Simultaneous Branch and Warp Interweaving for Sustained GPU Performance

Nicolas Brunie, Caroline Collange, Gregory Diamos

Simultaneous Branch and Warp Interweaving for Sustained GPU Performance

Nicolas Brunie
Kalray and ENS de Lyon
nicolas.brunie@kalray.eu

Caroline Collange
Universidade Federal de Minas Gerais
caroline.collange@dcc.ufmg.br

Gregory Diamos
NVIDIA Research
gdiamos@nvidia.com

Abstract

Single-Instruction Multiple-Thread (SIMT) micro-architectures implemented in Graphics Processing Units (GPUs) run fine-grained threads in lockstep by grouping them into units, referred to as warps, to amortize the cost of instruction fetch, decode and control logic over multiple execution units. As individual threads take divergent execution paths, their processing takes place sequentially, defeating part of the efficiency advantage of SIMD execution. We present two complementary techniques that mitigate the impact of thread divergence on SIMT micro-architectures. Both techniques relax the SIMD execution model by allowing two distinct instructions to be scheduled to disjoint subsets of the the same row of execution units, instead of one single instruction. They increase flexibility by providing more thread grouping opportunities than SIMD, while preserving the affinity between threads to avoid introducing extra memory divergence. We consider (1) co-issuing instructions from different divergent paths of the same warp and (2) co-issuing instructions from different warps. To support (1), we introduce a novel thread reconvergence technique that ensures threads are run back in lockstep at control-flow reconvergence points without hindering their ability to run branches in parallel. We propose a lane shuffling technique to allow solution (2) to benefit from inter-warp correlations in divergence patterns. The combination of all these techniques improves performance by 23% on a set of regular GPGPU applications and by 40% on irregular applications, while maintaining the same instruction-fetch and processing-unit resource requirements as the contemporary Fermi GPU architecture.

1. Introduction

Graphics Processing Unit (GPU) architectures have proven successful for general-purpose parallel computing in various applicative areas [27]. GPUs group threads into packets, referred to as warps, and run them in lockstep on SIMD units. Architectures based on SIMD execution offer a potential efficiency advantage by sharing a common pipeline front-end across multiple execution units. However, this execution model only benefits applications whose control flow patterns and memory access patterns present enough regularity. Prior work has shown that the performance potential of GPUs is vastly underexploited by many irregular applications [21, 15, 8].

Our goal is to broaden the applicability of GPU architectures by substantially improving their performance on irregular applications, while maintaining their performance on regular applications. To achieve this goal, we introduce two complementary techniques that tackle the divergence problem. They both reclaim SIMD lanes that would have been left idle due to divergence to run instructions of other threads. The first technique, that we name Simultaneous Branch Interweaving (SBI), leverages the inherent parallelism available between divergent branches to co-issue instructions from the same warp. The second technique, Simultaneous Warp Interweaving (SWI), selects instructions from other partial warps to fill the gaps left by divergence.

The idea behind SBI and SWI comes from realizing that the benefits of SIMD execution arise from amortizing the cost of instruction fetch units, instruction decode units and control logic across many execution units. The two micro-architectures we propose maintain the

same ratio of instruction control units to execution units as some existing GPU architectures, but relax the constraints of SIMD execution to better tolerate thread divergence. Like Simultaneous Multi-Threading (SMT) on superscalar processors [31], SBI and SWI increase functional unit utilization by filling scheduling bubbles thanks to fine-grained resource allocation to multiple threads. Together, they improve performance by 40% on a set of irregular applications, and also accelerate regular applications by 23%.

We will first examine existing GPU micro-architectures and define our baseline model in section 2. We will then consider the SBI technique that simultaneously co-issues instructions from multiple branches, with a focus on ways to achieve thread reconvergence in section 3. The second technique, SWI, will be described in section 4. Finally, we will quantify the performance and overhead of the proposed solutions in section 5 and review related work in section 6.

2. Background

Current-generation GPUs operate according to an implicit SIMD execution model, also called SIMT (Single Instruction, Multiple Threads) by NVIDIA. From the programmer's and compiler's point of view, this model is similar to SPMD (Single Program, Multiple Data). The programmer writes a single program or *kernel*. A large number of instances of the kernel (or *threads*) are then run in parallel. During execution on a GPU, transparent hardware mechanisms group threads into warps, and execute their instructions in lockstep on SIMD units [27]. The execution order of threads is undefined by the programming model unless the programmer uses explicit synchronization primitives.

Each thread may logically follow a differentiated control flow path, even though it physically runs in lockstep with other threads of the warp. To maintain the illusion of differentiated control flow, most GPUs use implicit instruction predication. The context associated with future branches (PC and mask) are stored in a hardware stack [22]. Entries are popped from the stack as control flow reconverges. Reconvergence points are exposed at the architectural level, and are inserted by the compiler.

For scalability reasons, GPUs are made of several independent warp processors, that we name Streaming Multiprocessors (SMs). Multiple warps are maintained in flight in each SM, and their execution is interleaved at the granularity of an instruction for latency tolerance purposes. Recent GPU architectures such as NVIDIA Fermi have several clusters per SM, in a way that is reminiscent of clustered multi-threaded architectures [11, 19]. Each cluster has its own instruction fetch unit and scheduling logic [27]. This enables the sharing of instruction caches, data caches and lesser-used functional units at a coarse granularity, while keeping the warp width small enough to tolerate thread divergence.

