Les binaires existants peuvent être exécutés dans un cœur DITVA, sans nécessiter pas de recompilation. DITVA améliore le débit des cœurs existants tout en consommant moins d'énergie.

Introduction

In this thesis, we propose Dynamic Inter-Thread Vectorization Architecture (DITVA), a technique to improve the performance of multi-threaded Single-Program Multiple-Data (SPMD) applications in a general purpose microprocessor. Threads of SPMD applications often execute the same instructions on different data. To take advantage of the control redundancy in SPMD applications, DITVA dynamically aggregates identical instructions of several threads running in lockstep into a single *vector instruction* at runtime. Vectorization reduces the number of pipeline operations as a vectorized instruction shares the control across threads while operating on multiple data. DITVA extends a vector enabled in-order SMT processor with an inter-thread vectorization execution mode. In this mode, DITVA leverages existing SIMD units, thus improving the utilization of the *vector* capabilities of the existing microprocessors. DITVA maintains binary compatibility with existing CPU architectures. With the improved utilization of SIMD units and the reduction of pipeline operations, DITVA intends to improve the throughput of an x86_64 microprocessor chip while reducing the overall energy consumption.

Limitations at the transistor-level of a microprocessor Microprocessor technology has evolved a lot since the introduction of the first microprocessor Intel 4004 in 1971 by Intel Corp. Intel 4004 was a 10 μm process with just 2,300 transistors. Skylake processors, released by Intel in 2015, has a 14nm process with almost 1.9 billion transistors. Gordon E. Moore observed that the number of transistors in an integrated circuit doubles every year [M$^+$98]. The projection of process technology shrinkage remained accurate for several decades, but it is predicted that this will not be true in the near future [Moo].

Robert H. Dennard observed that with the scaling down of transistor size, the

required power stays constant [DRBL74]. Moore's Law coupled with Dennard scaling translates to exponential growth of performance-per-watt. Dennard's law began to fail when the transistors started to become too small. Current leakage and heat dissipation began to become a real issue for performance scaling by increasing clock rates. Hence, the clock rates started to plateau after the mid-2000s even after adopting different transistor designs.

Microarchitectural techniques to improve performance While increasing the clock rates is one way to improve the performance of the microprocessor, microarchitecture itself plays a significant role in optimizing microprocessors for throughput, latency and energy efficiency. Depending on the requirements, the processor manufacturers often have to balance the trade-offs when designing a general purpose processor. For example, a mobile processor, which uses *small cores* is optimized for energy at the cost of throughput, while a GPU is optimized for throughput at the expense of latency [1].

With the limited scalability concerning clock rates, Chip Multi-Processor(CMP), i.e., multi-core processors [ONH+96] are widely adopted in the past decade as an alternative design direction. Instead of a large monolithic core, a CMP consists of multiple cores in a single chip. The cores in a CMP shares some of the hardware resources such as caches. A single threaded application is executed on a single core of the CMP, and a parallel application can utilize multiple cores thanks to low inter-processor communication overhead. Heterogeneous multi-core processors [KTJR05] are also becoming popular. To maintain a better performance-energy profile, a chip may consist of multiple *large* and *small* cores that may be turned on and off or the workload can be migrated depending on the application requirements [PG13].

Large cores use techniques such as instruction pipelining, speculative execution, superscalar execution, out-of-order execution, etc., to improve the single thread performance. Single thread performance is bounded by the Instruction-level parallelism (ILP) of an application. Microarchitectural features can improve the ILP but are still limited by the inherent properties of the application. With

[1] Throughput is the number of instructions completed in a cycle and latency is the number of cycles taken to complete an instruction. In a single threaded microprocessor, latency is inversely proportional to throughput. However, in a multi-threaded processor, even though there are more instructions completed in a cycle, a thread could still slow down since it is sharing the resources with other threads

limited ILP, the resource utilization of a core goes down. As an afterthought multi-threading [TEL95, Upt, SBB+07, BEKK00] was implemented to improve the resource utilization of a core with a slight impact on single thread latency. *Small cores* avoid many of these optimizations to improve energy profile.

Microarchitecture for embarrassingly parallel applications There are broad classes of *embarrassingly parallel*, throughput oriented, latency-tolerant applications. GPUs, introduced in 1999, are more efficient in executing these applications [NBGS08, KWm12, LNOM08, ND10a]. With a large number of gaming and multimedia applications, GPUs became popular in 2000's. NVIDIA's Tesla architecture was introduced in 2006 to extend GPUs to enable the execution of high-performance parallel computing application written in C language using the Compute Unified Device Architecture(CUDA) for general purpose computing on GPUs. GPUs are not stand-alone computational devices. They are designed to compliment the existing CPU to improve the efficiency of executing high-performance, embarrassingly parallel applications. CPU's offload such workload to a GPU. On the other hand, GPUs cannot run critical processes required to run an operating system. CPU and GPU have different architectural design philosophies as the problem they are trying to solve is different. They implement different Instruction Set Architecture(ISA).

DITVA - an optimization for general purpose SMT architectures
SMT [TEL95] is a common implementation of multi-threading in modern microprocessors. In SMT, multiple thread context co-exists in a single core of a microprocessor. The threads in an SMT are scheduled simultaneously so that when one of the thread is blocked, some other thread can execute the instructions. SMT processors, unlike GPUs, are designed to execute both single threaded as well as multi-threaded applications. While SMT processors are beneficial for both single and multi-threaded applications, however, they fail to benefit from the unique property of embarrassingly parallel applications i.e., the instructions across the threads are mostly identical although they work with different data. DITVA architecture is designed to exploit this property that exists in parallel applications. By doing so, DITVA improves the throughput of the existing cores while consuming significantly less energy.

In this thesis, we propose Dynamic Inter-Thread Vectorization Architecture (DITVA), an intermediate design point between a CPU and a GPU. DITVA is

a hardware optimization for parallel applications running on a CPU while maintaining the single thread performance. DITVA optimizes the application threads by dynamically vectorizing instructions across threads. We evaluate DITVA on a x86_64 architecture and show that it supports the regular applications executed in a CPU. DITVA may be implemented as special cores in a hybrid CMP (multi-core) chip to optimize the execution of parallel applications. DITVA does not need any modifications in the existing x86_64 ISA. Existing binaries can be executed in a DITVA core as it does not require recompilation of the program with compiler hints.

Contributions

We developed a microarchitectural model, DITVA, to leverage the implicit data-level parallelism in SPMD applications by assembling dynamic vector instructions at runtime. DITVA extends an SIMD-enabled in-order SMT processor with an inter-thread vectorization execution mode. In this mode, multiple scalar threads running in lockstep share a single instruction stream and their respective instruction instances are aggregated into SIMD instructions. To balance thread and data-level parallelism, threads are statically grouped into fixed-size independently scheduled warps. DITVA leverages existing SIMD units and maintains binary compatibility with existing CPU architectures.

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

Organization

The thesis is organized as follows: Chapter 1 provides background for microarchitectural techniques used to exploit parallelism in a program. In chapter 2, we discuss inter-thread similarities and the techniques used to exploit them. We

also discuss various thread reconvergence techniques that enables dynamic vectorization of instruction across threads. In chapter 3, we present the Dynamic Inter-Thread Architecture (DITVA). We discuss the implementation details and the design aspects of DITVA. In chapter 4, we demonstrate the energy/performance benefits of DITVA over an SMT architecture. We show that DITVA is able to achieve this without significant hardware overhead. In chapter 5, we provide some insights for an out-of-order version of DITVA (OOO-DITVA). Materials from chapter 3 and chapter 4 were presented at SBAC-PAD 2016 [KCSS16].

Chapter 1

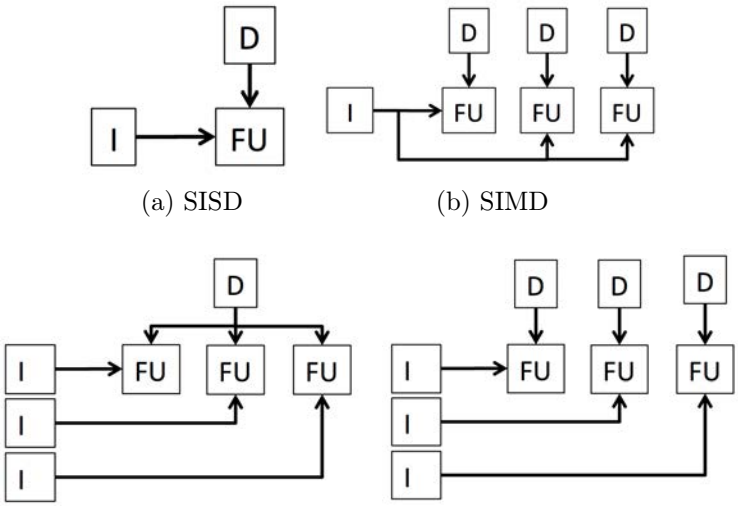
Background

Implementation level	DLP	TLP
Program	Vector	Threads
Compiler	Auto-vectorization	Auto-parallelization
Hardware	SIMD	MIMD

Figure 1.1: Source of parallelism in a program

In this chapter, we will discuss different kinds of parallelism that exist in a program, i.e., Instruction-level parallelism (ILP, section 1.1), Data-level parallelism (DLP, section 1.2) and Thread-level parallelism (TLP, section 1.3) and provides the background for the transformation of TLP to DLP using DITVA. We will also discuss different methods utilized by the current microarchitectures to exploit these parallelisms. Amdahl's law [Amd67] identifies the serial section of a program as the limiting factor in the speedup of a workload. The implicit and explicit parallelism in a program is exposed at the program level, compiler level or microarchitectural level. The source of TLP and DLP in a conventional program execution is shown in figure 1.1. DLP is used to form vector instructions, and TLP is used to create program threads. At the program level, vector instructions and parallel threads expose parallelism using programming intrinsics and algorithms that use programming models based on the data decomposition [Fos95]. At the

compiler level, automatic vectorization and automatic parallelization expose parallelism of the program. At the microarchitectural level, execution models based on Flynn's taxonomy [Fly66, Fly72] are used to exploit parallelism. Based on the instruction control and the number of data it is working with, Flynn's taxonomy [Fly66, Fly72] classifies the execution model of computer architecture into SISD (Single instruction stream, single data stream), MISD (Multiple instruction stream, single data stream), SIMD (Single instruction stream, multiple data stream) and MIMD (Multiple instruction stream, multiple data stream).



(a) SISD (b) SIMD

(c) MISD (d) MIMD

Figure 1.2: Flynn's taxonomy and MISD model

Flynn's classification	1 thread	> 1 thread
SISD	Scalar processor	Multi-thread
SIMD	Vector Processor	DITVA
MIMD	-	Multi-core

Figure 1.3: Classification of architectures based in thread count

Diagrams representing Flynn's taxonomy is shown in figure 1.2. Multi-

threaded architectures add another dimension to Flynn's taxonomy. For the sake of clarity, we can define the stream as a sequence of instructions that is mapped to a thread at the program level.[1] Some example architectures based on the number of streams in shown in figure 1.3. Traditional scalar uniprocessor architectures were SISD. A single threaded superscalar processor 1.1.2 is also classified as an SISD. SISD with more than one thread is a multi-threaded architecture. SIMD machines exploit data parallelism by using a single instruction stream to work on multiple data elements. Vector processors used for high-performance computing are SIMD architectures. Vector architectures are based on explicit vector instructions. MIMD consists of multiple processors that may be executing different instruction streams that are working with different data streams. A multi-core processor is primarily an MIMD architecture but many modern x86_64 architectures is a combination of different models, they support the execution of scalar instructions (SISD) and with SSE and AVX instruction sets they support the execution of vector instructions (SIMD). In addition to threads and vector instructions, microarchitecture techniques also exploit ILP from within the same thread.

SIMD architectures such as vector processors, GPUs, and CPUs with SIMD extensions, use vector instructions generated either by compiler or programmer. A large number of applications that are executed in x86_64 architectures does not efficiently utilize SIMD extensions even though the programs have data-parallelism (see section 2.5). The inefficiency is due to the existence of dynamic properties such as branches in the instruction flow. Such programs are often written using threads running the same code. Many multi-threaded programs have implicit data-parallelism.

In the following sections, we will discuss implicit and explicit techniques that are used to exploit parallelism in a program. The parallelism in SISD is the instruction-level parallelism, data-level parallelism is exploited by the SIMD model and MIMD is exposed by the thread-level parallelism. In this thesis, we will only consider single process model and hence we will not consider parallel programming with multiple processes.

[1] In section 3.2, we will redefine the notion of instruction stream as a sequence of instructions that is mapped to one or more threads at the program level, to adapt it for DITVA.

Listing 1.1: Array addition

```

void add() {
        for(int i = 0 ; i < N; i++){
                c_array[i] = a_array[i] + b_array[i];
                f_array[i] = d_array[i] + e_array[i];
        }
}
```

1.1 Instruction-level parallelism

Compiler techniques and programming constructions are used to exploit implicit parallelism between instructions of a sequential program. Several microarchitectural features efficiently utilize this parallelism and often enhance it at runtime. *Hazards* are factors limiting parallelism. Two of the most common hazards are *Control hazard* and *Data hazard*.

Control hazards occur with branches. A branch is an instruction which alters the sequence of execution based on a condition. Branches are often resolved at runtime and hence the instruction fetch should not be stalled until branch resolution. Modern branch predictors such as TAGE [Sez11] are highly accurate. Microprocessors use branch predictors to continue fetching from one of the predicted branches. In the case of a misprediction, the instructions that are already in the pipeline are flushed.

Data hazards occur when an instruction with data dependency changes the order of read/write operation. Data hazards can be classified as follows

1. Read after write (RAW)
2. Write after read (WAR)
3. Write after write (WAW)

RAW is a true dependency and cannot be circumvented as the instruction which is reading the value should wait until the previous instruction produces the result. WAR and WAW are false dependencies and can be prevented using microarchitectural techniques.

Listing 1.2: Array addition (ASM)

```
100000f15:  lea     0xe4(%rip),%rax    # 100001000 <_N>
100000f1c:  cmpl    $0x0,(%rax)
100000f1f:  jle     100000f6f <_add+0x5f>
100000f21:  movslq  (%rax),%rax
100000f24:  xor     %edx,%edx
100000f26:  lea     0xe3(%rip),%r8
# 100001010 <_a_array>
100000f2d:  lea     0x10c(%rip),%r9
# 100001040 <_b_array>
100000f34:  lea     0x135(%rip),%r10
# 100001070 <_c_array>
100000f3b:  lea     0x15e(%rip),%r11
# 1000010a0 <_d_array>
100000f42:  lea     0x187(%rip),%rcx
# 1000010d0 <_e_array>
100000f49:  lea     0x1b0(%rip),%rsi
# 100001100 <_f_array>
100000f50:  mov     (%r9,%rdx,4),%edi
100000f54:  mov     (%rcx,%rdx,4),%ebx
100000f57:  add     (%r8,%rdx,4),%edi
100000f5b:  add     (%r11,%rdx,4),%ebx
100000f5f:  mov     %edi,(%r10,%rdx,4)
100000f63:  mov     %ebx,(%rsi,%rdx,4)
100000f66:  lea     0x1(%rdx),%rdx
100000f6a:  cmp     %rax,%rdx
100000f6d:  jl      100000f50 <_add+0x40>
```

Consider the pseudocode in list 1.1 for array addition and its corresponding assembly code in list 1.2 produced after compilation. When the compiled code is executed in a scalar microprocessor with a single execution unit, the instructions are sequentially executed as shown in the figure. However, in practice, a program in execution have a large amount of implicit parallelisms called *Instruction-Level Parallelism (ILP)*. In listing 1.1, the statements in line 4 and line 5 are independent. Hence, the instructions corresponding to these statements can be executed in parallel. In listing 1.2, we find that the instructions from line 6 to line 11 are parallel. Similarly, inside the loop statements, instructions in line (12, 14, 16) and (13, 15, 17) can be executed in parallel.

Microarchitectural techniques such as instruction pipelining, speculative execution, superscalar execution, and out-of-order execution are used to exploit the ILP. Our DITVA proposal supports these microarchitectural techniques except out-of-order execution [2].

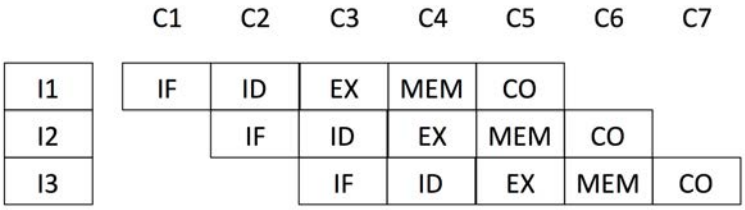


Figure 1.4: Instruction pipelining

1.1.1 Instruction pipelining

Instruction pipelining is a process an instruction technique in which the instruction processing is done in multiple stages and rather than waiting for the entire instruction processing to complete, subsequent instructions are processed concurrently. Figure 1.4 shows a simple 5 stage pipeline execution of instruction I1, I2, and I3. In the example, each stage has a single cycle latency. Instruction I2 starts fetch at cycle 2 as soon as instruction I1 completes the fetch stage. Similarly, I3 starts fetch at cycle 3. A non-pipelined execution would have taken 15

[2]DITVA can be extended to support out-of-order execution. We provide some insights for an out-of-order implementation in Chapter 5

(3×5) cycles to complete the execution of 3 instructions, with pipelining, the total latency reduces to 7 cycles.

	C1	C2	C3	C4	C5	C6
I1	IF	ID	EX	MEM	CO	
I2	IF	ID	EX	MEM	CO	
I3		IF	ID	EX	MEM	CO
I4		IF	ID	EX	MEM	CO

Figure 1.5: Superscalar execution (degree 2)

1.1.2 Superscalar execution

In superscalar processors, more than one instructions are processed at the same clock cycle. In Flynn's taxonomy, superscalar processors are classified as SISD. A superscalar processor has replicated pipelined resources that allow multiple fetches, decodes, executions and writebacks per cycle. Superscalar execution of degree 2 is shown in figure 1.5. A superscalar processor improves the execution throughput by issuing independent instructions in the replicated functional units. An in-order processor issues the instructions in the order of fetch. Hence, the adjacency of the independent instructions becomes the limiting factor in the scalability of a superscalar processor. To illustrate this limitation, let us consider listing 1.3, which is hypothetical reordered instruction sequence of listing 1.1. In listing 1.2 the instructions corresponding to *c_array[i] = a_array[i] + b_array[i];* and *f_array[i] = d_array[i] + e_array[i];* are interleaved to increase ILP. In listing 1.3, the interleaving is removed. And consequently, line 14 and 15 are the only instructions inside the loop which can be executed in parallel in an in-order machine. Even though modern compilers are smart enough to reorder the instructions to improve the ILP, the parallelism is limited by the program behavior. The added hardware remains idle when a superscalar machine executes a program with limited ILP. Considering the high cost of replicating the pipeline units, increase in hardware have diminishing returns in terms of performance [PJS97, KHL99].

Listing 1.3: Array addition (ASM) - Reordered

```
1     100000f15:  lea     0xe4(%rip),%rax    # 100001000 <_N>
2     100000f1c:  cmpl    $0x0,(%rax)
3     100000f1f:  jle     100000f6f <_add+0x5f>
4     100000f21:  movslq  (%rax),%rax
5     100000f24:  xor     %edx,%edx
6     100000f26:  lea     0xe3(%rip),%r8
  # 100001010 <_a_array>
7     100000f2d:  lea     0x10c(%rip),%r9
  # 100001040 <_b_array>
8     100000f34:  lea     0x135(%rip),%r10
  # 100001070 <_c_array>
9     100000f3b:  lea     0x15e(%rip),%r11
  # 1000010a0 <_d_array>
10    100000f42:  lea     0x187(%rip),%rcx
  # 1000010d0 <_e_array>
11    100000f49:  lea     0x1b0(%rip),%rsi
  # 100001100 <_f_array>
12    100000f50:  mov     (%r9,%rdx,4),%edi
13    100000f54:  add     (%r8,%rdx,4),%edi
14    100000f58:  mov     %edi,(%r10,%rdx,4)
15    100000f5c:  mov     (%rcx,%rdx,4),%ebx
16    100000f60:  add     (%r11,%rdx,4),%ebx
17    100000f64:  mov     %ebx,(%rsi,%rdx,4)
18    100000f68:  lea     0x1(%rdx),%rdx
19    100000f6c:  cmp     %rax,%rdx
20    100000f70:  jl      100000f50 <_add+0x40>
```

1.1.3 Out-of-order execution

Out-of-order processors improve the execution throughput by not strictly issuing the instructions in the order of fetch. For example, in listing 1.3, instructions in line 13 and line 16 can be issued in parallel as they are no longer required to strictly conform to the order. Out-of-order processors use techniques such as scoreboarding [BRI+90] to resolve the dependencies. In out-of-order processors instruction-level parallelism (ILP) is often limited by data hazards. As men-

tioned earlier, false dependencies can be avoided. False dependencies occur because of the limited number of registers provided by the ISA. This limitation forces the compiler to reuse registers in an instruction sequence that often result in producing false dependencies. Out-of-order processors uses register renaming [Tom67, Kes99, MPV93] to prevent false dependencies.

1.1.4 Clustering for ILP

Clustering allows the implementation of wide superscalar processors without compromising much on the clock rate. Essentially, clustering partitions a wide superscalar processor into multiple windows of smaller width. This reduces the delays in the critical path at various stages of the pipeline. Clustering can be done at different stages of the pipeline. [PJS97] study the complexity and delay characteristics of increasing the width of the superscalar processor. [CPG00, BM00, AF01, BDA03] discuss various techniques for clustering in superscalar processors. [Kes99] is one of the earliest implementation of a single threaded clustered micro-architecture that clusters the functional units.

1.2 Data-level parallelism

When a task operates on different pieces of data Data-level parallelism (DLP) arises. Data parallel algorithms perform independent operations on a large set of data and share the control flow. Many scientific applications, as well as multimedia applications, have DLP. DLP is exploited with the SIMD model of Flynn's taxonomy. Vectorization exposes DLP and SIMD/vector architectures are used to exploit them. Auto-vectorization [Nai04, NZ06, gcc, icc, MLN07] is done by many modern compilers for high-level programming languages such as C, C++, Fortran etc. Auto-vectorization compilers try to identify the Data parallel regions in the program code and vectorize them. Alternatively, the programmer rely on *intrinsics* [int] for vectorization efficiency. *Intrinsics* are often difficult to write and affects the portability. A relatively easier approach is SPMD-based programming environments such as CUDA, OpenCL, and OpenMP [KDT+12, DM98]. ISPC [PM12] is a compiler specifically aiming to exploit data parallelism in CPU.

1.2.1 Single Instruction Multiple Data

Microarchitectures such as vector processors and Graphics Processor Units (GPUs) exploit vectorization by using vector instructions. SIMD architectures optimize microprocessors for power efficiency by reducing the number of instruction operations at the front-end of the pipeline. x86 architectures implement SIMD exposed through MMX, SSE, and AVX [FBJ+08] instruction sets. In the Intel's implementation of SIMD, in addition to the supported instructions, MMX, SSE and AVX support varying data paths. MMX implementation reused the existing floating point registers for the 64-bit SIMD instructions. SSE implementation uses xmm registers with 128-bit wide data path [SKR00]. AVX instructions further widened the data path from 128-bit to 256-bit ymm registers (AVX-512 [R13], with zmm registers, supports 512-bit datapath) [3]. SIMD instructions may support different data types. For instance, SSE2 supports two 64-bit double-precision floating point numbers or two 64-bit integers or four 32-bit integers or eight 16-bit short integers or sixteen 8-bit characters.

Listing 1.4: Array addition (ASM) - SIMD

```
1     100000db4: lea     0x245(%rip),%rax
  # 100001000 <_N>
2     100000dbb: movslq (%rax),%r9
3     100000dbe: test    %r9,%r9
4     100000dc1: jle     100000e9a <_add+0xea>
5     100000dc7: xor     %eax,%eax
6     100000dc9: test    %r9d,%r9d
7     100000dcc: je      100000e50 <_add+0xa0>
8     100000dd2: xor     %eax,%eax
9     100000dd4: mov     %r9,%r8
10    100000dd7: and     $0xfffffffffffffffe,%r8
11    100000ddb: je      100000e4a <_add+0x9a>
12    100000ddd: mov     %r9,%r10
13    100000de0: and     $0xfffffffffffffffe,%r10
14    100000de4: lea     0x225(%rip),%r11
  # 100001010 <_a_array>
```

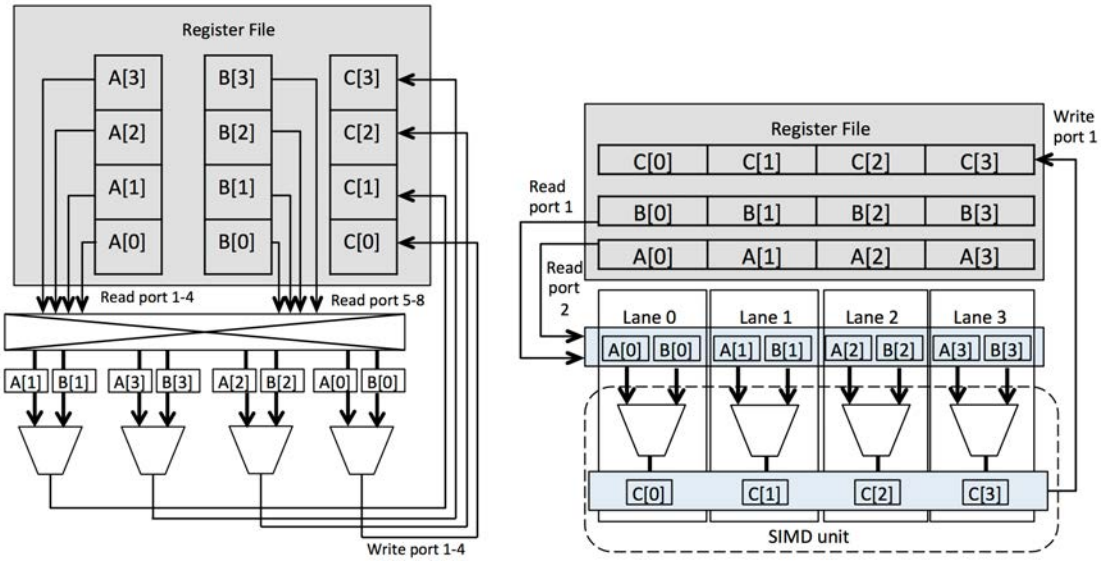
[3]Note that scalar double precision instructions in x86_64 architecture use half of the register width of xmm registers and 1/4 width of ymm registers [R14]. Similarly, SSE instructions are implemented using half of the register width of ymm registers.

```
    100000deb: lea     0x26e(%rip),%rdi
# 100001060 <_b_array>
    100000df2: lea     0x2b7(%rip),%rcx
# 1000010b0 <_c_array>
    100000df9: lea     0x300(%rip),%rax
# 100001100 <_d_array>
    100000e00: lea     0x349(%rip),%rdx
# 100001150 <_e_array>
    100000e07: lea     0x392(%rip),%rsi
# 1000011a0 <_f_array>
    100000e0e: xchg    %ax,%ax
    100000e10: vmovdqa (%rdi),%xmm0
    100000e14: vpaddq (%r11),%xmm0,%xmm0
    100000e19: vmovdqa %xmm0,(%rcx)
    100000e1d: vmovdqa (%rdx),%xmm0
    100000e21: vpaddq (%rax),%xmm0,%xmm0
    100000e25: vmovdqa %xmm0,(%rsi)
    100000e29: add     $0x10,%r11
    100000e2d: add     $0x10,%rdi
    100000e31: add     $0x10,%rcx
    100000e35: add     $0x10,%rax
    100000e39: add     $0x10,%rdx
    100000e3d: add     $0x10,%rsi
    100000e41: add     $0xfffffffffffffffe,%r10
    100000e45: jne     100000e10 <_add+0x60>
    100000e47: mov     %r8,%rax
    100000e4a: cmp     %rax,%r9
    100000e4d: je      100000e9a <_add+0xea>
    100000e4f: nop
    100000e50: lea     0x1b9(%rip),%rcx
# 100001010 <_a_array>
    100000e57: lea     0x202(%rip),%rdx
# 100001060 <_b_array>
    100000e5e: mov     (%rdx,%rax,8),%rdx
    100000e62: add     (%rcx,%rax,8),%rdx
    100000e66: lea     0x243(%rip),%rcx
# 1000010b0 <_c_array>
```

```
44      100000e6d: mov     %rdx,(%rcx,%rax,8)
45      100000e71: lea     0x288(%rip),%rcx
    # 100001100 <_d_array>
46      100000e78: lea     0x2d1(%rip),%rdx
    # 100001150 <_e_array>
47      100000e7f: mov     (%rdx,%rax,8),%rdx
48      100000e83: add     (%rcx,%rax,8),%rdx
49      100000e87: lea     0x312(%rip),%rcx
    # 1000011a0 <_f_array>
50      100000e8e: mov     %rdx,(%rcx,%rax,8)
51      100000e92: inc     %rax
52      100000e95: cmp     %r9,%rax
53      100000e98: jl      100000e50 <_add+0xa0>
```

Listing 1.4 is the assembly for the array addition when compiled for SSE architecture. In comparison to listing 1.2, the difference here is that the integer array elements (each element is 4-bytes) are grouped into 4 and are executed as SIMD instructions using 128-bit xmm registers. In listing 1.4, the block of instructions between line 21 and line 37 are the SIMD implementation of the array addition. However, as the array elements are grouped into 4, some of the last elements might not be vectorizable, these array elements are executed by the non-SIMD code block between line 39 and line 53.

SIMD instructions reduce pressure in the front end of a pipeline as there are fewer instructions to be fetched (and consequently fewer instructions in other frontend pipeline stages). For example, the number of loop iterations (and hence the number of fetched instructions) in listing 1.4 is approximately reduced by 4 times. At the backend, for performance reasons, a common implementation of SIMD functional units is by banked register file/wide registers [RDK+00]. However, SIMD is not equivalent to a wide superscalar processor. Figure 1.6 shows the execution of 32-bit integer array of size 4 in a 4-wide superscalar and a SSE unit. SIMD implementation partitions the register and data-path resources into multiple lanes. The reduced design complexity of the pipeline backend improves the scalability. The cost of implementation of SIMD architecture is much lower than a wide superscalar processor [PJS97].



(a) 4-wide superscalar (b) SSE

Figure 1.6: 32-bit integer execution in 4-wide superscalar processor and a SSE unit

1.2.2 Single Instruction Multiple Threads (SIMT)

Single Instruction Multiple Threads (SIMT) [LNOM08] is a parallel execution model that is used to capture implicit DLP from SPMD code. NVIDIA introduced SIMT in their Tesla GPU architecture. SIMT eased programmability of NVIDIA GPU's with the use of Compute Unified Device Architecture (CUDA) [Nvi07] programming libraries. NVIDIA later continued to use SIMT in their Fermi, Kepler, and Maxwell architectures. SIMT is generally combined with multi-threading and multi-core. SIMT relies on a large number of threads to keep the execution resources busy. SIMT works well for throughput oriented applications with large amount of implicit DLP. Hence, the program in execution may consist of a large number of threads. A simplified overview of array execution is shown in figure 1.7. Streaming Multiprocessors(SM) consists of many CUDA cores. An SM executes the SIMD instructions. Threads in CUDA is grouped into thread blocks. Using CUDA, the programmer specifies the number of threads per block and the number of thread blocks. CUDA maps multiple thread blocks to an SM. Hence, many warps are always active in an SM. A thread block is sub-divided

into groups of 32 threads (in our example this number is 4) called warps[4]. In SIMT, the threads in the warp form the lanes of the vectorized instructions.

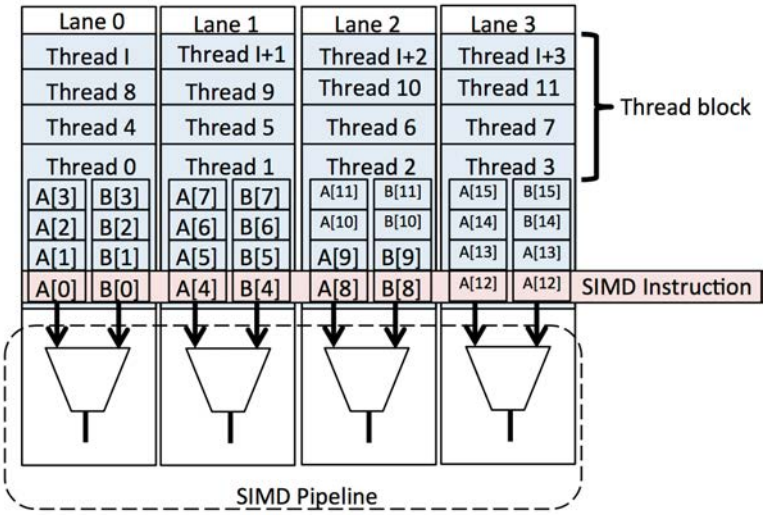


Figure 1.7: SIMT

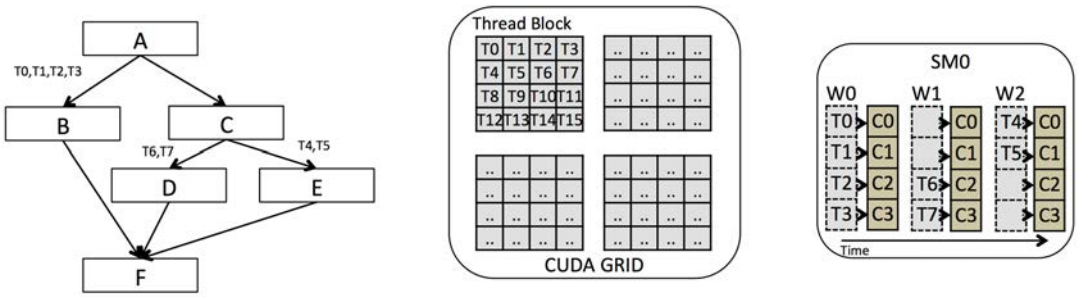


Figure 1.8: SIMT

Figure 1.8 shows a simplified overview of subdivision of a program with divergence into multiple thread blocks, their assignment to an SM and the execution of warps. A *thread* in SIMT is bound to a *lane* in SIMD. Threads in a warp

[4]Thread block is sub-divided into groups of 32 threads because there are 32 lanes (CUDA cores).

that are executing the same instructions form an SIMD instruction. When the threads are not executing the same PC, they are on a divergent path. SIMT uses an active bit mask to identify the participant threads in a vectorized instruction. Figure 1.9 illustrates the masks in SIMT, corresponding to the active threads, when there is a divergence. There is 1 bit corresponding to each thread in an SIMT. A set bit (i.e., 1) indicates that the thread is active and an unset bit (i.e., 0) corresponds to an inactive thread. SIMT can be considered as SIMD instructions with hardware assisted divergence management. Branches in a data-parallel application cause the execution to follow divergent paths. Divergence adversely affects the efficiency of SIMD execution. The divergent paths merge at the reconvergence point. In figure 1.9, block D is a reconvergence point. SIMT requires compiler hints to identify the reconvergence points after the divergence. NVIDIA GPUs uses a stack-based divergence management. Stack-based reconvergence is discussed in section 2.2.1.

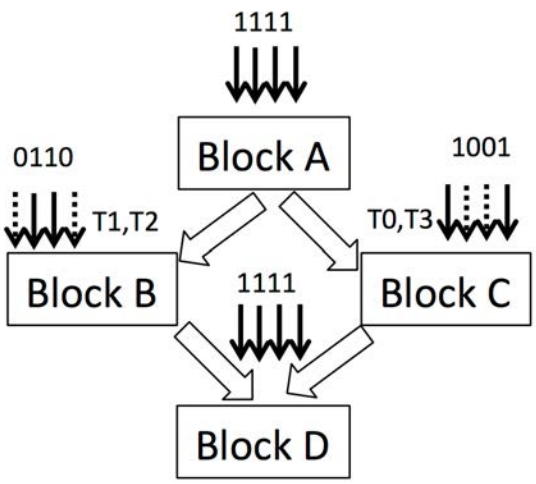


Figure 1.9: SIMT divergence

As we have seen in figure 1.8, a thread block consists of multiple warps. SIMT hides the execution latency by switching between warps.

1.3 Thread-level parallelism

Thread-level parallelism (TLP) arises when independent threads work on different data. Even though the thread/process works with own data, some of the data might be shared. Synchronization techniques such as locks, mutex, semaphores, etc., are used to prevent race conditions in those cases. TLP is exploited by using threads. Single Program Multiple Data (SPMD) is a sub-class of MIMD that consist of multiple threads and processes that execute nearly the same instruction but with its own data. In comparison with DLP, a program written to exploit TLP tends to work better when there is a possibility of multiple divergent paths of execution. DLP and TLP are often interchangeable, especially in a SPMD application in which the threads are on a convergent path. However, when there is a divergence, the execution path can only be determined at the runtime. Hence, SPMD applications often have implicit DLP that are often unexploited. Hardware optimizations to exploit MIMD is explained in section 1.3.

High-level programming languages such as C/C++, Java, Python, etc. provide abstractions to implement multi-threading with the support of operating system. Programmers use the libraries offered by the high-level languages to implement multi-threading in programs. POSIX threads [C+95], often referred to as PThreads, is an application programming interface for C and C++ languages.

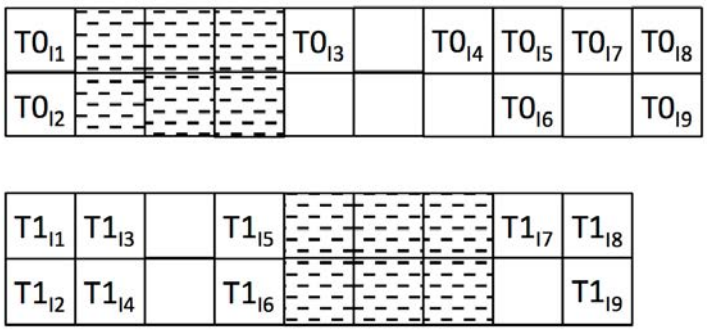


Figure 1.10: Thread execution without multi-threading

Explicit TLP Hardware multi-threading supports multiple thread contexts at the same time on the same processor core. Processes or threads are mapped to

each of the hardware threads. In general, software threads and process scheduling is done by the operating system. However, at a much finer granularity, the microprocessor schedules the hardware threads to maximize the resource utilization. Figure 1.10 shows the execution of two threads without hardware multi-threading. Hardware multi-thread allows both the context to execute at the same time. Depending on the scheduling technique, hardware multi-threading can be classified into

- Coarse grained multi-threading
- Fine grained multi-threading
- Simultaneous multi-threading

$T0_{I1}$	$T1_{I1}$	$T1_{I3}$		$T1_{I5}$	$T0_{I3}$		$T0_{I4}$	$T0_{I5}$	$T0_{I7}$	$T0_{I8}$	$T1_{I7}$	$T1_{I8}$
$T0_{I2}$	$T1_{I2}$	$T1_{I4}$		$T1_{I6}$				$T0_{I6}$		$T0_{I9}$		$T1_{I9}$

Figure 1.11: Coarse grained multi-threading (After cycle 1 T1 is scheduled as there is a long latency operation, T0 is rescheduled only when there is a long latency operation for T1)

1.3.1 Coarse-grained multi-threading

In coarse-grained multi-threading [ALKK90, SBCVE90, TE94] the processor switches the context when there is a very high latency operation such as a cache miss. Figure 1.11 illustrates how threads context switch when there is a long latency operation. Context switching could cost a few cycles to flush the pipeline. For coarse-grained muti-threading to be effective, the pipeline refill time should be much less than the stall time. IBM AS/400 [EJK+96] and Intel Itanium2 Montecito [MB05] cores implement coarse-grained multi-threading.