Baseline We consider in this paper a baseline SM micro-architecture closely inspired by Fermi, outlined in Figure 1. Like Fermi, it handles branch divergence using a hardware stack. Warps are split into two warp

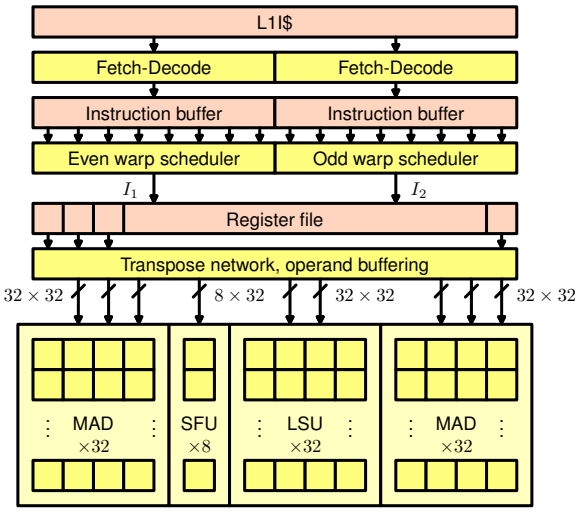


Figure 1. Baseline SM micro-architecture.

pools, respectively made of warps with even identifiers and warps with odd identifiers. Each pool has dedicated independent scheduling resources. Each clock cycle, one instruction from each pool is fetched from the cache and fed into an instruction buffer comprising one entry per warp. Two instruction schedulers pick one ready instruction each from their associated instruction buffer and issue it. Each instruction scheduler selects its oldest instruction [25]. Dependencies between instructions are tracked using a scoreboard. It consists of a table indexed by warp ID, where each entry contains the destination register IDs of the instructions in flight for the warp [7]. When a new instruction is decoded, its source and destination register IDs are compared against the scoreboard entries for its warp. The dependency bit vector that is produced by comparisons is kept in the instruction buffer alongside the decoded instruction. When an instruction is retired, both the scoreboard and the instruction buffer entry are updated to mark the dependency as cleared.

Instructions whose dependency mask reach zero are eligible for scheduling.

The pipeline back-end consists of four groups of SIMD units. To balance throughput and hardware cost, the SIMD width may be lower than the warp width. In this case, the warp is broken down into several waves sent through the pipeline. Operand buffers perform throughput matching between the register file and execution units. The Multiply-Add (MAD) units and Special Function Units (SFU) respectively execute most arithmetic instructions and transcendental functions as in NVIDIA's architectures [27]. The Load-Store Unit (LSU) arbitrates access to a single 128-byte port to the L1 cache. It can coalesce together multiple parallel accesses that fall within the same 128-byte cache block. Memory instructions that encounter conflicts are replayed with an updated activity mask reflecting the transactions that remain to be issued.

We contrast the techniques proposed throughout this paper with the baseline SIMT execution model on a simple example on figure 2. It depicts the execution pipeline when an *if-then-else* block with 2 warps of 4 threads is run. Each instruction is identified by its address, from 1 to 6. The *if* branch contains instructions 2 to 4, and the *else* branch contains instruction 5. SBI improves over the baseline SIMT by simultaneously scheduling instructions from different branches. In figure 2(b), a secondary scheduler issues instruction 5 of the *else* path together with instruction 2. Optional reconvergence constraints can be applied to SBI to re-align threads when control flow reconverges, as with instruction 6 of figure 2(c). SWI issues instructions from other warps, such as instruction 2 of warp 2 together with instruction 3 of warp 1 in figure 2(d). Finally, both techniques can be combined, is in figure 2(e).

3. Simultaneous branch interweaving

In this section, we focus on extracting parallelism from divergent branches of the same warp. Whether a branch is divergent or not is generally only known at runtime: hence, our approach is based on dynamic scheduling. When control flow splits into two branches, current SIMT architectures execute each branch sequentially. However, branch paths are taken by disjoint sets of threads. They are independent from each other and can be executed in any order, including in parallel, without violating the SIMT programming model.

3.1. Enabling parallel branch execution

Decoupling the execution of concurrent branch paths requires additional hardware support beyond conventional stack-based reconvergence. Dynamic Warp Subdivision was proposed to enable the shared instruction scheduler to interleave the execution of instructions from each path [24]. This technique improves memory latency hiding by providing more memory-level parallelism. Instruction throughput is not affected or may decrease, as the execution of individual instructions happens sequentially as in SIMT. Warps are decomposed into multiple *warp-splits*, which correspond to different paths concurrently taken. Each warp-split has its own divergence stack and is scheduled independently from the others. Reconvergence is made possible using a dedicated hardware table and/or by comparing the Program Counters (PCs) of each warp-split. However, current stack-based reconvergence systems do not allow several branches to be executed in parallel. The stack-based algorithm has to be complemented by heuristics, which provide no guarantee on when reconvergence will occur.