1.3.2 Fine grained multi-threading

In fine-grained multi-threading [Smi82] the thread interleaved execution between pipeline stages. However, only one thread is scheduled at a time in a given pipeline

$T0_{I1}$	$T1_{I1}$	$T1_{I3}$		$T0_{I3}$	$T1_{I5}$	$T0_{I4}$	$T0_{I5}$	$T0_{I7}$	$T0_{I8}$	$T1_{I7}$	$T1_{I8}$
$T0_{I2}$	$T1_{I2}$	$T1_{I4}$			$T1_{I6}$		$T0_{I6}$		$T0_{I9}$		$T1_{I9}$

Figure 1.12: Fine grained multi-threading (Rescheduling T0 do not wait until there is a very long latency operation for T1)

stage. The context switch happens without any cost. Figure 1.12 illustrates fine-grained multi-threading. In comparison with coarse-grained multi-threading, fine-grained multi-threading improves the throughput by hiding both short and long stalls. The disadvantage of fine-grained multi-threading is that it slows down the individual threads when there are no stalls [PH13]. Fine-grained multi-threading works well for the in-order processors with narrow issue width. Sun Microsystems Niagara [KAO05] uses fine-grained multi-threading.

$T0_{I1}$	$T1_{I1}$	$T1_{I3}$		$T0_{I3}$		$T0_{I4}$	$T0_{I6}$	$T0_{I8}$	$T0_{I9}$	$T1_{I8}$
$T0_{I2}$	$T1_{I2}$	$T1_{I4}$		$T1_{I5}$	$T1_{I6}$	$T0_{I5}$	$T0_{I7}$		$T1_{I7}$	$T1_{I9}$

Figure 1.13: Simultaneous multi-threading

1.3.3 Simultaneous multi-threading (SMT)

SMT improves the throughput of a superscalar core by enabling independent threads to share CPU resources dynamically. Resource sharing policies have [TEE+96a, CRVF04, LFMS01, EA03, EE07] 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 [LGF01]. However, these resource sharing heuristics essentially address multi-program workloads.

SMT architectures [TEL95] 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 [SFKS02], 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 [HS96].

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 has been shown to reach 85 % of the performance of an out-of-order 4-thread SMT 4-issue processor [HS99]. Therefore, in-order SMT appears as a good architecture tradeoff for implementing the cores of an SPMD oriented throughput processor.

In a conventional single-threaded processor low ILP phases leave unused resources. SMT processors can leverage higher issue width with the use of multiple threads. [PJS97] identifies the various source of complexity for implementing a wider superscalar microprocessor. With higher complexity, various delays at different pipeline stages increase, forcing the architecture to scale down the clock cycle rates at which the processor runs. Intel's implementation of SMT is called Hyperthreading [Upt], which appeared in 2002 and is still in use. [TT03, ECX+11] reports the performance improvement over a non-SMT architecture as around 20% in the older processors and 30% in the modern Intel processors.

1.3.4 Clustered multi-threading (CMT)

Clustering is also implemented in multi-threaded architectures. [KT98] proposed a clustered SMT architecture, which partitioned an aggressive, dynamic superscalar core into several independent clusters that do not share the resources. This architecture is analogous to a multi-core SMT. [CT04, DS02] proposed combining

SMT architectures with clustering to have higher IPC while maintaining higher clock rates. CASH [DS02] proposed a hybrid of SMT and CMP architectures, in which certain hardware is shared between the threads and use split resources for hardware whose implementation complexity is high. [CT04] explored the impact of clustering on instruction window wakeup and functional unit bypass logic, register renaming logic, fetch unit and integer register file on an SMT. AMD's Bulldozer [BBSG11] implements clustering in the multi-threaded processor.

SIMT is intended to run applications that are throughput oriented. SIMT does not support interrupts. Hence, it does not support operating system process. SIMT amortizes the control overhead. Unlike CMT, the pipeline operations are greatly reduced by sharing of instruction fetch and memory accesses. SIMT also needs specialized ISA for thread divergence management.

To improve the single thread performance CMT uses techniques such as superscalar execution, branch prediction, cache hierarchy, out-of-order execution, register renaming, speculative execution, etc. SIMT, on the other hand, is used in architectures aggressively optimized for throughput. SIMT supports a large number of cores. However, they are simple. They do not implement microarchitectural techniques to improve the single thread performance. Instead, it tries to improve the resource utilization with a large number of threads. More threads hide high latency operations. SIMT architectures have the benefits of clustering because of the independence of its participant threads. However, this can have adverse effects when there is less instruction occupancy or lots of thread divergence [5].

Multi-threading in DITVA DITVA is implemented over an SMT architecture. Whenever possible the instructions across SMT threads are vectorized. When vectorization is not feasible, the threads in DITVA executes as regular SMT. Like SIMT, DITVA groups the SMT threads into warps. Each thread in the warp occupies a lane. Consequently, DITVA also benefits from clustering, as the threads within the warp utilize independent resources.

[5] There are techniques such as Dynamic warp formation [FSYA07] and Dynamic warp subdivision [MTS10a] to have a higher tolerance for branch divergence

1.4 TLP vs DLP

As mentioned in section 1.3, TLP arises when independent task works on different data. In the code segment given in listing 1.1, exploiting TLP is straightforward.

Listing 1.5: Array addition with multi-threading

```
1
2 void add(int threadid, int num_threads) {
3         int N_per_thread = N/num_threads;
4         for(int i=threadid*N_per_thread; i<N_per_thread; i++){
5                 c_array[i] = a_array[i] + b_array[i];
6                 f_array[i] = d_array[i] + e_array[i];
7         }
8 }
```

A simple multi-threaded implementation of the code segment is shown in listing 1.5. All the threads execute the add function. In the given example, each thread adds a slice of the array. Parallel execution of the slices hides the long latency operations. Transformation of an algorithm with DLP to TLP is simple. However, vice versa is not straightforward. To illustrate this, consider listing 1.6, with 'condition' variable which is known at the run-time. Even though there is data independence, the control flow of the program becomes dependent on the outcome of the conditional variable that is independent between iterations. The code segment in listing 1.6 still has inherent TLP. To exploit DLP in this example, the programmer has to rely on software based SPMD vectorization languages such as OpenCL [Mun09, SGS10] and ISPC [PM12], or hardware based SPMD techniques such as Single Instruction Multiple threads (SIMT).

Listing 1.6: Array addition with branch

```
1
2 void add() {
3         for(int i = 0 ; i < N; i++ ){
4                 if(condition[i]){
5                         c_array[i] = a_array[i] + b_array[i];
6                         f_array[i] = d_array[i] + e_array[i];
7                 }
8         }
9 }
```

1.5 Chip multiprocessor

In section 1.1.4 and section 1.3.4 we have seen that clustering reduces the hardware complexity that allows the microprocessors to run at a higher clock rate. SMT architectures are introduced to improve resource sharing, but this increases the complexity. On the other hand, simultaneous multiprocessors (SMP) are computing elements that share nothing at all. However, on an SMP, the interprocessor communication is expensive. Moreover, the process or the threads running on different processors may share data. Caches, which reduce the latency of expensive memory operations when there is data reuse, are not shared on an SMP.

Chip multiprocessors (CMP) consist of multiple computing elements (cores) on a single chip. The cores may have a separate L1 cache but may share caches at other levels. It has to be noted that the each core in a CMP may still support SMT. CMP aims to improve the scalability by clustering less critical hardware resources. A typical desktop computer may consist of a microprocessor having up to 8 cores (Intel Core i7 Extreme Edition 5960X [ci7]) and microprocessors designed for high performance may have even more cores.

An alternative CMP design consists of clustering resource by type. A CMP with different types of cores is called heterogeneous CMP. A heterogeneous CMP may consist of some specialized cores. For instance, Cell processors [PAB+05] consists of 1 Power Processing Element (PPE) for general purpose computing and 8 Synergistic Processor Elements(SPE) optimized for SIMD execution. AMD Fusion architecture [BFS12] consists of CPU and GPU in a single chip,
big.LITTLE [Gre11] from ARM reduce energy consumption by coupling powerful large cores(e.g. Cortex-A15) with smaller cores (e.g. Cortex-A7) for energy savings.

1.6 Conclusion

In this chapter, we saw different techniques to exploit parallelism in a program. These techniques are not mutually exclusive. Depending on the program behavior (e.g., threadable vs non-threadable) and user requirement (e.g., latency vs throughput) an application can be programmed. Many general purpose microprocessors incorporate all these techniques to exploit parallelism. Scalability

and effective resource utilization still remain an issue in these microprocessors. For example, SMT processors do not scale well beyond eight threads. Also, a program compiled with inefficient static vectorization does not use the SIMD execution units very well. There are inter-thread similarities (chapter 2) that allows threads to share the control flow. DITVA removes these inefficiencies of modern general purpose microprocessors. DITVA improves the throughput and overall energy consumption of the cores while maintaining the latency of the threads. DITVA is designed keeping in mind a heterogeneous CMP that may consist of regular cores coupled with DITVA cores optimized for SPMD applications.

DITVA does mask based vectorization of instruction like SIMT. Unlike SIMT, which uses a stack based divergence management, DITVA is heuristic based and hence it does not require the support of any specific ISA for divergence management. Moreover, the use of intrinsics and compiler hints for reconvergence makes SIMT comparable to a predicate based static vectorization. DITVA, on the other hand, is entirely dynamic and even support speculative execution of branches with the help of a branch predictor. SIMT relies on fine-grained multi-threading (section 1.3.2) to maximize resource utilization while DITVA relies on simultaneous multi-threading technique (section 1.3.3).

Inter-thread similarity forms the basis of dynamic vectorization in DITVA. dynamic vectorization enables sharing of control flow when the threads are doing similar instruction operations. Dynamic vectorization results in reduction in pipeline operations in the microprocessor, thus reducing energy consumption and improving the utilization of existing SIMD units. In chapter 2, we discuss inter-thread similarities and the ways to exploit it to dynamically vectorize instructions.

Chapter 2

Exploiting inter-thread similarity in microprocessors

In this chapter, we will discuss the presence of inter-thread instruction similarities that cause redundancy while running SPMD application in an in-order SMT [TEE+96a] architecture. Dynamic Inter-Thread Vectorization Architecture (DITVA) avoids this redundancy by dynamically vectorizing the instructions across threads. The efficiency of DITVA relies on convergent execution for dynamic vectorization. Hence, early reconvergence of threads after following divergent paths is of utmost importance. In this chapter, we discuss various reconvergence techniques and our preferred reconvergence heuristics for DITVA. We will also discuss related microarchitectural techniques that exploit inter-thread redundancies.

2.1 Inter-thread similarities in SPMD applications

SPMD provides an easy-to-program model for exploiting data-level parallelism. A programmer may use programming libraries such OpenMP or Pthread to write SPMD application. SPMD is a higher level abstraction that uses of multiple instruction streams (threads or processes) that operate on a different subset of the data. The instruction streams would be loaded from the same executable.

And hence, each instruction streams often execute the same instructions. The instruction streams may also be on a divergent path. In this case, the instructions streams may not share the instruction flow. The instruction streams are not required to execute the same instruction at every time step and hence, instruction stream has its own program counter that is updated independently. SPMD applications may share the same program memory. The instruction streams are executed simultaneously. When an instruction stream is blocked for a long latency operation, an SMT processor may find another instruction stream that is not blocked.

SPMD assumes that the parallel instruction streams execute identical instructions. In a single core SMT processor, SPMD applications benefits from shared instruction caches as the instruction streams share most of the instructions. The instruction streams may also share some data in the data caches. However, the flow of instructions in the pipeline front-end is redundant [KJT04], resulting in resource wastage and high energy consumption.

2.1.1 Source of inter-thread similarities

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 [BKSL08] in figure 2.1.

All the threads execute the convergent blocks while only some threads execute the divergent blocks. Moreover, more than one thread often executes each divergent block. Threads that are in the convergent path fetch the same instructions. This results in instruction redundancy. The redundancy in SPMD applications could be instruction redundancy or data/value redundancy. With instruction redundancy, operations at the pipeline front end are repeated for each of the instruction across threads. With data redundancy, instructions execute with the same data.

Since the threads are inherently independent, they are not guaranteed to execute in lockstep i.e., some threads may be ahead of the others in the execution flow. SPMD applications typically resort to explicit synchronization barriers at certain execution points to enforce dependencies between tasks. Such barriers are

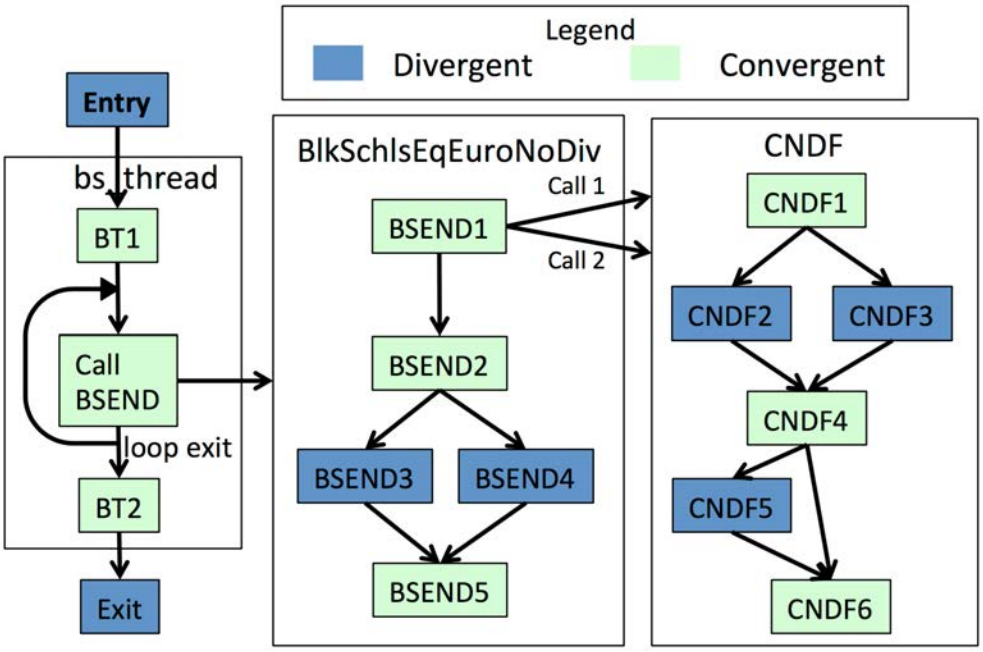
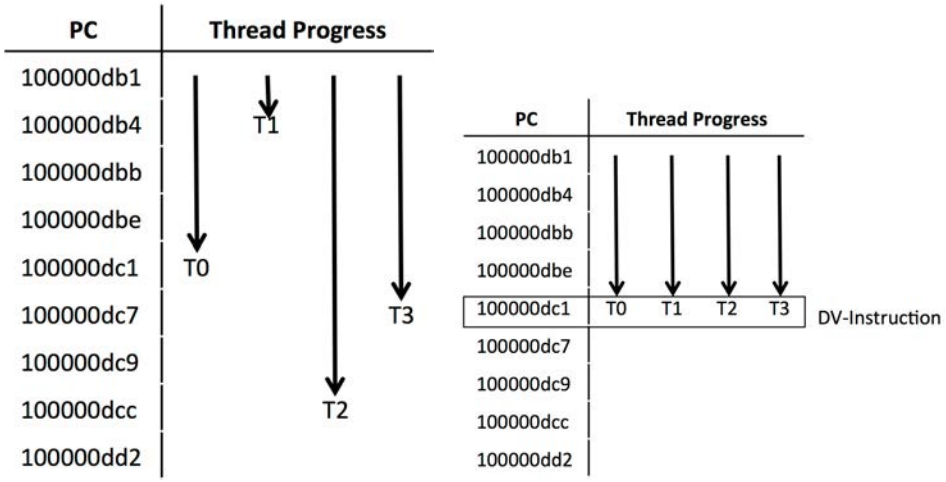


Figure 2.1: Control flow graph of blackscholes benchmark

natural control flow reconvergence 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 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 [MCP+14].



(a) Thread execution in SMT (b) Lockstep execution and formation of vectorized instruction

Figure 2.2: Overview of dynamic vectorization

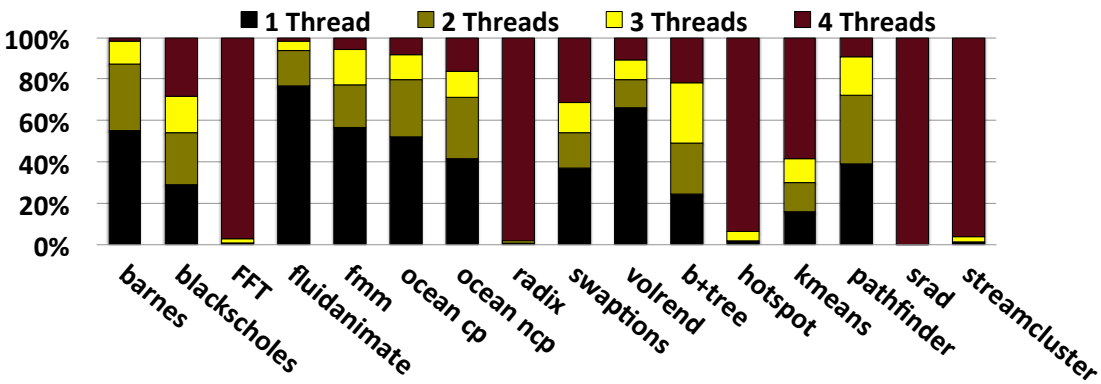


Figure 2.3: Breakdown of average vectorizable instructions for 4-way SMT

2.1.2 Vectorization efficiency in an SMT

Figure 2.3 illustrates the functional unit occupancy with a breakdown of vectorizable instruction count per for a 4-way SMT[1] with MinSP-PC-Roundrobin fetch

[1]The programs shown in the figure are executed with 16 threads. The 16 threads are grouped into *warps* of 4 threads. Warps are defined in section 3.2

heuristic[2]. Functional unit occupancy is the occupancy of active threads in an instruction. This is equivalent to the active threads represented by the set bits of a mask in an SIMT (see section 1.2.2). The *k thread* bar represents the average number of active threads in vectorized instructions throughout the execution of a program. For example, in blackscholes, around 30% of the instructions are non vectorizable, since they have occupancy one. Efficiency increases with the occupancy. Hence, the best case here are the instructions with occupancy 4 since they are perfectly vectorized. 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 vectorized instructions. On the other hand, the threads of benchmarks that have low exploitable DLP like *Fluidanimate* tend to diverge. Only 23% of vectorized instructions contain more than one instruction on *Fluidanimate*.

2.1.3 Dynamic Vectorization

Dynamic vectorization leverages the 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.

Figure 2.2a shows the execution of a code block by threads in SMT. In this example, we assume that all the threads are executing the same code block. However, the threads may still fetch different PC at a given time. Figure 2.2b shows the dynamic vectorization across threads in an SMT processor. The vectorization involves two steps

1. Maximize lockstep execution
2. Vectorize instructions across threads that are running in lockstep

Maximize lockstep execution Threads run in lockstep when they fetch the same PC at the same time. Lockstep execution is only possible among the threads that are in the convergent path. Lockstep execution is achieved by using special heuristics that prioritizes the slow running thread to catch-up with the faster running thread. When the threads are in the divergent path, the threads are

[2]We will discuss MinSP-PC heuristic in section 2.2.4

prioritized for early reconvergence. In section 2.2, we discuss some of the reconvergence techniques and our motivation to use a hybrid MinSP-PC-Roundrobin heuristics in DITVA to maximize the convergent execution.

2.2 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 2.1. 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 2.1: A program with divergence

```
1 for(i = 0 to n)
2         if((i+tid) %c))
3                 A();
4         else
5                 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.

2.2.1 Stack-based explicit reconvergence

Levinthal [LP84] proposed a stack-based approach that uses ISA support for the Chap SIMD graphics processor. The processor uses a run flag register (predication mask) and two mask stacks to determine the active SIMD lanes, with each bit of the stack and run flag register corresponding to an SIMD lane. The stacks and the run flags are updated at the beginning and end of a *basic block*[3]. Essentially, an entry is created in the stack for each divergent path and it is removed after it completes execution. This mechanism supports nested branch and looping statements. NVIDIA GPU uses a stack-based approach, with additionally storing the address of reconvergent point generated at compile time.

Keryell and Paris [KP93] used *Activity counters* to replace stack of masks by counters to track the lane activity. *Activity counter* is the nesting level of a thread when it was last active. The thread is active when the activity counter is equal to the current nesting level. The assumption here is that, when a thread becomes inactive at a certain nesting level, it will only be active again upon reaching the same nesting level.

2.2.2 SIMT extensions for stack-based implicit reconvergence

With the stack-based approach, a warp may have multiple instruction streams (IS) when there is a divergence. Each of these warps may be further sub-divided when there are divergences with those warps. This approach hurts the lane utilization for several applications which may have a significant number of divergences. Fung et al [FSYA07] proposed Dynamic Warp Formation, which merges threads from different warps when they are fetching the same PC. At reconvergence, the stacks are updated. Dynamic warp formation considers several scheduling policies to enhance early reconvergence of threads. Fung et al. observe that reconvergence at the immediate post-dominator is nearly optimal with respect to oracle implementation. A basic block A post dominates another block B if all the paths from B to the exit node passes through A. There may be multiple paths between A and B. An immediate post-dominator is a block which does not strictly post-dominate

[3]A basic block is a section of a program that has a single entry and exit point. The threads may be executed without divergence within a basic block

any other post-dominator.

A drawback of Dynamic warp formation is that it does not support execution from parallel branches. They hide the latency with deep multi-threading. Deep multi-threading increase the cost of register file implementation. Dynamic warp subdivision [MTS10b] proposes intra-warp latency hiding by parallel execution of divergent branches and subdivision of warps on memory divergences by instruction streams (warp-split). Instruction streams introduce a problem of the reconvergence point. Some instruction streams may reconverge late if they are allowed to run ahead. On the other hand, trying to enforce an early reconvergence may defeat the purpose of instruction streams as the warp running ahead has to wait until the others catch up. Dynamic warp subdivision explores several opportunistic reconvergence heuristics.

2.2.3 Stack-less explicit reconvergence

Lorie-Strong [LSJ84] used a stack-less implementation that uses PC and logical block ordering performed by the compiler. A block with at the lowest order gets the priority for execution. At a potential reconvergence point, the block numbers are compared, and the blocks are merged when there is a match.

Thread frontiers [DKW+11] uses a similar approach where a compiler assign a priority to basic blocks. The scheduling order of the basic blocks is used to determine the *thread frontiers* statically at the compile time and it is used to insert divergent and reconvergent instructions. The *thread frontiers* are the set of basic blocks which the divergent threads may be executing which is a possible point for reconvergence, i.e., a thread frontier of a warp executing in a basic block has all the divergent threads of that warp. *Thread frontiers* acts as synchronization points for the divergent threads while preventing deadlock situations. Hardware implementation uses the compiler hints for thread reconvergence. In an Intel GPU implementation *per-thread PCs* are used to selectively enable or disable the threads in a warp by comparing the thread PCs to the warp context PC. On a branch, the priority is given to the highest priority block in the thread frontiers. A native implementation uses a sorted predicate stack to dynamically assign priority for thread frontiers.

2.2.4 Stack-less implicit reconvergence

Takahashi [Tak97] proposed thread scheduling based on a branching mechanism for reconvergence without compiler hints. The mechanism prioritizes a basic block with the lowest value of PC upon a divergence in order not to pass a potential reconvergence point. A set *activity flag* in the lane indicates that the instruction in that lane is executed The activity flag is set when a lane is active in the current basic block. The mechanism realizes barrier synchronization and skipping of *then* block in an *if-else* condition, when no lane is active in the *then* block. However, it cannot skip the *else* block. The author suggests insertion of an instruction as a workaround for the problem.

MinPC [QHJ88] is a simple stack-less heuristics that prioritizes the threads based on *textually earliest program point* in the source code. 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. This heuristics will fail if the compiler heavily re-orders the basic blocks. MinPC heuristic may not work well with function calls. In comparison with takahashi, MinPC need not execute both the if-else blocks if the threads dont execute in that path. As MinPC heuristic is PC value based, it does not require compiler hints.

MinSP-PC [Col11] 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, resulting in a lower value of stack pointer in a downward growing stack. Therefore, the threads with highest call depth are given top priority.

Let us consider threads T0 and T1 executing the code blocks shown in figure 2.4. 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 2.5 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 2.5a. 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

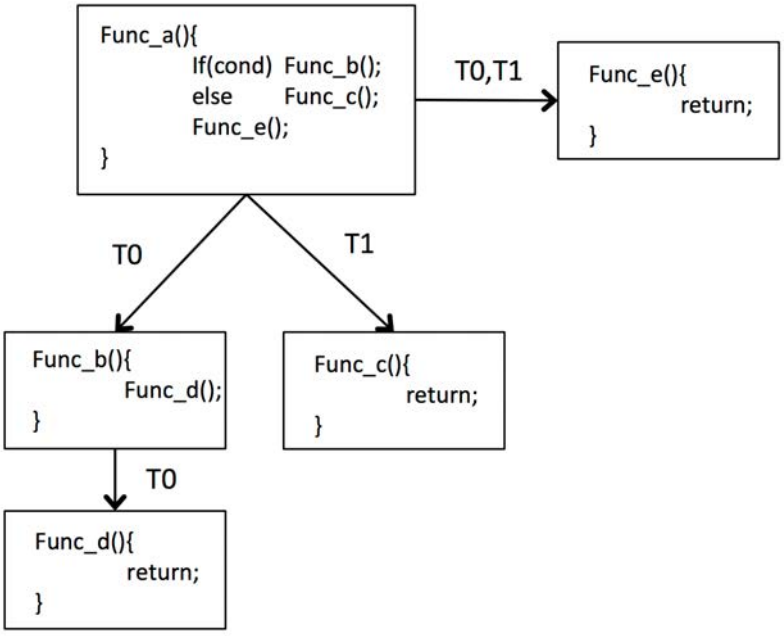
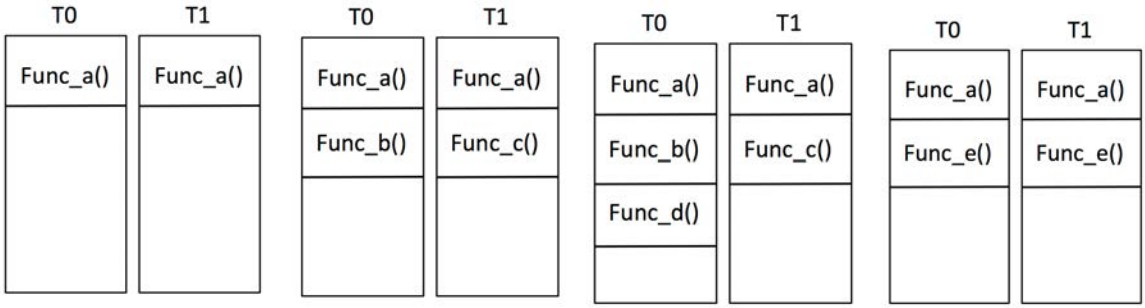


Figure 2.4: Control flow



(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 2.5: Growth of stack in time for threads T0 and T1

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 2.6 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.

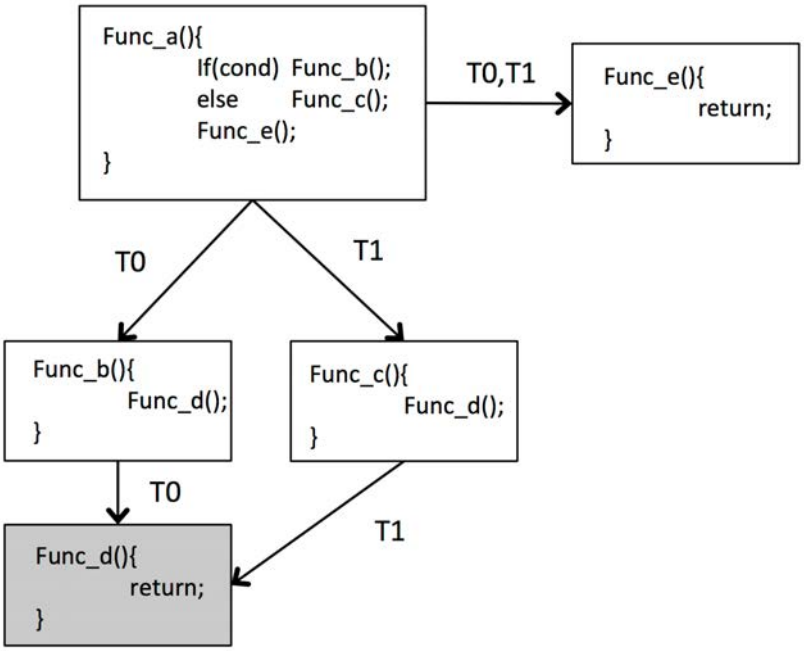


Figure 2.6: Control flow with synchronization point at Func_d

2.3 Vectorization of instructions across threads

Instructions across threads are vectorized when they run in lockstep. Each thread occupies a fixed lane of the vector instruction. A dynamically vectorized instruction may be partially occupied when some of the threads are in divergent paths. In this case, the threads in each divergent path form partially occupied vector instruction.

2.4 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 focus instruction 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 [KJT04] 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 [GCC+08], fused instructions across threads in a 2-way SMT when both the threads are executing identical instructions. A fused instruction only has one instance in the pipeline front-end. At the back-end pipeline, the fused instruction is treated as two separate instructions. The instructions are executed parallelly if there are free resources; otherwise, they are pipelined. For thread fusion both the threads should be executed in lock-step mode (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. [GCC+08] defines synchronization point 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 [LFB+10] improves it further by achieving this without the 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) [DFR10] uses the instruction similarity and retires the 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 [MBW14] focuses on the energy efficiency of an in-order core processor (OpenSPARC T1 [PWH+08]) 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 [QHJ88] 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 [MBW14] 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.

2.5 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 [DKW+11, MDKS12] or more flexible execution [FSYA09, BCD12, LKB14]. However, they all target applications specifically written for GPUs, rather than general-purpose parallel applications.

2.6 Conclusion

SMT is an efficient architectural model that improves the CPU resource utilization when the program in execution have long latency operations. SMT works well when multiple programs are in execution in a single processor core. However, when we execute the threads from the same SPMD application in an SMT, they often execute the same instructions. In this chapter, we discussed the source of instruction redundancy and the use of dynamic vectorization to eliminate the redundancies. We observe that almost 80% of the instructions are dynamically vectorizable.

Dynamic vectorization requires lockstep execution which can only be assured with good thread reconvergence techniques. Thread reconvergence techniques could be broadly classified into stack-based and stack-less. GPUs rely on stack-based reconvergence technique, which requires ISA support. Stack-based techniques are hardware assisted software-based technique. Hardware assisted techniques rely on hard explicit reconvergence points and do not have the support for legacy applications. DITVA is a hardware only dynamic vectorization architecture and hence, DITVA relies on stack-less heuristics for reconvergence.

DITVA exploits 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 (section 1.2.1), SIMT (section 1.2.2) also known as GPU architectures and SMT architectures (section 1.3.3). Static DLP, detected in 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 [Rus78]. 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 [PW86]. In the mid-1990s, SIMD instructions were introduced in the microprocessor ISAs, first to deal with multimedia applications [Lee97] 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 [MGG+11] 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.

We discussed several CPU based techniques to avoid the resource wastage. Most of them focus on energy efficiency by optimizing pipeline frontend. DITVA not only aims to provide energy efficiency, but also improve the speedup by utilizing the unused SIMD resources. DITVA aims to do this with minimal overhead.

GPUs have different design goals. Their stack-based reconvergence technique and is ideal for embarrassingly parallel applications specifically programmed for the architecture. They have high latency tolerance and are not ideal for latency oriented process like an operating system process. They support a large number of threads. To support large number of threads GPU consists of large number of slow registers. DITVA is an intermediate between CPU and a GPU. DITVA supports more threads than a CPU but much less threads in comparison with a GPU. It also supports latency oriented applications. 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 [ND10b].

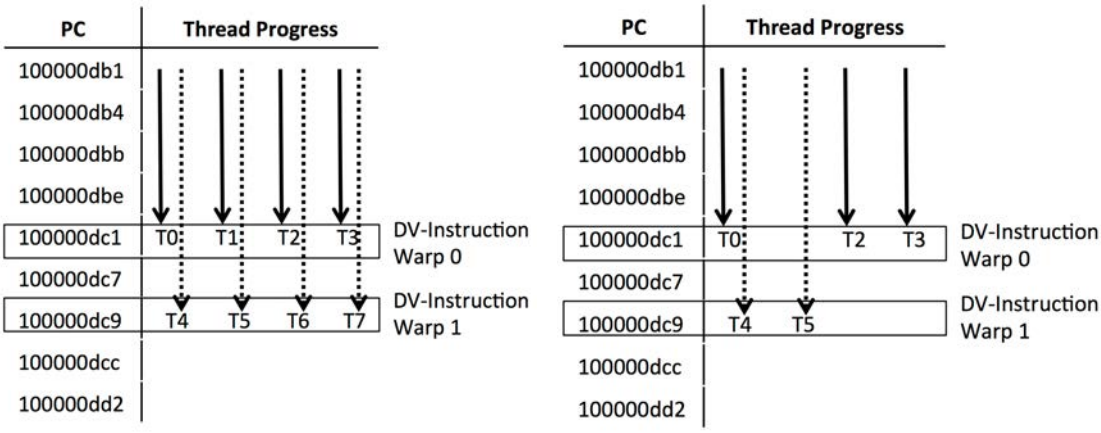
Putting together chapter 1 and chapter 2, we discussed the scope of DITVA architecture and the methods adopted by DITVA to reduce inter-thread similarities. In the next chapter, we describe the detailed architecture of our implementation of dynamic vectorization.

Chapter 3

Dynamic Inter-Thread Vectorization Architecture

Modern microprocessors have support for SIMD instruction execution. SIMD instructions are formed by exploiting DLP in a program. A program compiled for SIMD is often power efficient since it avoids the redundancy in the front-end pipeline operations. SPMD applications have implicit DLP that are often not exploited due to inefficiency in the program or compiler optimizations. The transformation from TLP to DLP often require recompilation and hence the access to the source code of the applications.

DITVA extracts DLP from TLP in microarchitecture without recompilation of programs and hence, DITVA supports legacy applications. DITVA dynamically vectorize instruction across threads. *Dynamic vectorization* in DITVA is the ability of the processor to vectorize instructions at the runtime at the microarchitecture without any human or compiler intervention. It exploits the missed vectorization opportunities by the programmer and compiler in SPMD applications. The instructions executed by the threads could be of varying data width. For example, 8-bit char, 32-bit int, 64-bit float, 128 bit SSE, 256-bit AVX etc. DITVA is capable of vectorizing instructions of different data width. Therefore, DITVA does not impact the performance of SPMD applications with SIMD instructions. In addition to this, dynamic vectorization allows efficient utilization of large SIMD units when the actual program is compiled for a target architecture with smaller SIMD units.



(a) Dynamically vectorized instruction 100000dc1 and 100000dc9 occupying Warp 0 and Warp 1 respectively

(b) Warp 0 and Warp 1 with some of the threads re-in divergent path

Figure 3.1: Overview of DV-instruction formation in DITVA

3.1 Warps in DITVA

In section 1.3.3 we discussed SMT architecture, which supports multiple threads to hide the long latency operations. With dynamic vectorization, the control flow of SMT threads is shared by multiple threads that are on a convergent path. When all the threads are in the convergent path, the control flow is equivalent to a single thread processor. Perfect lockstep execution reduces the efficiency of dynamic vectorization since it cannot hide the long latency operations very well. With perfect lockstep execution, an entire stream of instruction, which may consist of multiple scalar threads, may be blocked when there is a data cache miss, even for one of the thread. To overcome this issue, DITVA groups multiple threads into a statically formed *warp*. A *warp* in DITVA is similar to SIMT execution model used on NVIDIA GPUs. In DITVA, the thread count could be modelled based on the SIMD execution units supported by the architecture. A DITVA processor supports more than one such warp. Having multiple warps provides the benefit of allowing instructions from other warps to be scheduled when the threads of a warp are blocked due to long latency operation. In this thesis, we evaluate several DITVA configurations based on the thread count in the warp as well as the number of warps.

Figure 3.1a shows threads $T0$ to $T3$ grouped into $warp0$ and $T4$ to $T7$ grouped into $warp1$. Each thread is strictly associated with a single warp and it occupies a fixed *lane* in the warp. For instance, $T0$ is always associated with $lane0$ of $warp0$. There may be divergent control flow within a warp. In this case, the dynamically vectorized instructions are partially occupied as shown in figure 3.1b. For simplicity, only one divergent path is shown in the figure. In the given example, $T0$ of warp 0 forms another vectorized instruction (with single thread occupancy) and in warp 1, $T4$ and $T5$ forms 1 (with two thread occupancy) or 2 (with single thread occupancy) vectorized instruction.

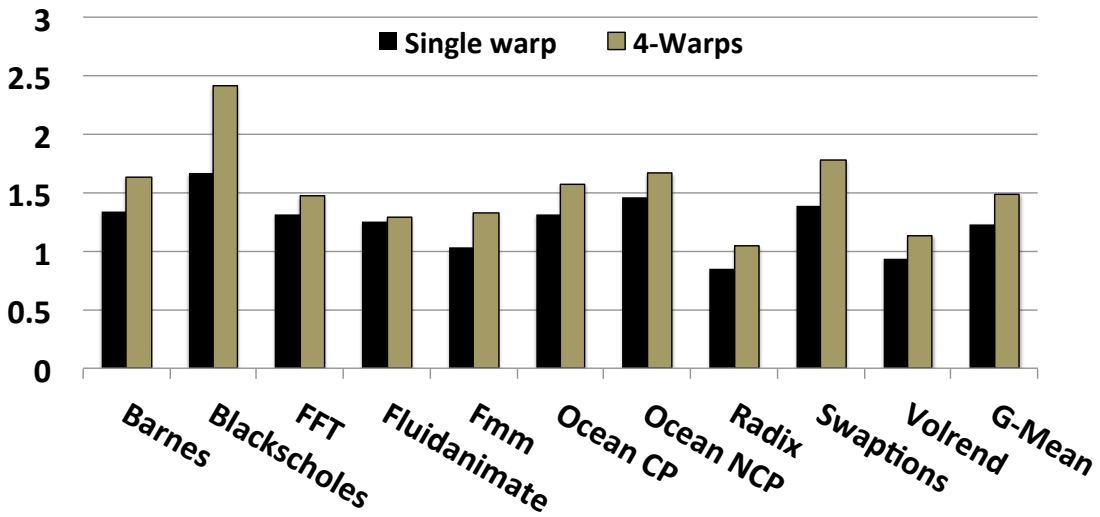


Figure 3.2: Speed up of single warp of 16 threads and 16 threads grouped into 4 warps with dynamic vectorization using DITVA