To work around these issues, we advocate to switch from stack-based reconvergence to thread-frontier based reconvergence [10]. Diamos et al. have shown that PC-based reconvergence can be made optimal in the sense that it always reconverge at the earliest possible point, given hardware support for a sorted heap and compiler support for laying out the code in the order dictated by thread frontier analysis [10]. It works by always scheduling the warp-split of minimal PC[1]. Like stack-based reconvergence, thread frontier reconvergence runs divergent branches sequentially. However, we show it is amenable to parallel execution by relaxing the scheduling constraints.

We refer to the Common PCs as CPCs to avoid ambiguities. In addition to the first minimum of thread PCs ($\mathrm{CPC}_1 = \min(\mathrm{PC}_i)$), we also compute the second minimum when it exists ($\mathrm{CPC}_2 = \min(\mathrm{PC}_i, \mathrm{PC}_i \neq \mathrm{CPC}_1)$). As a consequence, we have the order $\mathrm{CPC}_1 < \mathrm{CPC}_2 < \mathrm{PC}_i, \mathrm{PC}_i \notin \{\mathrm{CPC}_1, \mathrm{CPC}_2\}$. We will refer to the warp-splits whose PC is CPC_1 and CPC_2 as the primary and the secondary warp-split, respectively. We will assume throughout the next section that both minimums are available at all times. Concrete hardware implementations and their tradeoffs will be described in section 3.4.

[1] We assume in this work that thread-frontier priorities are implicitly encoded in the program order.

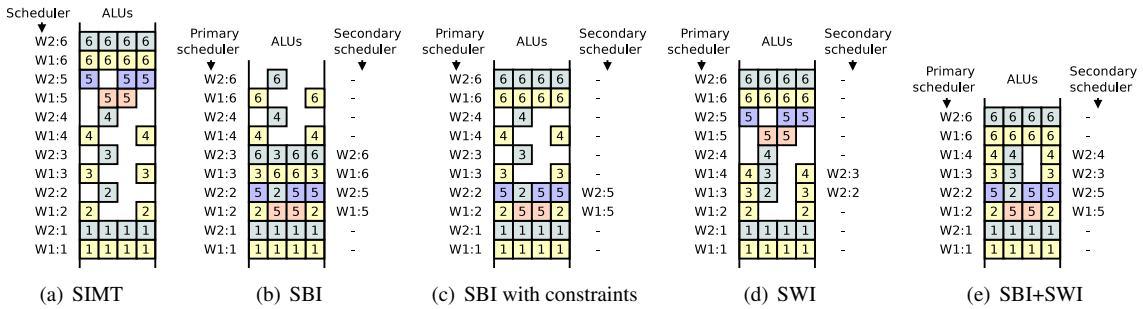


Figure 2. Comparison of the contents of the execution pipeline using classic SIMT, Simultaneous Branch Interleaving with optional contraints, Simultaneous Warp Interleaving, and both.

3.2. SBI architecture

SBI can be implemented with minimal modifications to our baseline architecture if the warp size is doubled to 64 threads. Instead of two warp pools of 32-wide warps, a single warp pool contains entries for two 64-wide warp-splits. The left-hand front-end depicted on figure 3 is essentially unchanged. It schedules instruction I_1 following CPC_1. The second front-end issues instruction I_2 following CPC_2 of the same warp. Instruction delivery is extended to broadcast both instruction I_1 and I_2 to all processing lanes. Per-lane multiplexing selects which instruction to execute according to individual bits in the warp-split predication masks m_1 and m_2.

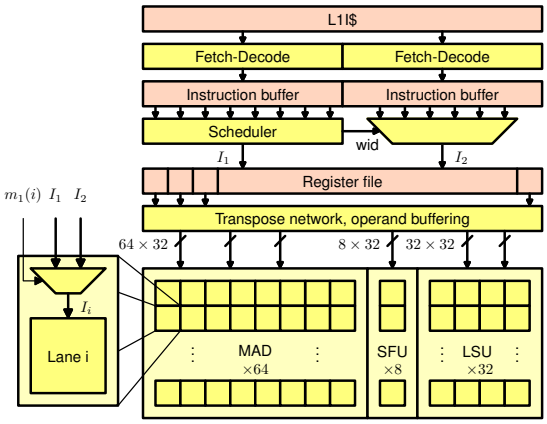


Figure 3. Simultaneous Branch Interweaving micro-architecture.

3.3. Ensuring reconvergence

Warp-split desynchronization Greedy scheduling may delay reconvergence by continuously letting a secondary warp-split run ahead of the primary warp-split, as in figure 2(b). In this example, the secondary scheduler issues instruction 6 early for threads 2 and 3 of warp 1. Several cycles later, the primary scheduler then issues the same instruction for threads 1 and 4. As long as the secondary warp-split does not encounter resource conflicts, warp-splits can continue running out-of-sync from each other. This desynchronization can lead to power consumption increase through redundant instruction fetch, and conflicts for memory resources [20].