Our experiments on SPMD applications from the PARSEC and SPLASH benchmarks [BKSL08, Uni06] 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 speedup of 1.44× over 4-thread in-order SMT, a very significant performance gain. Figure 3.2 shows the speedup of 16 threads, single warp DITVA [1] and 16 threads, grouped into 4 warps, over a 4-way SMT. 4-warp DITVA has an average speedup of around 50% that clearly shows the

[1]Please note that 16 thread, single warp DITVA is difficult to build as it would need very large vector units

benefits of having multiple warps to hide high latency operations. Compared to 16 threads, single warp DITVA, 4-warp DITVA, that has the same number of threads achieves around 25% speedup. We will discuss the application behavior in section 4.

DITVA provides these benefits at a limited hardware complexity in terms of the silicon area 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.

3.2 Overview of the Dynamic Inter-Thread Vectorization Architecture

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

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 an instruction stream from the notion of a 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.

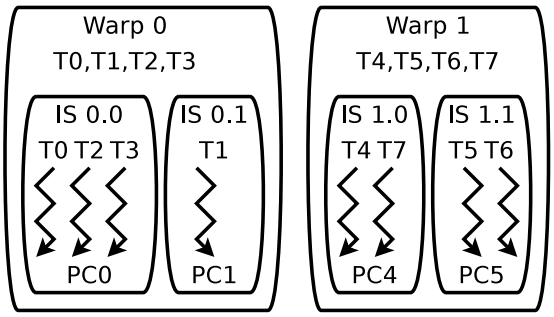


Figure 3.3: Logical organization of hardware threads on 2-warp× 4-thread DITVA. Scalar threads are grouped into 2 warps of 4 threads. Scalar threads sharing the same PC within a warp form an Instruction Stream (IS).

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 3.3, 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). While thread-to-warp assignment is static, the 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 3.3, 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 3.5).

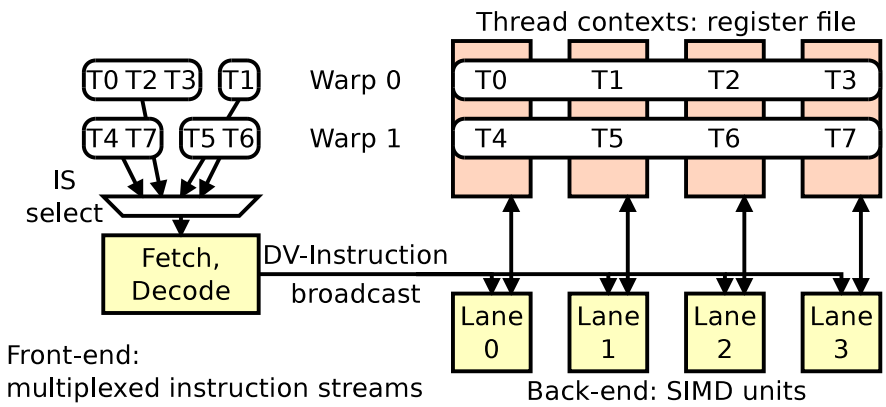


Figure 3.4: Thread mapping to the physical organization.

Mapping to physical resources DITVA consists in a front-end that processes Instruction Streams and a SIMD back-end (Figure 3.4).

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 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.

3.3 Pipeline architecture

We describe the stages of the DITVA pipeline, as illustrated in Figure 3.5.

3.3.1 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.

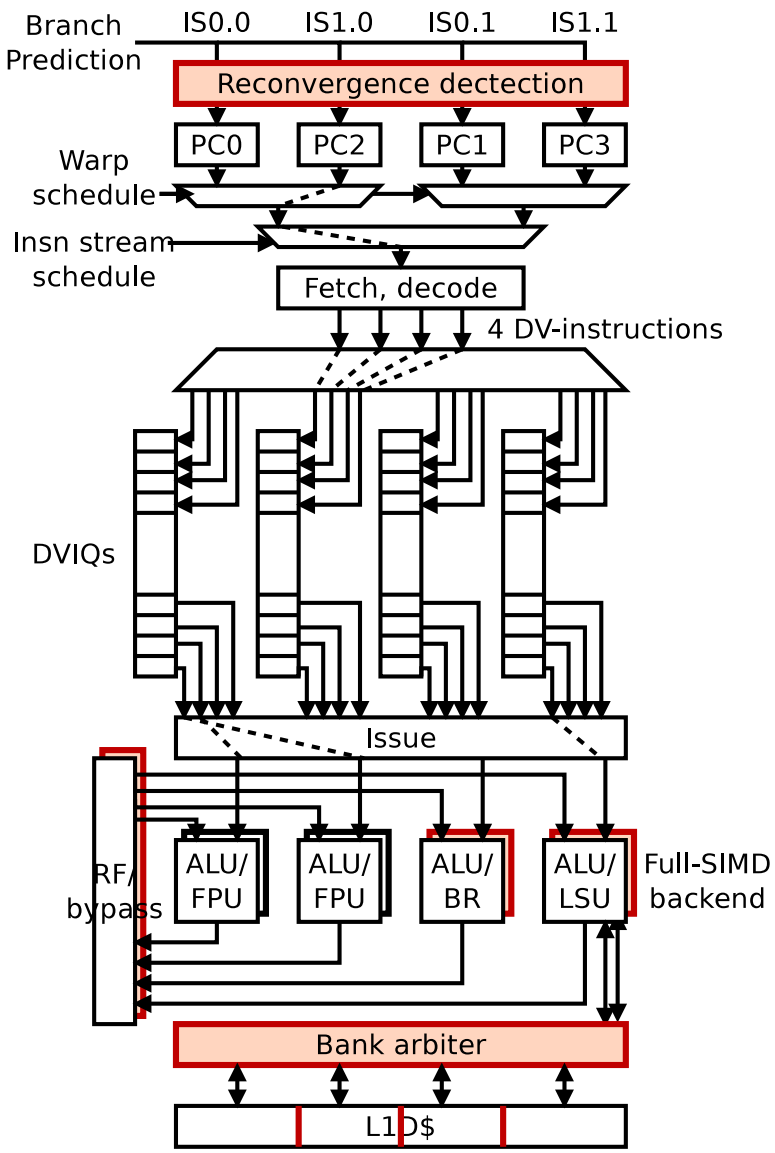


Figure 3.5: Overview of a $2W \times 2T$, 4-issue DITVA pipeline. Main changes from SMT are highlighted.

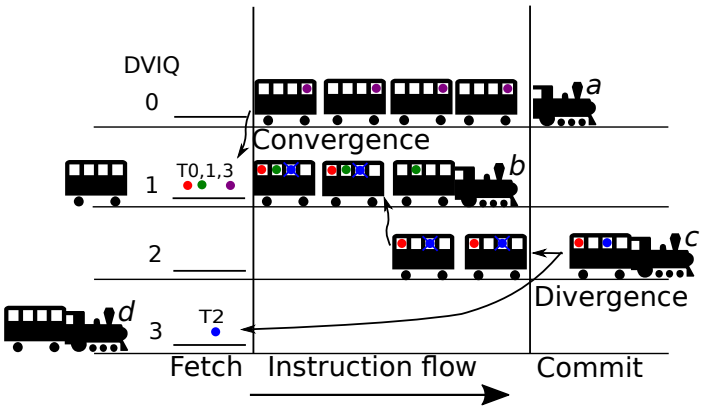


Figure 3.6: Instruction stream tracking in DITVA. Instruction Streams (a, b, c, d) are illustrated as trains, DVIQs as tracks, DV-instructions as train cars, and scalar instructions as passengers.

Branch prediction and reconvergence detection Within the front-end, both the PC and inclusion mask of each IS are speculative. An instruction address generator initially produces a PC prediction for one IS based on the branch history of the first active scalar thread of the IS. After instruction address generation, the PC and process identifier of the predicted IS_i, are compared with the ones of the other ISs of the same warp. A match between IS_i and IS_j indicates they have 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 3.6 illustrates convergence happening between threads 0 and 1 in IS b with thread 3 in IS a. IS b contains threads 0, 1 and 3 after convergence, so its inclusion mask is now 1101. IS a is aborted. Earlier in time, convergence did also happen between threads 0 and 2 in IS c and thread 1 in IS b. All the threads of an IS share the same instruction address generation, by speculating that they will all follow the same branch direction. Unlike convergence, thread divergence within an IS is handled at instruction retirement time by leveraging branch misprediction mechanisms, and will be described in Section 3.3.5.

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 [TEL95, EE07]. Then, an intra-warp instruction fetch steering policy selects one IS within the selected warp. The specific policy will be described in Section 3.5. 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.

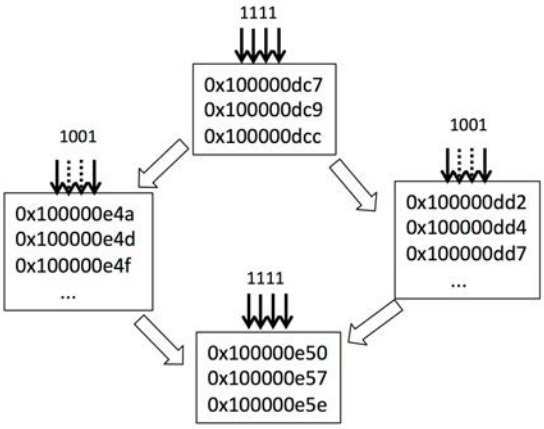
Instruction Queue After the decode stage, the DV-instructions are pushed in a DV-instruction queue (DVIQ) associated with the IS. In a conventional SMT, instruction queues are typically associated with individual threads. DITVA applies this approach at the IS granularity: each DVIQ tail is associated with one IS. Hence, for a $nW \times mT$ DITVA, there could be a maximum of $n \times m$ active DVIQ's and each warp may have upto m DVIQ's associated with it.

Unlike in SMT, instructions that are further ahead in the DVIQ may not necessarily belong to the IS currently associated with the DVIQ, due to potential IS divergence and convergence. For instance in Figure 3.6, DVIQ 2 contains instructions of threads T0 and T2, while IS 2 has no active threads. The DV-instruction mask avoids this ambiguity.

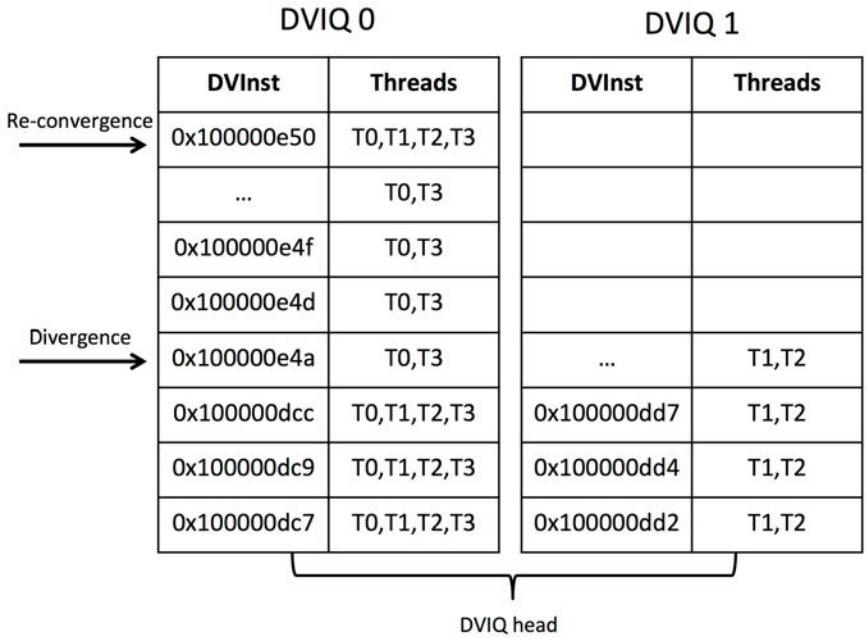
3.3.2 In-order issue enforcement and dependency check

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.

Figure 3.7 illustrates this scenario with an example. Consider the program flow with divergence and reconvergence shown in figure 3.7a. In this example, threads *T1* and *T2* takes a divergent path after instruction *0x100000dcc* and forms IS 1. The instructions following the divergence are inserted into *DVIQ 1*. *T1* and *T2* later reconverges at instruction *0x100000e50*. DV-instructions after reconvergence are inserted in DVIQ 0. In the given example, instruction *0x100000dcc* must be issued before instruction *0x100000dd2*. Similarly, all the



(a) Program flow with divergence and reconvergence



(b) DVIQ state

Figure 3.7: Illustration of instructions from a scalar thread occupying multiple DVIQ's

instructions in the divergent path must be issued before instruction *0x100000e50*

. 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.

DVIQ 0

DVInst	Threads	Seqno
0x100000e50	T0,T1,T2,T3	18,24,24,18
...	T0,T3	
0x100000e4f	T0,T3	9,-,-,9
0x100000e4d	T0,T3	8,-,-,8
0x100000e4a	T0,T3	7,-,-,7
0x100000dcc	T0,T1,T2,T3	6,6,6,6
0x100000dc9	T0,T1,T2,T3	5,5,5,5
0x100000dc7	T0,T1,T2,T3	4,4,4,4

DVIQ 1

DVInst	Threads	Seqno
...	T1,T2	
0x100000dd7	T1,T2	-,10,10,-
0x100000dd4	T1,T2	-,9,9,-
0x100000dd2	T1,T2	-,8,8,-

DVIQ head

Figure 3.8: Instruction issue with sequence number

Figure 3.8 shows instruction issue with the use of sequence number. *DVIQ 0* and *DVIQ 1* belongs to the same warp and the instructions in both the queues are candidates for issue. The processor keeps track of the expected issue sequence number. At the current state, the expected issue sequence number is {4,4,4,4}. DV-instruction *0x100000dc7* has all the four sequence number matching and hence it is the next instruction to be issued. After the issue of *0x100000dc7*, the expected sequence number if incremented to {5,5,5,5}. After issuing *0x100000dcc*, the threads takes divergent paths. At this stage, the expected sequence number of the processor is {7,7,7,7}. However, both instructions *0x100000e4a* and *0x100000dd2* in the divergent paths are candidates for instruction to be issued.

Hence, the instructions in the divergent paths can be issued in parallel irrespective of the order of instruction fetch. Similarly, at the reconvergence point, instruction *0x100000e50* cannot be issued until the expected sequence number is {18,24,24,18}.

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.

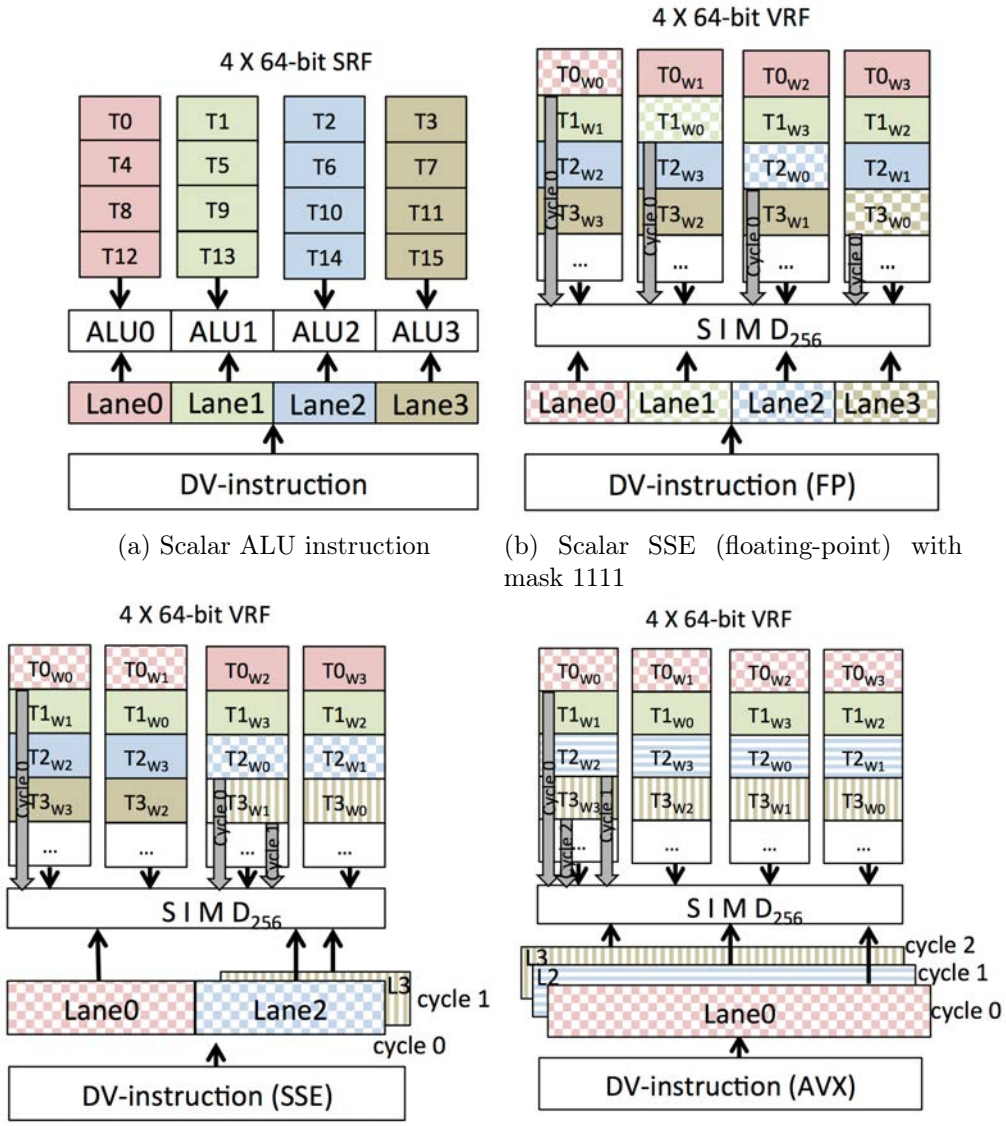
A DV-instruction cannot be launched before all its operands are available. A scoreboard tracks instruction dependencies. In an SMT having n threads with r architectural registers each, the scoreboard consists of a nr data dependency table with 8 ports indexed by the source register IDs of the 4 pre-issued 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.

3.3.3 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 subfiles 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 3.9a shows the execution of a scalar DV-instruction (i.e. dynamically



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

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.

3.3.4 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 3.9b 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 3.9c shows the pipelined execution of a SSE DV-instruction with mask 1011 in a 4W×4T DITVA. In figure 3.9c, 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 3.9d shows the execution of a AVX DV-instruction with mask 1101 in a $4W \times 4T$ DITVA.

3.3.5 Handling misprediction, exception or divergence

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

In DITVA, we generalize branch divergence, misprediction and exception handling through a unified mechanism.

Branch divergence is detected at branch resolution time, when some threads of the current IS, IS_i, actually follow a different control flow direction than the direction the front-end predicted. IS_i is split into two instruction streams: IS_i continues with the scalar threads that were correctly predicted, and a new stream IS_j is spawned in the front-end for the scalar threads that do not follow the predicted path. The inclusion masks of both IS are adjusted accordingly: bits corresponding to non-following threads are cleared in IS_i mask and set in IS_j mask. For instance, in Figure 3.6, 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 3.6, they correspond to disabling thread T2 in the DV-instructions from IS_2 and IS_1.

Evolution of the IS and DVIQ states upon thread divergence and reconvergence Figure 3.10, illustrates the evolution of the IS and DVIQ states upon thread divergence and reconvergence.

Divergence Figure 3.10b shows the state of the pipeline when the instruction PC-11d of thread T5 illustrated in figure 3.10a triggers a branch misprediction. In the given example, a new IS 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 partial misprediction in a warp, the mask corresponding to the lane of all the DV-instructions in the pipeline corresponding to that warp should be updated. 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.

Reconvergence Figure 3.10 also illustrates reconvergence for T2 (IS2).The PC of each of the IS is compared. At the reconvergence point, IS0 and IS1 have the same PC. One or more of the IS having the same PC is merged with the other IS and the resultant mask is updated and is used while decoding the DV-instruction. In figure 3.10b, IS2 is merged with IS0 and the DV-instruction with PC-142 is decoded with mask 1110.

3.4 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 4 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 4 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.

3.5 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.

Conventional instruction fetch policies such as Icount [TEE+96b] focus on fairness and efficient resource utilization. These policies try to optimize execution throughput or fairness in the context of conventional SMT. They are not adapted to the context of DITVA. On DITVA, and if the threads belong to the same process, one should try to reinitiate lockstep execution whenever possible.

In DITVA, we use a hybrid Round-Robin/MinSP-PC instruction fetch policy. Round-Robin/MinSP-PC is a stack-less implicit reconvergence heuristics. 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[2], 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 make progress. When two or more threads are executed in lockstep mode, this advantage is increased and is expected to favor re-synchronization. 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.

3.6 Clustered Multi-threading in DITVA

The introduction of more threads to an SMT processor introduce the challenge of reducing the complexity of implementation. In general, sharing resources increase the complexity. For example, the issue width of a superscalar processor is limited by the cost of its implementation. With the lanes in warps, DITVA logically

[2]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.

partitions the resources across threads, that would result in smaller hardware structures without the need for crossbars. The threads from different warps may still share the resources, however, they are time interleaved. DITVA partitions TLB, DVIQ, issue logic and execution units (ALU as well as SIMD lanes).

3.7 Conclusion

In this chapter, we discussed the implementation details of the Dynamic Inter-Thread Vectorization Architecture (DITVA). Some of the most important design aspects of DITVA are

1. **Instruction streams:** Dynamic formation of instruction stream allows DITVA to share the instruction flow control across threads.

2. **Warps:** The best case scenario for DITVA is perfect lockstep. However, it completely defeats the purpose of SMT, as any one thread blocked on a long latency operation will block the entire pipeline. The use of multiple warps that consists of statically assigned threads keeps the pipeline busy even when some of the warps are blocked.

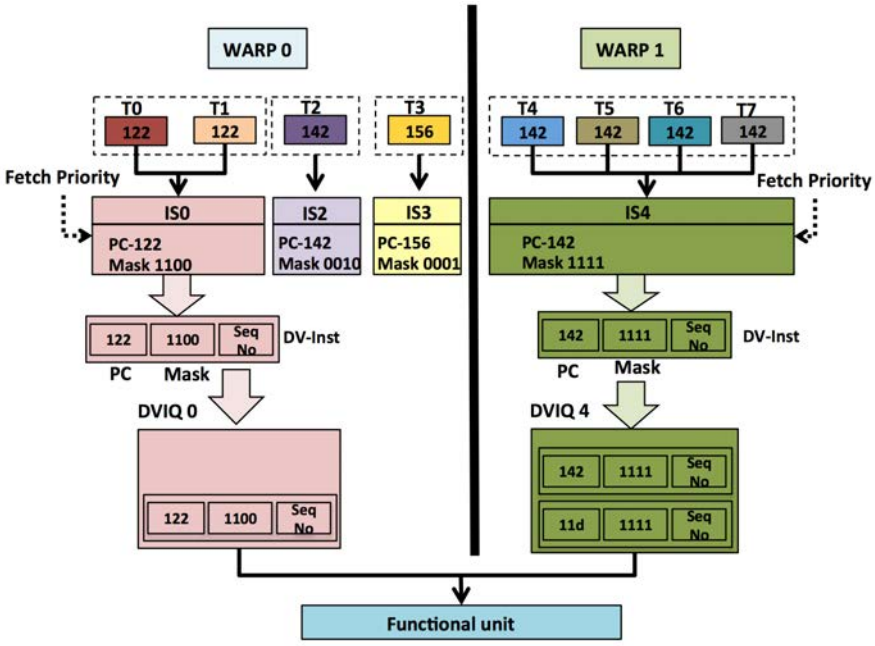
3. **Masks:** Masks determine the lane activity of warps. This ensures correctness of the program execution. Also, reduced lane activity (eg: register reads) will reduce the energy requirements of DITVA.

4. **DVIQ:** Dynamic steering of vectorized instruction enables the parallel execution of independent execution paths with the help of stackless reconvergent heuristics.

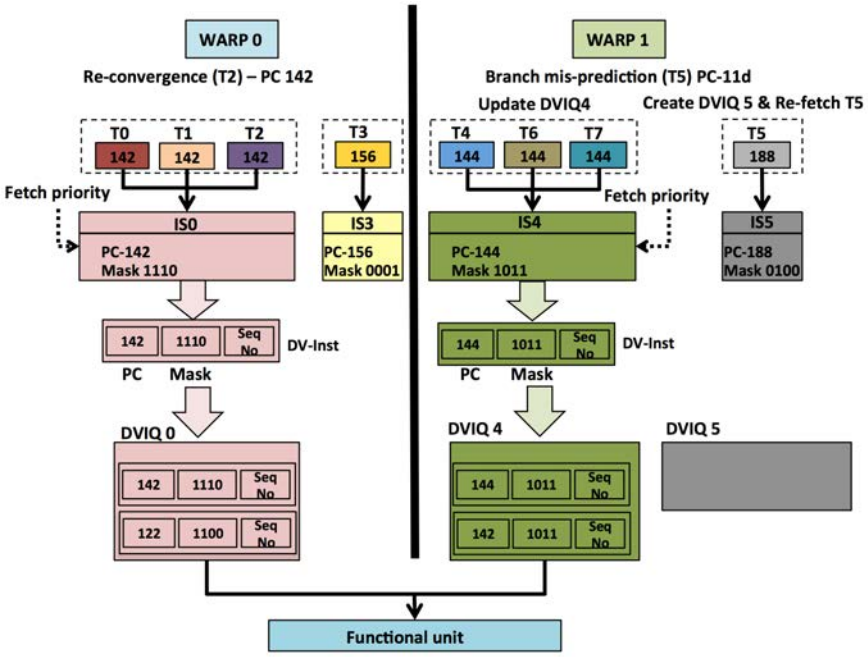
5. **Banked registers:** Smart banking of registers allows efficient utilization of SIMD units of execution.

6. **Banked memory:** Hashing algorithm used for bank assignment reduces the bank conflict. Bank conflict in DITVA is an important issue since the data may be used by the threads in a vectorized instruction may be read from non-contiguous memory location, unlike static vectorization.

In chapter 4, we demonstrate that DITVA indeed provides, performance and energy benefits in comparison an SMT architecture. We also show that DITVA is able to achieve this without significant hardware overhead.



(a) Initial state



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

Figure 3.10: Evolution of the IS and DVIQ states upon thread divergence and reconvergence

Chapter 4

Evaluation

In chapter 3, we discussed DITVA and its implementation. In this chapter, we discuss DITVA-SIM, an in-house cycle accurate simulator to simulate DITVA and SMT to evaluate its performance and design tradeoffs. We also simulate the cost of implementation and the energy profiles of DITVA in comparison with SMT, using CACTI [TMAJ08] and McPAT [LAS+09].

4.1 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.

4.1.1 DITVA-SIM

We model DITVA using an in-house trace-driven x86_64 simulator, DITVA-SIM. DITVA-SIM consists of two components. A trace generator and DITVA simulator. DITVA-SIM can simulate both SMT as well as DITVA. An $mW \times 1T$ corresponds to an m-way SMT. The trace generator uses Intel Pin [BCC+10,

LCM+05] to record one execution trace per thread of one SPMD application.

Intel Pin Pin allows inserting arbitrary code in the executable while it is running. Pin works like a just in time (JIT) compiler that has an executable as the input. Using the instruction sequence from the running program, the pin generates a new executable core. The generated code sequence may contain user injected code. For the execution of user injected code, pin branches out the execution to the user injected code and it regains the control after the completion of its execution.

Pin comprises of two components, instrumentation and analysis code, which is defined in an executable plugin called pintool. Pintool registers instrumentation callback routine, it inspects static properties to decide where and what analysis code is inserted. Analysis code gathers application data. Pintool can also register callback routines for events such as thread creation. For our study, DITVA-SIM records traces at the time of thread creation. DITV-SIM stores the pre-decoded instructions in a file. The index of the executed instruction in the pre-decoded instruction file is stored in an execution trace file. Decoded information are static and hence storing the index to the decoded instructions prevent the creation of large trace files and to reduce the memory requirements. Along with these, DITVA-SIM records other dynamic information such as data memory access, branch information etc.

DITVA Simulator DITV-SIM is a cycle-accurate simulator. 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. Since DITV-SIM is a trace-based simulator, it cannot simulate the wrong path after a branch instruction. To simulate branch prediction, TAGE [Sez11] predicts the taken or not taken direction when a branch instruction is encountered. This is compared against the against the actual direction taken by the execution path. When there is a misprediction, the mispredicted threads are blocked until the branch is resolved.

Thread synchronization primitives such as locks need a special handling in this multi-thread trace-driven approach since they affect thread scheduling. When a barrier is encountered, the threads wait until all other threads hit the barrier. We record all calls to synchronization primitives and enforce their behavior in

Table 4.1: 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 [Sez11]
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

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.

4.1.2 Evaluations using DITVA-SIM

Just like SMT, DITVA can be used as a building block in a multi-core processor. However, to prevent multi-core scalability issues from affecting the analysis, we focus on the micro-architecture comparison of a single core in this study. To account for memory bandwidth contention effects in a multi-core environment, we simulate a throughput-limited memory with 2 GB/s of DRAM bandwidth per core. This corresponds to a compute/bandwidth ratio of 32 Flops per byte in the $4W \times 4T$ DITVA configuration, which is representative of current multi-core architectures. We compare two DITVA core configurations against a baseline SMT processor core with AVX units. Table 4.1 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 [BKSL08] and Rodinia [CBM+09] suites. We use PARSEC benchmark applications that have been parallelized with pthread library. We considered the OpenMP version of the

Table 4.2: 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

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 for Rodinia benchmarks are shown in Table 4.2.

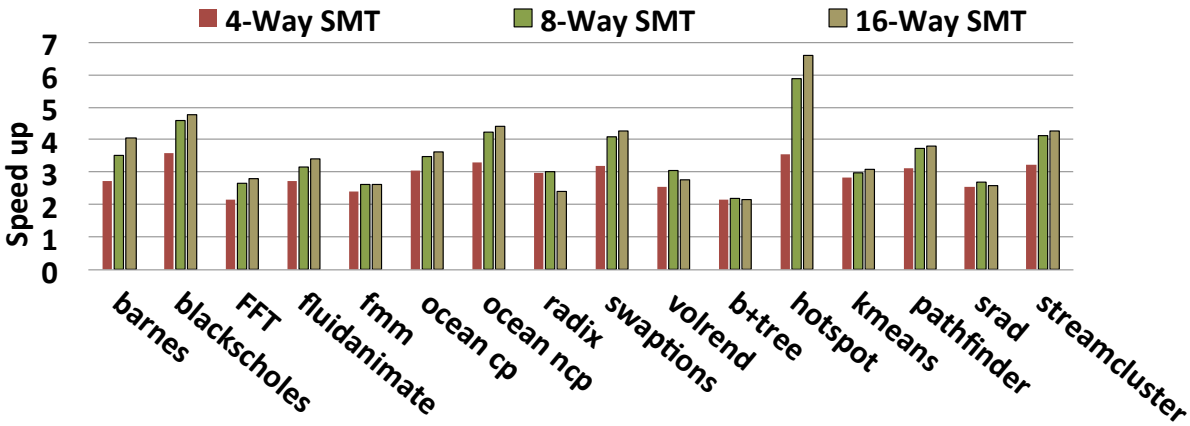


Figure 4.1: Speedup with thread count in the baseline SMT configuration, normalized to single-thread performance

Figure 4.1 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.

4.2 Performance evaluation

4.2.1 Throughput

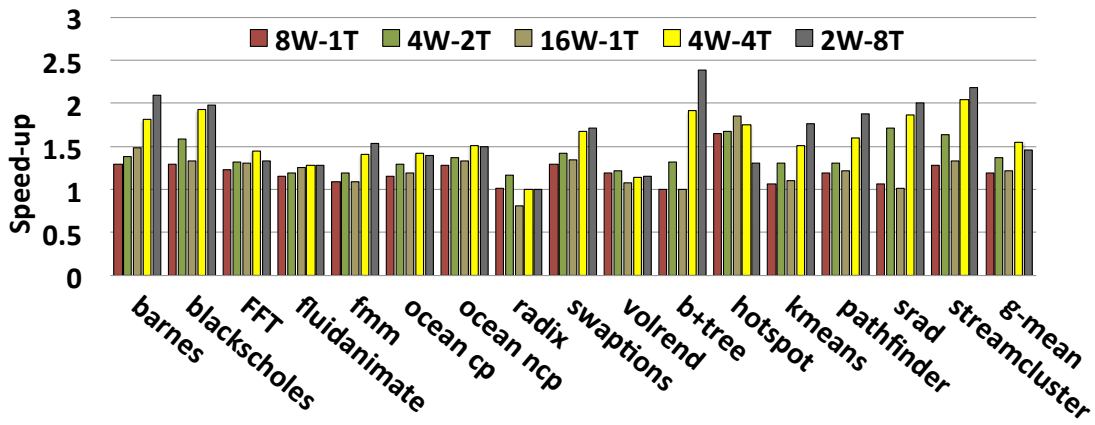


Figure 4.2: Speedup over 4-thread SMT as a function of warp size

Figure 4.2 shows the speedup 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. 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 speedup 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.

4.2.2 Divergence and mispredictions

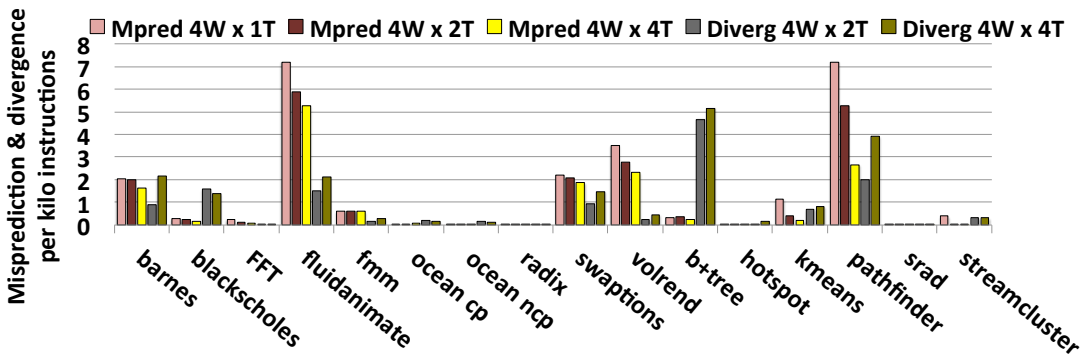


Figure 4.3: Divergence and mispredictions per thousand instructions

Figure 4.3 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 reconvergence, 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.

4.2.3 Impact of split data TLB

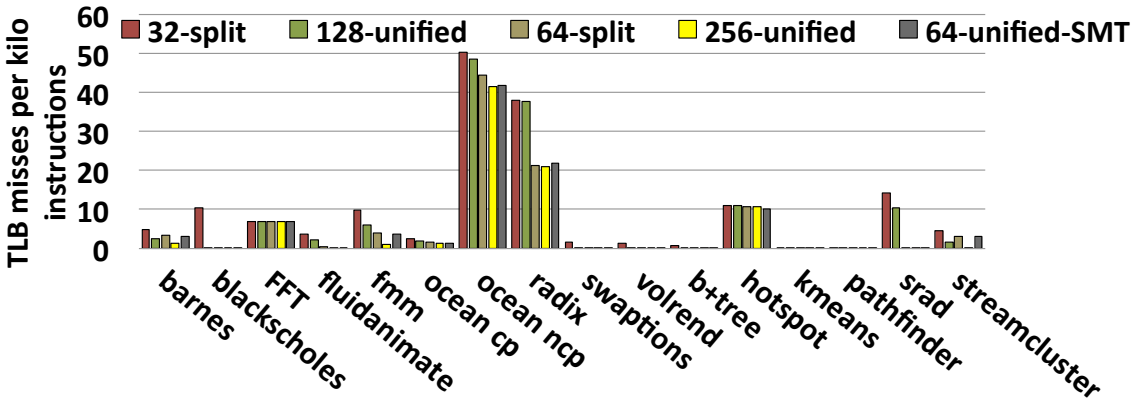


Figure 4.4: TLB misses per thousand instructions for split or unified TLBs on $4W \times 4T$ DITVA

As pointed out in Section 3.4, there is no need to maintain equal contents for the TLBs of the distinct lanes. Assuming a 4KB page size, Figure 4.4 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.

4.2.4 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 4.5. 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 [MS09, MCP+14].

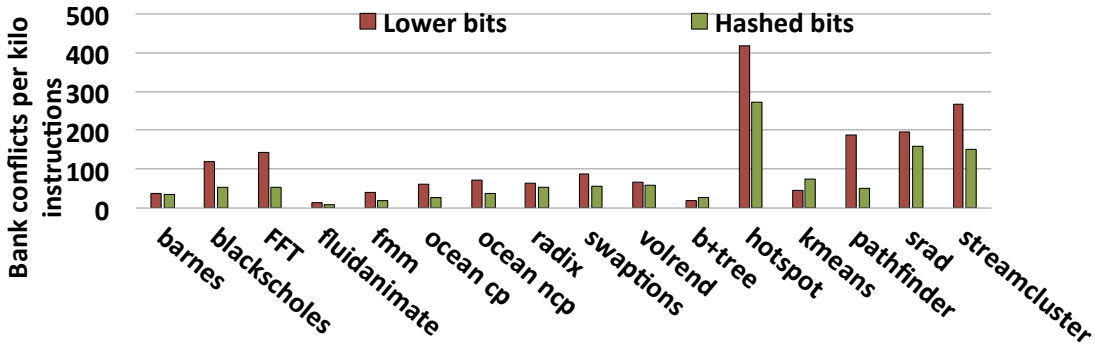


Figure 4.5: Bank conflicts for $4W \times 4T$ DITVA

To reduce such bank conflicts for DV-loads and DV-stores, we use a hashed set index as introduced in Section 3.4. 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 4.5 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.

4.2.5 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 [O'C14]. The performance scaling of 16 GB/s relative to 4-thread SMT with 2 GB/s DRAM bandwidth is illustrated in Figure 4.6.

For many benchmarks, 2 GB/s bandwidth is sufficient. However, as discussed in Section 4.1, the performance of *Srad*, *Hotspot*, *Ocean*, *Radix* and *FFT* is bound

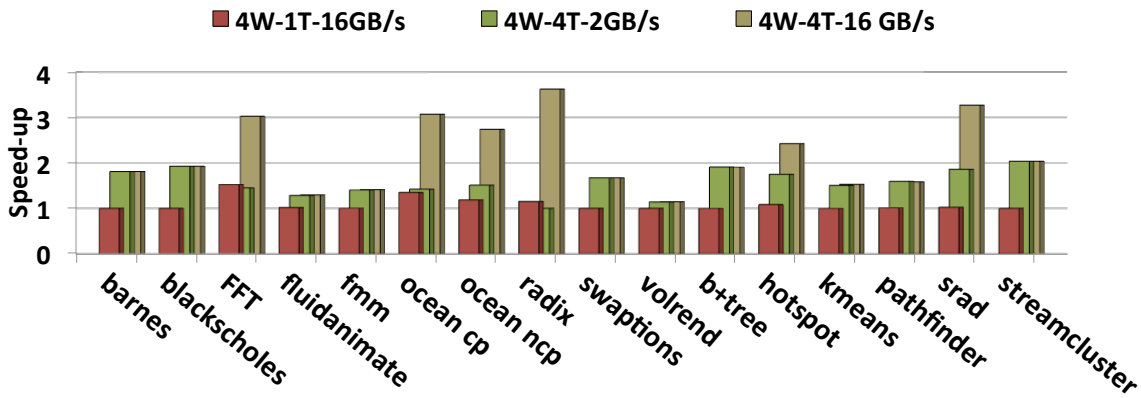


Figure 4.6: Performance scaling with memory bandwidth, relative to 4-thread SMT with 2 GB/s DRAM bandwidth

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.

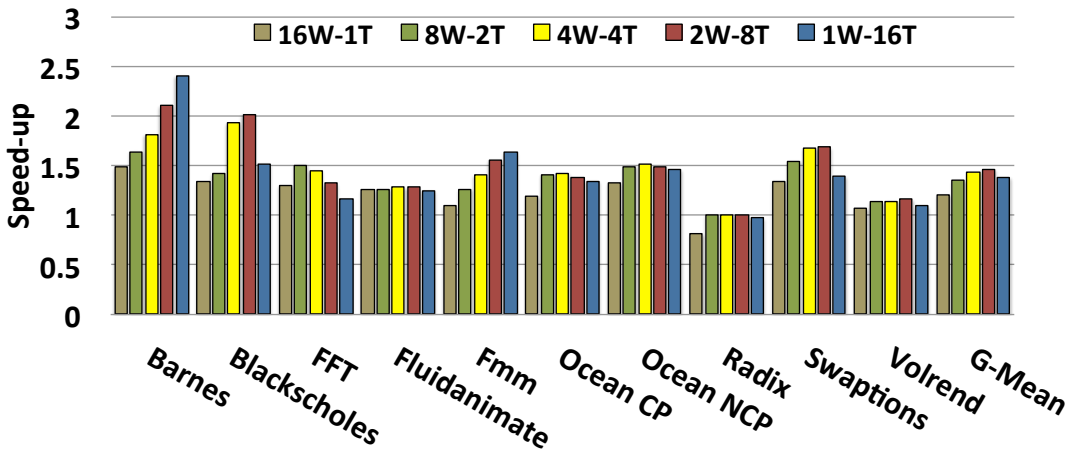


Figure 4.7: Impact of warp size

4.2.6 Impact of Warp size

Figure 4.7 illustrated the impact of warp sizes on a 16-thread DITVA. For *Barnes* and *Fmm* we observe an increase in speedup with the warp sizes. *FFT*, on the other hand, shows a decreasing trend with increasing warp sizes even with near perfect vectorization. This slow down is because any stall in the pipeline will affect the entire warp. *Blackscholes* and *Swaptions* shows a slightly different behavior. The speedup tend to increase up until $2W \times 8T$, but they slow down with 16 thread warp. The reason is similar to what we observe in *FFT*, but with lesser efficiency in vectorization (as compared to *FFT*), multiple DV-threads in a warp compensates for the DV-thread stalls. Even though, *Radix* also has good vectorization, we do not observe this behavior in *Radix* because the memory bandwidth is already a bottleneck in smaller warp sizes. For the same reason we do not observe large variations in *Ocean* benchmarks.

4.2.7 Impact of banked DV-SIMD register banking

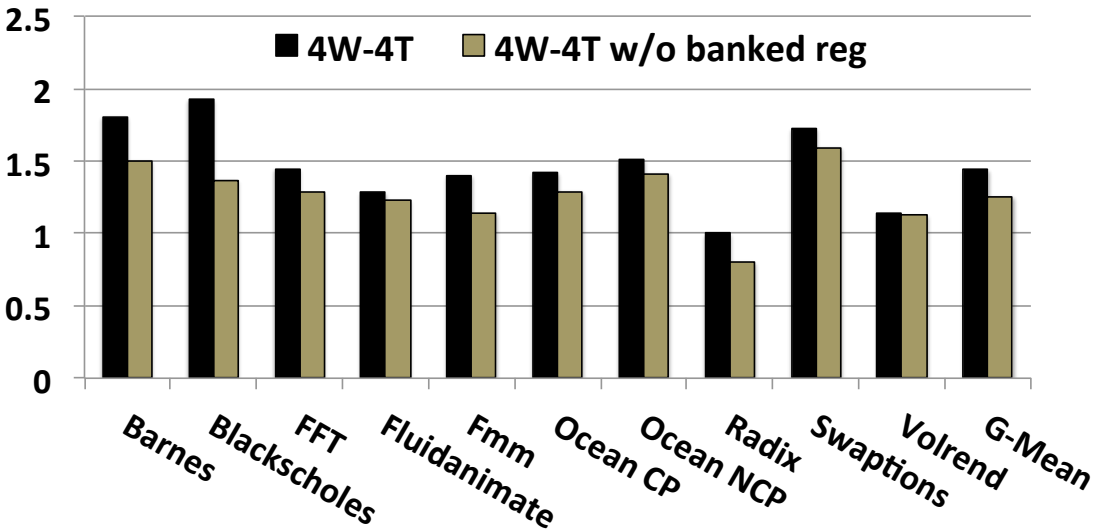


Figure 4.8: Impact of register banking

Register banking enables efficient utilization of 256-bit datapath with contiguous execution of banked lanes. Figure 4.8 shows the speedup of applications in a

$4W \times 4T$ DITVA without banked registers. All the SSE and AVX instructions are serialized at the backend for execution.

We observe a slowdown for almost all the applications. On an average, we observe a 20% slowdown for a $4W \times 4T$ DITVA without register banking with with a maximum slowdown of 55% for *Blackscholes*.

4.3 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.

4.3.1 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 [TMAJ08] and McPAT [LAS+09] for 45nm technology 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.

4.3.2 Quantitative evaluation

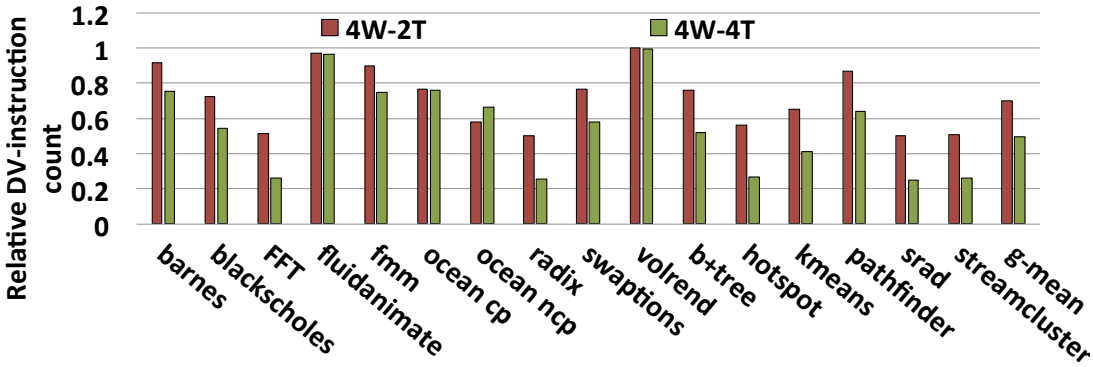


Figure 4.9: DV-instruction count reduction over 4-thread SMT as a function of warp size

Dynamic vectorization reduces the number of DV-instructions over original instructions. Figure 4.9 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 69% (31% reduction) for $4W \times 2T$ DITVA and 49% (51% reduction) 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 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 [LAS+09]. 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 4.1, 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 4.10 while modeled silicon area and static energy are reported in Table 4.3. As in Section 4, 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.

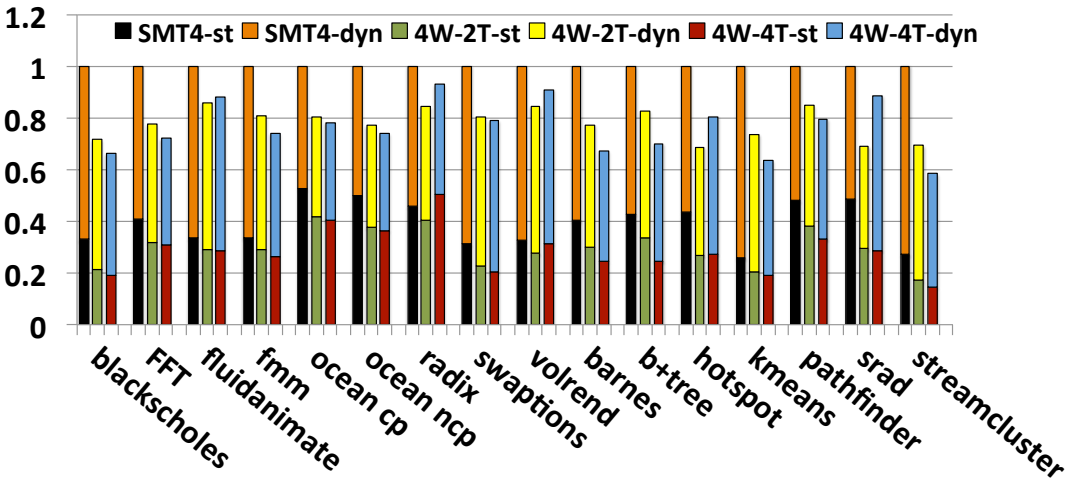


Figure 4.10: Relative energy consumption over 4-thread SMT. *-dyn* and *-st* are dynamic and static energy, respectively.

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 [XJB+15]. 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 4.10 and Table 4.3 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 4.2) and a reduction in the number of fetched instructions, mitigated by an increase in static power from the wider execution units.

4.4 Conclusion

SMT implementations have limited issue width scalability due to increase in hardware complexity. We use clustering In DITVA to decrease the complexity of

Table 4.3: 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

hardware. In this chapter, we have seen that with a 4-issue SMT, the thread count does not have good scalability beyond 8 threads. With an increase in area of around 12% have shown that $4W \times 4T$DITVA is a scalable solution that improves the speedup by 55% on average. We also showed the trade-off for varying warp sizes. Memory intensitive applications tends to suffer with larger warp sizes and lesser warp count. $4W \times 4T$ DITVA reduces the overall energy consumption by 24%. We also showed that DITVA memory accesses suffer due to memory bank conflicts. We provided a solution that decreases the bank conflicts. For memory intensive applications, increasing in memory bandwidth to 16 GB/s immensely the performance of DITVA in comparison with 16 GB/s SMT. We showed that branch predictions work well with DITVA. However, other performance optimization techniques such as prefetching and value predictions have not been evaluated.

For applications with good vectorization ratio, DITVA have lower dynamic energy. However, the overall static energy increase due to the increase in physical resources. Hence, DITVA is less effective in terms of overall energy consumption in the case of applications that has bad vectorization ratio.

Chapter 5

Future work

In this thesis, we have seen Dynamic Inter-Thread Vectorization Architecture for an in-order SMT processor. Modern microprocessors incorporate out-of-order issue to improve the processor throughput. Out-of-order processor fetches and commits the instruction in-order. However, independent instructions may be scheduled out-of-order for execution even if previous instructions have not been issued. Out-of-order processors may improve the single thread performance. DITVA is capable of out-of-order issue of the dynamically vectorized instructions. We have seen in figure 4.1 that the throughput of the processor saturates when there are enough threads to keep the resources busy. On the other hand, out-of-order processors starts to saturate at a lower thread count [HS99]. Hence, OOO-DITVA is expected to have better single thread performance in comparison with in-order DITVA. In this chapter, we will discuss an out-of-order issue extension for the DITVA architecture.

5.1 Out-of-order execution

A conventional in-order processor design maintains the order of instructions at every stage of the pipeline. The instructions are fetched from the memory in the order of the program flow (often speculated by the branch predictor). After moving through the intermediate pipeline stages, the instructions are checked for the availability of input operands. If all the input operands are available, the instruction is dispatched to the appropriate functional unit. On the other

hand, if one or more operands is not available, the instruction stream is stalled until they are available. At the end of the pipeline, the state of the processor is updated. Throughout these stages, the instructions maintain a strict sequential order as defined by the order of instructions in the program. This technique is called *static scheduling* as the instruction scheduling is based on the instruction flow produced by the compiler.

In-order processor design works with the disadvantage that would block the instruction fetch when there are too many long latency instruction operations in the pipeline. And the subsequent instructions, even if they do not cause any hazards, are still not scheduled for execution thus inhibiting achievement of peak performance.

Out-of-order processor attempts to execute the independent instructions with *dynamic scheduling*. The out-of-order processor enables dynamic scheduling with the help of a reservation station (or issue queue). In an out-of-order processor, the instructions, after decoding, waits in the reservation station until all its input operands are available. An independent instruction may have its operand ready even before an earlier instruction. Hence, in an out-of-order processor, the instruction scheduling may not follow the program order. In the following sections, we will discuss the required changes in the pipeline to enable out-of-order execution of instructions.

5.1.1 Register renaming

Out-of-order processors improve the peak performance of a microprocessor by dynamically scheduling independent instructions out-of-order. For dynamic scheduling, the consumer instruction of a data has to be linked with its producer and at the same time should avoid any possible data hazards 1.1. In out-of-order processors, these dependencies are handled by register renaming.

Register renaming replaces the architectural register names with a new name (tag) for each output operand from a free list. This would eliminate output and anti-dependencies and will be left with only true dependencies. With register renaming, at any given time, the pipeline may have an architectural register renamed more than once. The state of the processor is updated when an instruction is committed in-order. The renamed register of the committed instruction may be still in use until a new version of the register is committed. Hence, at commit,

a previous version of the renamed register is released.

5.1.2 Reservation station / Issue queue

A Reservation station is an instruction buffer where the renamed instructions wait until all its input operands are available. When a value is produced, the tags are broadcasted. The instructions waiting in the reservation station, which has its input operands renamed to source tags compares them against the broadcast tags. When there is a match, the source operand is marked as ready. When all the source operands are available, the instruction is *woken up* and is dispatched to the available functional units.

5.1.3 Reorder buffer

Re-order buffer (ROB) allows instructions to be committed in-order, thus updating the system state in the program order. Commit state updates the registers and the memory. Until the instruction commit, the write result may be stored in the re-order buffer. In addition to the write result, the re-order buffer may have a valid bit, destination address as well as the instruction type. ROB enables precise exceptions. When there is a misprediction the instructions prior to the mispredicted instruction in the ROB are flushed, reservation stations are re-initialized and register files are recovered. ROB ensures that the speculated instructions are not committed.

5.1.4 Branch misprediction

Branch misprediction penalties limit the performance of a deeply pipelined microprocessor. Execution of the branch instruction determines the outcome of a branch. Handling branch misprediction in an in-order microprocessor is fairly straight forward. All the instruction following the branch is removed from the pipeline. In an out-of-order processor, instructions after the branch may complete the execution even before branch resolution. Moreover, after the misprediction, the architectural state including rename table and the register values of the process, has to be restored to the valid state.

Out-of-order processors use a Retirement Register File (RRF) that holds the committed state of the process which cannot be rolled back. Out-of-order processors may have multiple unresolved branches in the ROB. Hence, out-of-order processors use checkpointing for RAT recovery on branch misprediction. A copy of RAT is created at each branch instruction. A checkpoint is released as soon as a branch is resolved. On misprediction, the instructions are flushed, checkpoints are deallocated and the RAT is recovered to a valid checkpoint. Subsequent renaming is valid as the RAT is a snapshot of the state just before the last mispredicted branch. Similarly, as the instructions after the mispredicted branch are not committed, RRF also remain valid.