Selective synchronization barrier We consider in this section a conservative policy, which prevents parallel execution of divergent branches to happen past the reconvergence point. We emit a synchronization instruction at each reconvergence point. Its payload is the address $\mathrm{PC}_{\mathrm{div}}$ of the divergence point, which we define as the last instruction of the immediate dominator. The thread-frontier aware program layout ensures that each reconvergence point occurs at a higher address in the code than the divergence point [10]. Synchronization instructions are treated as selective synchronization barriers among warp-splits. The secondary warp-split is suspended if $\mathrm{CPC}_1 \in [\mathrm{PC}_{\mathrm{div}}, \mathrm{PC}_{\mathrm{rec}}]$, and can continue otherwise, where $\mathrm{PC}_{\mathrm{rec}}$ is the address of the reconvergence point.

The idea behind selective synchronization is to allow parallel execution across outer branches of nested control-flow blocks, while still ensuring synchronization at the end of inner blocks. We illustrate this idea on the control-flow graph of figure 4, which consists of two nested *if-then-else* blocks. Basic blocks are labeled from

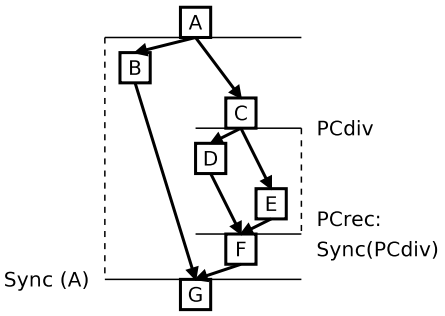


Figure 4. Control flow graph laid out by PC order, with divergence and reconvergence point annotations.

A to G. The vertical axis denotes the program-counter order. We suppose the secondary warp-split has just reached the F block, so that $\mathrm{CPC}_2 = \mathrm{PC}_{\mathrm{rec}}$. It encounters a synchronization instruction that points at $\mathrm{PC}_{\mathrm{div}}$, the last instruction of block C. We distinguish two cases.

1. $\mathrm{PC}_{\mathrm{div}} < \mathrm{CPC}_1 < \mathrm{PC}_{\mathrm{rec}}$. The secondary warp-split is suspended until the primary warp-split, which is currently in D or E, reaches the reconvergence point at the beginning of the F block.

2. $\mathrm{CPC}_1 \leq \mathrm{PC}_{\mathrm{div}}$. As CPCs are strictly ordered, no other warp-split has its PC between $\mathrm{PC}_{\mathrm{div}}$ and $\mathrm{PC}_{\mathrm{rec}}$. Synchronization for the inner *if-then-else* block is achieved, and the execution of the secondary warp-split can continue through F. This way, blocks B and F are free to run in parallel, which does not delay reconvergence.

Though we illustrated this technique on an *if-then-else* nest, it can be applied to other control structures such as loop nests as well. Unstructured control flow may involve multiple divergence points for a single reconvergence point. Our choice of the immediate dominator is conservative as it may place the divergence point earlier than strictly necessary. While it means that opportunities for parallel execution may be missed, it guarantees that synchronization will occur at the reconvergence point.

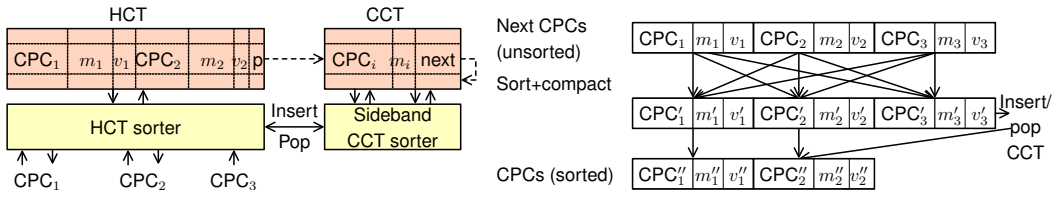
No additional hardware is needed to release warp-splits from their wait-state. As the primary warp-split reaches the synchronization point, its CPC_1 matches the CPC_2 of the secondary warp-split, causing both warp-splits to be merged together as the new primary warp-split.

3.4. Implementation

Sorted heap We use a sorted-heap based implementation as proposed by Fung et al. [15] and Diamos et al. [10] to store warp-splits. Each warp-split context is an entry (CPC, m, v) containing the program counter, the activity mask and a valid bit. Contexts are arranged into a Hot Context Table (HCT) and a Cold Context Table (CCT), as outlined in figure 5(a). The HCT is indexed by the warp identifier and contains the two active contexts of each warp, as well as a pointer to the next inactive context in the CCT. The CCT is organized as a linked list of contexts per warp, where each entry has a pointer to the next element. Another linked list contains free blocks. All lists share the same CCT.