5.2 Out-of-order extension for DITVA architecture

In this section, we will discuss OOO-DITVA, an extension for DITVA to support out-of-order issue. OOO-DITVA requires changes in the pipeline to support register management and instruction wake up as the registers are renamed and independent instructions might be ready to issue out-of-order when their source operands are available. Per-lane register renaming is expensive as the rename unit has to be replicated for each lane. For an nW×mT OOO-DITVA, there will be m rename units and the number of rename operations are increased by (m-1)×. Similarly, with traditional out-of-order SMT design, during convergent execution, the complexity of instruction wake-up increases as a single DV-instruction of an nW×mT OOO-DITVA could have up to m× operands, that will be ready at the same time. Thus to use the traditional out-of-order SMT design, ROB would require several additional ports. In order to avoid these complexities, our OOO-DITVA proposes a single rename for the threads in an instruction stream.

In addition to the pipeline architecture changes, OOO-DITVA uses a *Merge* instruction for register compaction and reconvergence. In the following subsections, we will discuss various pipeline architecture changes in OOO-DITVA.

5.2.1 Register management

We have seen in section 5.1.1 register renaming is one of the techniques that enables out-of-order execution.

A-Reg	D-Mask				P-reg			
r1	1111				p1			
r2	1000	0111			p2	p4		
r3	1000	0110	0001		p3	p5	p6	
r4	1111				p7			

Figure 5.1: OOO-DITVA RAT & RRAT

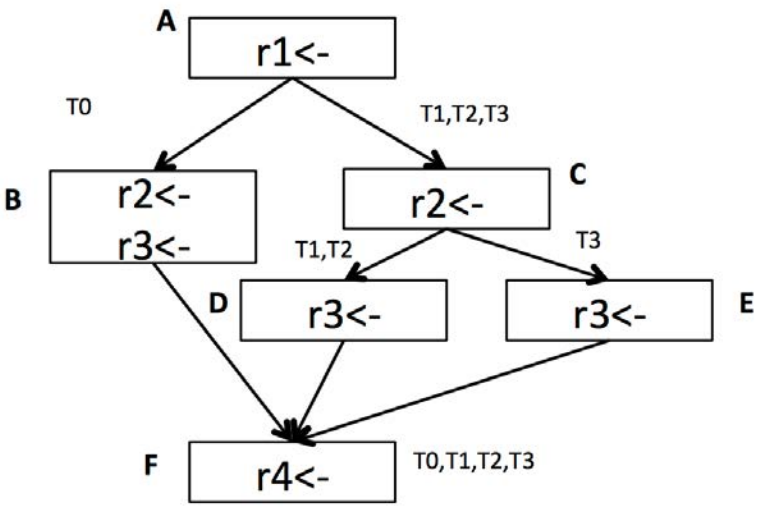


Figure 5.2: Program control flow for threads T0,T1,T2 and T3. <- represents write to a register.

Figure 5.1 shows the state of Register Alias Table (RAT) for an nW×4T DITVA with control flow shown in figure 5.2. In addition to architectural to physical register mapping, OOO-DITVA keeps the mask information (D-Mask)

in the RAT. Each mask in D-Mask represents a divergent path. The physical register corresponding to a mask is given in *P-reg* column. OOO-DITVA has partitioned RAT, with n partitions, each corresponding to a warp. The logical representation of the physical register file with data from threads T0, T1, T2 and T3 occupying lanes L0, L1, L2 and L3 respectively, are shown in figure 5.3. In OOO-DITVA, the physical register free list is shared by all the warps.

	L0	L1	L2	L3
p0				
p1	T0	T1	T2	T3
p2	T0			
p3	T0			
p4		T1	T2	T3
p5		T1	T2	
p6				T3
p7	T0	T1	T2	T3

Figure 5.3: Physical register file in OOO-DITVA

5.2.1.1 Register allocation

When renaming encounters a destination register, a new physical register is allocated from the free list (and subsequently removed from the free list) and the speculative mask corresponding to the fetched DV-instruction is inserted in the RAT. When there is a branch misprediction, these speculative masks are updated to correspond the divergent paths of the instruction streams after the execution of branch instruction. Since each mask in D-Mask represents a divergent path, there are no overlapping set bits in any of the masks in the D-Mask of an architectural register and there will be exactly one set bit corresponding to each lane, across the D-Masks. In the subsequent instructions, the source registers referring to the

same architectural registers are renamed to a valid physical register allocated by previous destination register reference. The source registers are renamed by an 'AND' operation of DV-instruction mask with each mask in D-Mask. There will exactly one non-zero value for the 'AND' operation. P-reg corresponding to the mask is assigned as the renamed register for the source architectural register.

5.2.1.2 Register deallocation

As we have seen in section 5.1.1, physical registers are deallocated when there is a new version of the destination register. The current version of the committed architectural register is maintained in Retirement Register Alias Table (RRAT). RRAT in OOO-DITVA is similar to RAT, with architectural to physical register mapping and non-speculative D-Mask. When there is a new destination register corresponding to an architectural register, a 'NAND' operation in done on DV-instruction mask with each mask in D-Mask and the resulting mask is updated in the corresponding D-Mask. If the result of 'NAND' operation is zero then the older version of P-reg corresponding to the mask is deallocated and moved to the physical register free list. Subsequently, RRAT is updated with the mask and physical register of the committing DV-instruction.

5.2.1.3 Handling divergence and reconvergence

OOO-DITVA does a single architectural to physical register rename for a DV-instruction. With divergence and reconvergence, two issue arises. Firstly, with divergence, a previous result may be consumed by multiple threads in divergent paths. For instance, in figure 5.2, r1 produced in block A may be consumed by r2<-r1+1 in block B and block C. At the rename stage no special care has to be taken for divergence because DV-instruction consuming the result will be marked *ready-to-issue* only after the instruction producing the result completes its execution. However, at commit stage, when the divergent path is committing a new version of the architectural register, the older version cannot be deallocated immediately. To deallocate the older version of the register, all the divergent paths should have a new version. This case is covered by the 'NAND' operation at the commit stage, which will produce a zero only when all the paths have a new version.

The second case to consider is the reconvergence point. In this case, a DV-

instruction after reconvergence may consume the results produced by more than one divergent paths. For instance, in figure 5.2, r4<-r3+1 in block F consumes r3 from block B, block D and block F. To avoid this issue, OOO-DITVA inserts one or more (one for each reconvergent path) *merge instructions* at the reconvergence point. A *merge instruction* merges multiple registers. The merge instructions are generated by the microarchitecture whenever there is a reconvergence. Intel has a similar implementation for merging registers [B14] (this may be further optimized). When a *Merge {r3}* instruction is applied to the execution state shown in figure 5.1, the RAT is updated with a newly allocated p8 physical register and all the physical registers merges with p8<-{p3,p5,p6}. Subsequent consumers of r3 will have the source register renamed to p8. At the commit of the *merge instruction*, p3,p5,and p6 registers are deallocated.

5.2.2 Handling branch misprediction

After a branch instruction, all the threads in the current path takes the speculative path determined by the branch predictor. A subset of the threads in the speculative path may be mispredicted resulting in a partial misprediction. A full misprediction occurs when all the threads in the speculative path are mispredicted.

OOO-DITVA uses a checkpointed RAT for its recovery in the case of a branch misprediction. When a full misprediction is encountered, instructions in ROB that are fetched after the mispredicted branch are flushed, the RAT checkpoints after the branch is deallocated and the RAT is recovered to a valid checkpoint.

5.3 Conclusion

Out-of-order execution is an architectural technique that is common in a modern day microprocessor. It is very effective to provide high single thread performance at a moderate hardware cost. With DITVA, we modelled an architecture with an in-order core model because of its simplicity and low implementation cost. In this section, we provided some insights for out-of-order implementation of DITVA. OOO-DITVA could help to achieve high single thread performance. OOO-DITVA may achieve high performance with lesser number of warps. This section discussed some of the implementation issues of OOO-DITVA, most importantly register

renaming and result broadcast. We leave the performance evaluation, including the speedup analysis, hardware costs and implication on energy due to register merges and discarded register reads for mispredicted paths, as a future work.

Conclusion

Many parallel applications are developed using the SPMD programming model. On those applications, the parallel sections are executed by parallel threads executing the same code. In-order SMT cores represent a cost-effective design point to achieve high throughput on the parallel sections. The parallel threads often exhibit very similar control flows, i.e., execute essentially the same instruction sequences, but on different data inputs: they offer implicit data-level parallelism. GPUs can exploit these characteristics to maximize the throughput of massively parallel SPMD kernels, but demand extensive application re-writing.

DITVA provides a range of design tradeoffs between an in-order SMT core and a SIMT GPU core. It adopts the dynamic vectorization capabilities of GPUs while retaining the general-purpose capabilities and low latency of conventional CPUs. Design parameters of DITVA may be configured to achieve different tradeoffs between SMT and SIMT. Warp size and warp count are the natural vehicle to adjust the tradeoff between throughput on regular kernels and latency on irregular workloads.

Compared to 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 has been designed to execute the same instruction from different threads running in lockstep mode whenever possible. It mutualizes the pipeline front-end for the threads. To enable parallel execution in lockstep mode, the register file of a superscalar processor and its functional units are replicated.

Unlike SIMT GPUs, DITVA can run general-purpose parallel applications written in the ubiquitous SPMD style. Applications require no source modification nor re-compilation. While GPUs are heavily optimized for throughput at the expense of latency, DITVA seeks a more balanced tradeoff, and its latency stays

competitive with an equivalent SMT CPU. To achieve its latency target, DITVA uses 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, we avoid most of the complexity associated with tracking and predicting thread divergence and reconvergence.

SPMD applications written for multi-core processors often exhibit less inter-thread memory locality than GPU workloads, due to coarser-grain threading and avoidance of false sharing. Design of the DITVA memory access unit requires special care to prevent memory accesses from becoming a bottleneck. SPMD applications are sharing data, therefore the data cache is not replicated. As a fully multi-ported cache would be too complex, DITVA implements a bank-interleaved cache and its execution is stalled on bank conflicts. DITVA leverages the possibility to use TLBs with different contents for the different threads. It uses replicated but smaller TLBs than the one of an in-order SMT core.

Replicated functional units and replicated data TLBs are the main hardware overhead of DITVA over an in-order SMT core. However, thanks to masked execution, on a given SPMD workload, the dynamic energy consumption of the pipeline back-end is not increased, and even decreased since DITVA uses smaller data TLBs than an in-order SMT core.

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 [FSYA09] or Simultaneous Branch Interweaving [BCD12], Dynamic Warp Subdivision [MTS10b] could improve latency tolerance by allowing threads to diverge on partial cache misses, and Dynamic Scalarization [CDZ09] could further unify redundant data-flow across threads.

Bibliography

[AF01] Aneesh Aggarwal and Manoj Franklin. An empirical study of the scalability aspects of instruction distribution algorithms for clustered processors. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, pages 172–179. IEEE, 2001.

[ALKK90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. April: a processor architecture for multiprocessing. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 104–114. IEEE, 1990.

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[B14] Tamar B. Migrating from sse2 vector operations to avx2 vector operations, 2014.

[BBSG11] Michael Butler, Leslie Barnes, Debjit Das Sarma, and Bob Gelinas. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, (2):6–15, 2011.

[BCC+10] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, et al. Analyzing parallel programs with pin. *Computer*, 43(3):34–41, 2010.

[BCD12] 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.

[BDA03] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. *ACM SIGARCH Computer Architecture News*, 31(2):275–287, 2003.

[BEKK00] John M Borkenhagen, Richard J Eickemeyer, Ronald N Kalla, and Steven R Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, 2000.

[BFS12] Alexander Branover, Denis Foley, and Maurice Steinman. Amd fusion apu: Llano. *Ieee Micro*, (2):28–37, 2012.

[BKSL08] 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.

[BM00] Amirali Baniasadi and Andreas Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 337–347. IEEE, 2000.

[BRI+90] David Budde, Robert Riches, Michael T Imel, Glen Myers, and Konrad Lai. Register scorboarding on a microprocessor chip, January 2 1990. US Patent 4,891,753.

[C+95] IEEE Portable Applications Standards Committee et al. Ieee std 1003.1 c-1995, threads extensions, 1995.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[CDZ09] Sylvain Collange, David Defour, and Yao Zhang. Dynamic detection of uniform and affine vectors in gpgpu computations. In *European Conference on Parallel Processing*, pages 46–55. Springer, 2009.

[ci7] Intel® core™i7-5960x processor extreme edition. `http://ark.intel.com/products/82930`. Accessed: 2016-02-22.

[Col11] Sylvain Collange. Stack-less simt reconvergence at low cost. Technical report, Technical Report HAL-00622654, INRIA, 2011.

[CPG00] Ramon Canal, Joan Manuel Parcerisa, and Antonio González. Dynamic cluster assignment mechanisms. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 133–142. IEEE, 2000.

[CRVF04] 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.

[CT04] Jamison D Collins and Dean M Tullsen. Clustered multithreaded architectures-pursuing both ipc and cycle time. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 76. IEEE, 2004.

[DFR10] Mark Dechene, Elliott Forbes, and Eric Rotenberg. Multithreaded instruction sharing. *Department of Electrical and Computer Engineering, North Carolina State University, Tech. Rep*, 2010.

[DKW+11] Gregory Diamos, Andrew Kerr, Haicheng Wu, Sudhakar Yalamanchili, Benjamin Ashbaugh, and Subramaniam Maiyuran. SIMD reconvergence at thread frontiers. In *MICRO 44: Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, December 2011.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[DRBL74] Robert H Dennard, VL Rideout, E Bassous, and AR Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[DS02] Romain Dolbeau and André Seznec. Cash: Revisiting hardware sharing in single-chip parallel processor. 2002.

[EA03] 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.

[ECX+11] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M Blackburn, and Kathryn S McKinley. Looking back on the language and hardware revolutions: measured power, performance, and scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 319–332. ACM, 2011.

[EE07] 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.

[EJK+96] Richard J Eickemeyer, Ross E Johnson, Steven R Kunkel, Mark S Squillante, and Shiafun Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *ACM SIGARCH Computer Architecture News*, volume 24, pages 203–212. ACM, 1996.

[FBJ+08] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 2008.