Each table is managed by a separate unit. The HCT sorter unit is responsible for keeping the first two entries of each warp sorted, and merge them as needed. It receives the updated CPC_1 and CPC_2 from the PC update logic. When divergence occurs, it also receives an additional CPC_3 from the branch and memory arbitration logic. We enforce the restriction that at most one divergence (branch or memory) can happen each cycle. It guarantees that at most one new warp-split can be created each cycle. The HCT sorter consists of a sorting network that can compact and merge entries (fig. 5(b)). During the first stage, entries $(\mathrm{CPC}_1, \mathrm{CPC}_2, \mathrm{CPC}_3)$ are sorted, compacted and merged into $\mathrm{CPC}'_1 < \mathrm{CPC}'_2 < \mathrm{CPC}'_3$, and their valid bits are updated. If three valid entries remain after compaction ($v'_3 = 1$), then the third entry is inserted into the CCT. If there is one valid entry only ($v'_2 = 0$), the second entry is popped from the CCT.

Likewise, the CCT is kept sorted by a dedicated unit that also handles insertions. While the HCT runs synchronously with the pipeline, insertions in the CCT are asynchronous. The sideband sorter is a state machine that walks linked lists in the CCT to perform an insertion sort. In the worst case, an insertion can take 64 non-pipelined cycles. However, warp-split order in the heap does not affect execution correctness ; it is only an optimization that avoids unnecessary thread divergence. In case the sideband sorter is unable to keep up with insertions of new warp-split, the sorted heap will be degraded into a stack, where popping the top entry will just return the last inserted entry. Interestingly, this degraded mode matches the behavior of divergence stacks used on today's GPUs. Prior work has also shown that the heap size including hot entries rarely exceed 3 in real programs, so even an $\mathrm{O}(n^2)$ sorting algorithm is adequate [10].



(a) General architecture (b) HCT sorter

Figure 5. Dual context table organization

Scoreboard As warp-splits diverge and reconverge, individual threads may "jump" from one warp-split to the other, creating data dependencies between instructions. Conversely, register read/write dependencies between non-intersecting warp-splits should be ignored. The brute-force approach to dependency tracking would store the execution mask of each instruction in the pipeline alongside its scoreboard entry. To reduce storage requirements, we compute instead dependencies between instructions by taking the transitive closure of the warp-split divergence-convergence graph.

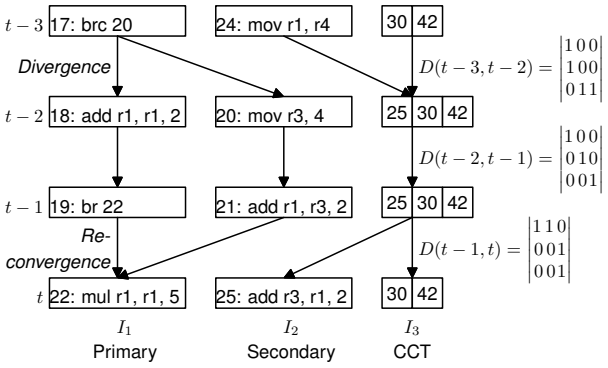


Figure 6. Divergence-convergence graph and dependency matrices for an example instruction sequence.

We define the dependency matrices $D(t_1, t_2)$ between two scheduling cycles t_1 and t_2 such that $D_{i,j}(t_1, t_2)$ is set when common threads execute $I_i(t_1)$ and $I_j(t_2)$. I_3 wraps all inactive entries in the heap. Figure 6 illustrates the dependency matrices on a divergence-convergence graph example. Along with the destination registers of instructions I_1 and I_2, each scoreboard entry k contains the dependency matrix $D(t-k, t)$ with the instructions waiting to be issued. When new instructions are scheduled, dependency matrices are multiplied by the dependency matrix $D(t, t+1)$ of the instruction just issued and shifted by one position. Dependency bits are ANDed together with the result of the register ID comparison to form the dependency mask stored in the instruction buffer. The complexity of the proposed scoreboard only depends on the warp count, pipeline depth and instruction issue width. It is not affected by the warp size.

4. Simultaneous warp interweaving

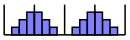
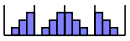
SBI improves throughput when branch paths are balanced, but provides no benefit when the workload of each thread of a warp is unbalanced, as with *if* blocks with no *else* counterparts. We consider in this section Simultaneous Warp Interweaving, the counterpart of SBI that schedules other warps in the gaps left by the first scheduled warp. It is described in isolation with SBI in this section, although both techniques can be combined.

Our asymmetric SM design is based on two cascaded schedulers. As with SBI, the primary scheduler selects a ready instruction I_1 from the instruction buffer. The issue of I_1 is then delayed during one pipeline stage dedicated to secondary scheduling. The secondary scheduler receives the predicate mask m_1 from the initial instruction and looks for a non-conflicting instruction I_2. The secondary instruction may be scheduled to the same SIMD group as the primary instruction, as long as its predicate mask m_2 does not overlap with the primary mask m_1. Alternatively, it can be scheduled to another free SIMD group, as in a conventional multiple-issue SIMD processor. Both instructions I_1 and I_2 are then issued simultaneously to the execution units. The main idea consists in trading scheduler latency for execution unit throughput. The front-end latency increases due to cascaded scheduling. On the other hand, the utilization rate of the back-end execution units benefits from the additional scheduling opportunities created.

Lane shuffling Many parallel algorithms exhibit regular thread imbalance patterns inside each warp. For instance, the first thread of each warp may receive a larger

share of work than its neighbors. Such correlation increases the likeliness of conflicts for execution lanes. To turn the otherwise negative effect of correlations to our advantage, we considered several static thread rearrangement heuristics, that we list on table 1. The physical lane id is computed from the thread-in-warp ID tid and warp ID wid. $\oplus$ is the XOR operator and bitrev is the bit-reversal function. The diagrams on the right illustrate the effect on 4 warps of 4 threads each by plotting the lane ID as a function of $4 \times wid + tid$. Inside each warp, we apply a permutation to the thread-to-lane mapping. At this point, this amounts to changing the way their thread identifier is computed. It requires no additional hardware nor data migration. The mapping

Table 1. Lane shuffle functions.

Name	Function	Illustration
Identity	tid	
MirrorOdd	$n - tid$ if wid odd, tid otherwise	
MirrorHalf	$n - tid$ if $wid > m/2$, tid otherwise	
Xor	$tid \oplus wid$	
XorRev	$tid \oplus \mathrm{bitrev}(wid)$	

functions preserve memory locality and allow the same memory coalescing opportunities as the straightforward mapping. We found that *XorRev* provides the most consistent performance gain across the applications considered (described in section 5).

Scheduler conflict avoidance As both schedulers operate in parallel, the secondary scheduler on phase n may pick the same instructions as the primary scheduler on phase $n + 1$. Rather than prevent this situation by introducing tighter coupling between the two schedulers that would increase their complexity, we detect conflicts *a posteriori*. After a conflict is identified, the instruction copy selected by the primary warp is discarded, and its execution mask set to empty. The other copy is issued on phase n as the secondary warp. During the next cycle, the secondary scheduler is free to select any ready instruction, substituting itself to the primary scheduler.

As conflicts still restrict the scheduling choice and decrease the overall power-efficiency, we ensure they remain unlikely by avoiding correlations between the scheduling policies of each scheduler. Primary scheduling is based on instruction age, while secondary scheduling uses a best-fit policy (maximize occupancy) with pseudo-random tie-breaking.

Limited associativity The secondary scheduler looks for a ready instruction whose mask is a subset of the free lane mask. This can be achieved using a Content-Addressable Memory (CAM). Zero bits in masks are interpreted as "don't care" bits. As CAMs are power-hungry structures [29], we consider set-associative lookup techniques as an alternative. The mask inclusion lookup hardware is similar to a cache tag lookup. Instead of looking for an address, we look for the subset of a mask. In a way similar to caches, tags can be partitioned into multiple sets. A set-associative lookup only searches through elements of a single set. The set index is computed from the low-order bits of the primary warp identifier. The benefits of set-associative lookup are two-fold. First, the need for large and expensive CAMs is avoided. Second, the instruction buffer memory and scoreboard memory can each be partitioned into separate banks, reducing the number of ports of the instruction buffer. Scheduling restrictions from the set-associative design guarantee conflict-free access.

5. Evaluation

5.1. Performance evaluation

Methodology We used a cycle-accurate simulator of the SM pipeline based on the Barra functional simulator [3]. Simulation parameters are listed on table 2. We follow the methodology of Gebhart et al. [16] by modeling a throughput-limited constant-latency memory, but also simulate a local L1 data cache to better take into account memory divergence effects [20]. We add one extra pipeline stage in addition to scheduling stages to account for the wire delay of instruction delivery.

We observed that NVIDIA's CUDA compiler backend would generally lay out code in the exact same order as the order dictated by divergence frontier analysis. In fact, we found only one CUDA kernel for which a different order was used. We include it in our evaluation as the *TMD1* benchmark. This observation matches the results of the analysis conducted on the SPEC INT 2000 by Collins et al. in the context of control-flow reconvergence for out-of-order processors, who find that 94% of reconvergence points are placed below divergent branches [6]. The synchronization instructions described in section 3.3 are placed at the same

Table 2. Micro-architecture parameters.

Parameter	Baseline	SBI	SWI
Warps × warp-width	32×32	16×64	16×64
Clock rate	1 GHz		
Scheduler latency	1 cycle	1 cycle	2 cycles
Instruction delivery latency	0 cycle	1 cycle	1 cycle
Execution latency	8 cycles		
Scoreboard	6 entries / warp		
L1 cache	48K, 6-way, 128B blocks, 3 cycles		
Memory	10 GB/s (1 SM), 330 ns		

addresses as reconvergence markers in the Tesla binary code. Divergence points are computed from Tesla's divergence instructions. We consider benchmarks from CUDA benchmark of Rodinia [2] and applications from the NVIDIA CUDA SDK [28], as well as two implementations of the Table Maker Dilemna (TMD) application in computer arithmetic that exhibit highly irregular control flow [13]. We distinguish regular application, whose average IPC with 64-wide warps is above 30, from irregular applications.