[Fly66] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[Fos95] Ian Foster. Designing and building parallel programs, 1995.

[FSYA07] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420. IEEE Computer Society, 2007.

[FSYA09] 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.

[gcc] Auto-vectorization in gcc. https://gcc.gnu.org/projects/tree-ssa/vectorization.html. Accessed: 2016-01-13.

[GCC+08] José González, Qiong Cai, Pedro Chaparro, Grigorios Magklis, Ryan Rakvic, and Antonio González. Thread fusion. In *Proceedings of the 2008 international symposium on Low Power Electronics & Design*, pages 363–368. ACM, 2008.

[Gre11] Peter Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, pages 1–8, 2011.

[HS96] 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.

[HS99] 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.

[icc] A guide to vectorization with intel® c++ compilers. https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGu Accessed: 2016-01-13.

[int] Intel® intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGui Accessed: 2016-01-13.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, 2005.

[KCSS16] Sajith Kalathingal, Sylvain Collange, Bharath Narasimha Swamy, and André Seznec. Dynamic inter-thread vectorization architecture: extracting dlp from tlp. In *International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, 2016.

[KDT+12] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. Extending openmp* with vector constructs for modern multicore simd architectures. In *International Workshop on OpenMP*, pages 59–72. Springer, 2012.

[Kes99] Richard E Kessler. The alpha 21264 microprocessor. *Micro, IEEE*, 19(2):24–36, 1999.

[KHL99] Bradley C Kuszmaul, Dana S Henry, and Gabriel H Loh. A comparison of scalable superscalar processors. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 126–137. ACM, 1999.

[KJT04] Rakesh Kumar, Norman P Jouppi, and Dean M Tullsen. Conjoined-core chip multiprocessing. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 195–206. IEEE Computer Society, 2004.

[KP93] Ronun Keryell and Nicolas Paris. Activity counter: New optimization for the dynamic scheduling of simd control flow. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 2, pages 184–187. IEEE, 1993.

[KT98] Venkata Krishnan and Josep Torrellas. A clustered approach to multithreaded processors. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 627–634. IEEE, 1998.

[KTJR05] Rakesh Kumar, Dean M Tullsen, Norman P Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, (11):32–38, 2005.

[KWm12] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[LAS+09] 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.

[LCM+05] 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.

[Lee97] Ruby Lee. Multimedia extensions for general-purpose processors. In *Proc. IEEE Workshop on Signal Processing Systems*, pages 9–23, 1997.

[LFB+10] Guoping Long, Diana Franklin, Susmit Biswas, Pablo Ortiz, Jason Oberg, Dongrui Fan, and Frederic T Chong. Minimal multithreading: Finding and removing redundant instructions in multithreaded processors. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 337–348. IEEE Computer Society, 2010.

[LFMS01] 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.

[LGF01] 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.

[LKB14] 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.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, (2):39–55, 2008.

[LP84] Adam Levinthal and Thomas Porter. Chap-a simd graphics processor. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 77–82. ACM, 1984.

[LSJ84] Raymond A Lorie and Hovey R Strong Jr. Method for conditional branch execution in simd vector processors, March 6 1984. US Patent 4,435,758.

[M+98] Gordon E Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[MB05] Cameron McNairy and Rohit Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE micro*, (2):10–20, 2005.

[MBW14] Michael Mckeown, Jonathan Balkind, and David Wentzlaff. Execution drafting: Energy efficiency through computation deduplication. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 432–444. IEEE Computer Society, 2014.

[MCP+14] 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.

[MDKS12] 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.

[MGG+11] 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.

[MLN07] Douglas Miles, Brent Leback, and David Norton. Optimizing application performance on x64 processor-based systems with pgi compilers and tools. Technical report, Technical report, The Portland Group, 2008.[cited at p. 33], 2007.

[Moo] Gordon moore: The man whose name means progress. `http://spectrum.ieee.org/computing/hardware/gordon-moore-the-man-whose-name-means-progress`. Accessed: 2016-02-23.

[MPV93] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th annual international symposium on Microarchitecture*, pages 202–213. IEEE Computer Society Press, 1993.

[MS09] 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.

[MTS10a] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 235–246. ACM, 2010.

[MTS10b] 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.

[Mun09] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[Nai04] Dorit Naishlos. Autovectorization in gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[ND10a] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, (2):56–69, 2010.

[ND10b] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, March 2010.

[Nvi07] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[NZ06] Dorit Nuzman and Ayal Zaks. Autovectorization in gcc–two years later. In *Proceedings of the 2006 GCC Developers Summit*, pages 145–158. Citeseer, 2006.

[O'C14] Mike O'Connor. Highlights of the high-bandwidth memory (hbm) standard. In *Memory Forum Workshop*, 2014.

[ONH+96] Kunle Olukotun, Basem A Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *ACM Sigplan Notices*, 31(9):2–11, 1996.

[PAB+05] D Pham, S Asano, M Bolliger, MN Day, HP Hofstee, C Johns, J Kahle, A Kameyama, J Keaty, Y Masubuchi, et al. The design and implementation of a first-generation cell processor-a multi-core soc. In *Integrated Circuit Design and Technology, 2005. ICICDT 2005. 2005 International Conference on*, pages 49–52. IEEE, 2005.

[PG13] ARM Peter Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7, 2013.

[PH13] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[PJS97] Subbarao Palacharla, Norman P Jouppi, and James E Smith. *Complexity-effective superscalar processors*, volume 25. ACM, 1997.

[PM12] Matt Pharr and William R Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[PWH+08] Ishwar Parulkar, Alan Wood, James C Hoe, Babak Falsafi, Sarita V Adve, Josep Torrellas, and Subhasish Mitra. Opensparc: An open platform for hardware reliability experimentation. In *Fourth Workshop on Silicon Errors in Logic-System Effects (SELSE)*. Citeseer, 2008.

[QHJ88] 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.

[R13] James R. Intel® avx-512 instructions, 2013.

[R14] James R. Additional intel® avx-512 instructions, 2014.

[RDK+00] Scott Rixner, William J Dally, Brucek Khailany, Peter Mattson, Ujval J Kapasi, and John D Owens. Register organization for media processing. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 375–386. IEEE, 2000.

[Rus78] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978.

[SBB+07] Manish Shah, J Barren, Jeff Brooks, Robert Golla, Gregory Grohoski, Nils Gura, Rick Hetherington, Paul Jordan, Mark Luttrell, Christopher Olson, et al. Ultrasparc t2: A highly-treaded, power-efficient, sparc soc. In *Solid-State Circuits Conference, 2007. ASSCC'07. IEEE Asian*, pages 22–25. IEEE, 2007.

[SBCVE90] Rafael Saavedra-Barrera, D Culler, and Thorsten Von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, pages 169–178. ACM, 1990.

[Sez11] 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.

[SFKS02] 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.

[SGS10] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[SKR00] Jagannath Keshava Srinivas K. Raman, Vladimir Pentkovski. Implementing streaming simd extensions on thethepentium iii processor. 2000.

[Smi82] Burton J Smith. Architecture and applications of the hep multiprocessor computer system. In *25th Annual Technical Symposium*, pages 241–248. International Society for Optics and Photonics, 1982.

[Tak97] Yoshizo Takahashi. A mechanism for simd execution of spmd programs. In *High Performance Computing on the Information Superhighway, 1997. HPC Asia'97*, pages 529–534. IEEE, 1997.

[TE94] Radhika Thekkath and Susan J Eggers. The effectiveness of multiple hardware contexts. In *ACM SIGPLAN Notices*, volume 29, pages 328–337. ACM, 1994.

[TEE+96a] 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.

[TEE+96b] 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.

[TEL95] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403. ACM, 1995.

[TMAJ08] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. Cacti 5.1. Technical report, HP Laboratories, 2008.

[Tom67] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[TT03] Nathan Tuck and Dean M Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 26–34. IEEE, 2003.

[Uni06] University of Washington. Splash-2, 2006. Benchmark package.

[Upt] Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal Q*, 1:2002.

[XJB+15] 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.

List of Figures

1.1 Source of parallelism in a program 15

1.2 Flynn's taxonomy and MISD model 16

1.3 Classification of architectures based in thread count 16

1.4 Instruction pipelining 20

1.5 Superscalar execution (degree 2) 21

1.6 32-bit integer execution in 4-wide superscalar processor and a SSE unit 27

1.7 SIMT 28

1.8 SIMT 28

1.9 SIMT divergence 29

1.10 Thread execution without multi-threading 30

1.11 Coarse grained multi-threading (After cycle 1 T1 is scheduled as there is a long latency operation, T0 is rescheduled only when there is a long latency operation for T1) 31

1.12 Fine grained multi-threading (Rescheduling T0 do not wait until there is a very long latency operation for T1) 32

1.13 Simultaneous multi-threading 32

2.1 Control flow graph of blackscholes benchmark 41

2.2 Overview of dynamic vectorization 42

2.3 Breakdown of average vectorizable instructions for 4-way SMT . . 42

2.4 Control flow 48

2.5 Growth of stack in time for threads T0 and T1 48

2.6 Control flow with synchronization point at Func_d 49

3.1 Overview of DV-instruction formation in DITVA 56

3.2 Speed up of single warp of 16 threads and 16 threads grouped into 4 warps with dynamic vectorization using DITVA 57

3.3 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). 58

3.4 Thread mapping to the physical organization. 59

3.5 Overview of a $2W \times 2T$, 4-issue DITVA pipeline. Main changes from SMT are highlighted. 61

3.6 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. . 62

3.7 Illustration of instructions from a scalar thread occupying multiple DVIQ's 64

3.8 Instruction issue with sequence number 65

3.9 Operand collection on $4W \times 4T$ DITVA depending on DV-instruction type and execution mask. 'w' represents a 64-bit word 67

3.10 Evolution of the IS and DVIQ states upon thread divergence and reconvergence 75

4.1 Speedup with thread count in the baseline SMT configuration, normalized to single-thread performance 80

4.2 Speedup over 4-thread SMT as a function of warp size 81

4.3 Divergence and mispredictions per thousand instructions 82

4.4 TLB misses per thousand instructions for split or unified TLBs on $4W \times 4T$ DITVA 83

4.5 Bank conflicts for $4W \times 4T$ DITVA 84
4.6 Performance scaling with memory bandwidth, relative to 4-thread SMT with 2 GB/s DRAM bandwidth 85
4.7 Impact of warp size 85
4.8 Impact of register banking 86
4.9 DV-instruction count reduction over 4-thread SMT as a function of warp size 89
4.10 Relative energy consumption over 4-thread SMT. -*dyn* and -*st* are dynamic and static energy, respectively. 90

5.1 OOO-DITVA RAT & RRAT 97
5.2 Program control flow for threads T0,T1,T2 and T3. <- represents write to a register. 97
5.3 Physical register file in OOO-DITVA 98

Abstract

Many modern microprocessors implement Simultaneous Multi-Threading(SMT) to improve the overall efficiency of superscalar CPU. SMT hides long latency operations by executing instructions from multiple threads simultaneously. SMT may execute threads of different processes, threads of the same processes or any combination of them. When the threads are from the same process, they often execute the same instructions with different data most of the time, especially in the case of Single-Program Multiple Data (SPMD) applications.

Traditional SMT architecture exploit thread-level parallelism and with the use of SIMD execution units, they also support explicit data-level parallelism. SIMD execution is power efficient as the total number of instructions required to execute a complete program is significantly reduced. This instruction reduction is a factor of the width of SIMD execution units and the vectorization efficiency. Static vectorization efficiency depends on the programmer skill and the compiler. Often, the programs are not optimized for vectorization and hence it results in inefficient static vectorization by the compiler.

In this thesis, we propose the Dynamic Inter-Thread vectorization Architecture (DITVA) to leverage the implicit data-level parallelism in SPMD applications by assembling dynamic vector instructions at runtime. DITVA optimizes an SIMD-enabled in-order SMT processor with inter-thread vectorization execution mode. When the threads are running in lockstep, similar instructions across threads are dynamically vectorized to form a SIMD instruction. The threads in the convergent paths share an instruction stream. When all the threads are in the convergent path, there is only a single stream of instructions. To optimize the performance in such cases, DITVA statically groups threads into fixed-size independently scheduled warps. DITVA leverages existing SIMD units and maintains binary compatibility with existing CPU architectures.

Résumé

De nombreux microprocesseurs modernes mettent en œuvre le multi-threading simultané (SMT) pour améliorer l'efficacité globale des processeurs superscalaires. SMT masque les opérations à longue latence en exécutant les instructions de plusieurs threads simultanément. Lorsque les threads exécutent le même programme (cas des applications SPMD), les mêmes instructions sont souvent exécutées avec des entrées différentes.

Les architectures SMT traditionnelles exploitent le parallélisme entre threads, ainsi que du parallélisme de données explicite au travers d'unités d'exécution SIMD. L'exécution SIMD est efficace en énergie car le nombre total d'instructions nécessaire pour exécuter un programme est significativement réduit. Cette réduction du nombre d'instructions est fonction de la largeur des unités SIMD et de l'efficacité de la vectorisation. L'efficacité de la vectorisation est cependant souvent limitée en pratique.

Dans cette thèse, nous proposons l'architecture de vectorisation dynamique inter-thread (DITVA) pour tirer parti du parallélisme de données implicite des applications SPMD en assemblant dynamiquement des instructions vectorielles à l'exécution. DITVA augmente un processeur à exécution dans l'ordre doté d'unités SIMD en lui ajoutant un mode d'exécution vectorisant entre threads. Lorsque les threads exécutent les mêmes instructions simultanément, DITVA vectorise dynamiquement ces instructions pour assembler des instructions SIMD entre threads. Les threads synchronisés sur le même chemin d'exécution partagent le même flot d'instructions. Pour conserver du parallélisme de threads, DITVA groupe de manière statique les threads en warps ordonnancés indépendamment. DITVA tire parti des unités SIMD existantes et maintient la compatibilité binaire avec les architectures CPU existantes.