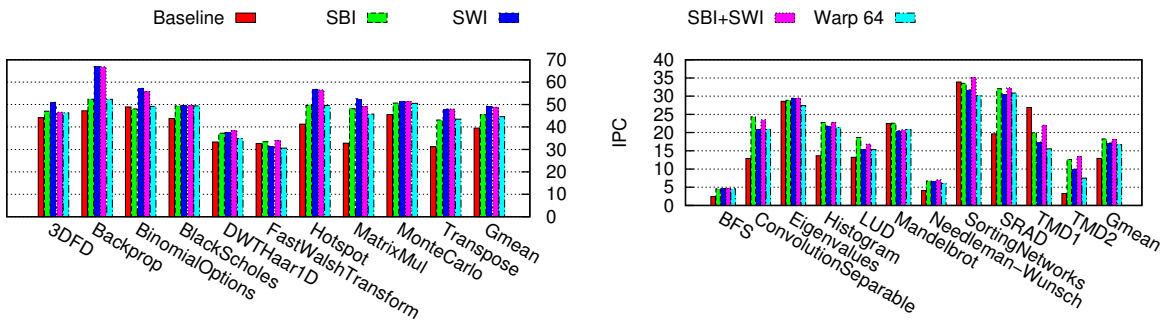
Summary Figure 7 summarizes the relative performance of the baseline architecture, SBI with and without constraints, SWI, and combinations of SBI and SWI. We plot the number of thread instructions per cycle on the SM. The peak IPC of the baseline architecture is 64, as it is limited by the issue rate of two 32-wide warps. SBI and SWI move the peak to 104, as processing units can benefit from the increased relative instruction issue bandwidth. For reference, we consider an implementation based of thread frontiers and 64-wide warps in the comparison. On regular applications, SBI and SWI provide the advantage of wide warps by increasing the relative front-end bandwidth, resulting in average performance increases of 15% and 25%, respectively. In this case, the benefit comes from scheduling more instructions to distinct SIMD groups, rather than warp interweaving per se. The benefits of SBI and SWI are fully realized on irregular applications. Despite the larger warp size and increased front-end latency in the case of SWI, performance is respectively increased by 41% and 33% on average using SBI and SWI.

Specific benchmark discussion *TMD1* performs worse in our experiments, due to the improper code layout issue mentioned above. *TMD2* shows vastly improved performance compared to stack-based execution, even though the latter has dedicated break and return support as the Tesla implementation does [14, 10]. It illustrates the benefits of thread frontier-based reconvergence on unstructured control flow. As the TMD application reflects properties of thread-frontier based reconvergence rather than SBI and SWI, we do not take it into account when computing the performance means. Mandelbrot shows no noticeable performance variation throughout the experiments. We found that a thread block synchronization barrier instruction prevents warp-splits from running ahead across iterations.

Constraints When applied to SBI in isolation, constraints have negligible (less than 0.1%) effect on performance, as seen on figure 8(a). However, constraints reduce the number of instruction issued by 1.3% on regular applications and by 5.5% on irregular applications. SWI takes advantage of the execution resources freed to improve execution throughput : *SortingNetworks* is 2.4% faster when reconvergence constraints are applied to the combination of SBI and SWI. On the other hand, *BFS* and *Histogram* benefit from running threads ahead past reconvergence points and their performance is held back by reconvergence constraints. Overall, the performance impact is modest: we found that most applications have explicit synchronization barriers in their main loop, which enforce warp-split synchronization regardless of reconvergence constraints.

Lane shuffling Figure 8(b) illustrates the effect of the lane shuffling policies listed on table 1 on irregular applications. Speedup of the *XorRev* policy over *Linear* ranges from -1.8% (3dfd, not shown) to +7.7% (Needleman-Wunsch). The geometric mean is respectively +0.3% and +1.4% on regular and irregular applications. While the improvements are small, they come at essentially no extra cost.

Set-associative matching Figure 9 quantifies the performance impact of reduced lookup associativity of SWI on irregular applications. We find that associativity bears a moderate impact on performance: even a direct-mapped configuration achieves at least 85% of the performance of the fully-associative configuration. Compared to the baseline, direct-mapped SWI achieves a speedup of 26%, while fully-associative SWI achieves



(a) Regular applications

(b) Irregular applications

Figure 7. Performance of SBI, SWI, and combination of SBI and SWI, with a thread-frontier based 64-wide warp implementation for reference.

34%. On regular applications, the performance difference is even thinner, as even fully associative lookup finds few warp interweaving opportunities. Direct-mapped SWI achieves 96% of the fully-associative performance, maintaining an 18% speedup over the baseline.

5.2. Hardware overhead

We now estimate the hardware cost of SBI and SWI. Table 3 summarizes the main hardware requirements of each technique. We conservatively assume that the fully-populated CCT can accommodate 8 entries per warp, totaling 10 warp-splits per warp with the HCT. Prior work suggests that the CCT could be made smaller with no significant performance impact [24, 10]. The baseline implementation uses a stack with 3 blocks of 4 entries of 64-bit each per warp.

To deliver two instructions to each lane, the fanout of the instruction broadcast network is increased by a factor of two, and additional wiring is needed. The width of a half-datapath on Fermi is 0.65mm from our measurements on a die photography. This distance is within reach of single clock cycle at 1GHz. The register file needs to be extended to support two distinct addresses. The overhead can be bound by the cost of breaking the register file into one bank per lane, which was estimated at 0.722mm^2 in 90nm for a 8192-entry register file by Fung et al. [15]. Accounting for process scaling and register file size differences, our conservative estimate is 0.57mm^2. We synthesized the other major components of the design using a production RTL compiler with a gate library from a state-of-the-art process technology. To allow comparison with the baseline implementation, area results are scaled to the older 40nm process used by Fermi. Results are reported on table 4. A Fermi SM being 15.6 mm^2 from measurements on a publicly-available die photograph, the respective area overheads of SBI, SWI and both are 3.0%, 2.9% and 3.7%.

6. Related work

Dynamic warp formation (DWF) mitigates the impact of divergence by dynamically building warps [15]. While it improves the utilization of execution units, it may act at the expense of memory divergence and thread reconvergence may be delayed [14, 20]. SBI and SWI preserve memory access locality by keeping threads of the same warp together, and reconvergence constraints guarantee that threads reconverge. DWS [24] was discussed in section 3.1. SBI goes beyond DWS by running concurrent branches simultaneously rather than interleaved in time, with the same execution resources requirements. The reconvergence policy and constraints we propose may be applied to both DWF and DWS.

Block Compaction [14] and Large Warps [26] rely on coarser scheduling units than the warps of today's architectures and compact successive pipeline waves. Our work is orthogonal to these approaches, as SBI and SWI could be used together with compaction of larger warps. Thread frontiers [10] define an optimal execution order of basic blocks that ensures early reconvergence for arbitrary control flow, assuming sequential execution of branch path. We generalize it to parallel execution of basic blocks for the same warp. Control-flow reconvergence was extensively studied in the context of superscalar processors to reduce branch misprediction penalties [6, 1]. Al-Zawawi et al. use a control-flow stack

Table 3. Summary of the hardware requirements for each proposed technique.

Component	Baseline	SBI	SWI	SBI+SWI
RF	Single-decoder	Segmented	Segmented	Segmented
Scoreboard	$2 \times 24 \times 48$-bit	24×144-bit	$2 \times 24 \times 48$-bit	24×288-bit
Scheduler	Symmetric	Warp-split	Associative lookup	Associative lookup
Warp pool/HCT	$2 \times 24 \times 64$-bit	24×201-bit	24×104-bit	24×201-bit, banked
Stack/CCT	144×256-bit	128×104-bit	128×104-bit	128×104-bit
Insn. buffer	48×64-bit	48×64-bit	24×64-bit, dual-ported	48×64-bit, dual-ported

Table 4. Area of each component.

	Area ($\times 1000\ \mu m^2$)			
Component	Baseline	SBI	SWI	SBI+SWI
RF	–	+570	+570	+570
Scoreboard	87.6	65.6	87.6	131.2
Scheduler	–	–	+27.4	+27.4
HCT	66.8	88.8	43.8	88.8
CCT	584.4	480.8	480.8	480.8
Insn. Buffer	52.8	52.8	33.4	67.4
Total	791.6	1258	1243	1365.6
Overhead	–	466.4	451.4	574

and PC comparisons to detect reconvergence in a checkpointing micro-architecture [1]. Stack updates happen in program order, so parallel path traversal was not considered.

Architectures originating from vector processors that blend together thread-level, data-level and instruction-level parallelism have been proposed [12, 18, 30]. These approaches also leverage a data-parallel computing substrate while allowing more flexible means of execution. The problems to address differ between these works and SBI/SWI, as the former use an explicit vector programming model and the latter are based on an SIMD model with implicit predication.

Minimal Multi Threading shares identical instructions across threads in an out-of-order SMT architecture [23]. Control-flow reconvergence is detected by comparing the PC histories of threads. The out-of-order micro-architecture described runs 4 threads, targeting a significantly different design point than the 1536-thread in-order GPU architecture we consider.

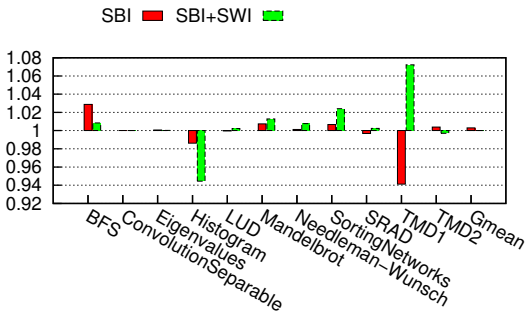
Glew outlines a Dual Instruction, Multiple Threads (DIMT) micro-architecture in a talk [17]. DIMT consists in issuing two different instructions to a wider array of execution units. SBI can be considered as an implementation of the DIMT idea. However, no discussion is made in [17] about concrete implementations, nor about ways to achieve reconvergence. These two issues and SWI are the main contributions of the present work. Dasika et al. propose Divergence-Folding [8], which improves utilization of functional units by statically scheduling instructions from different branch paths to subsets of the same datapath. Divergence-Folding uses two functional units and two RF write ports per lane, while SBI shares the same execution and RF resources and relies on dynamic scheduling.

In addition to sharing instructions, recent works also factor out common computations or common data across threads [4, 5, 9, 23]. We expect that SBI and SWI can be combined with data-sharing techniques to improve their flexibility in the face of data divergence.

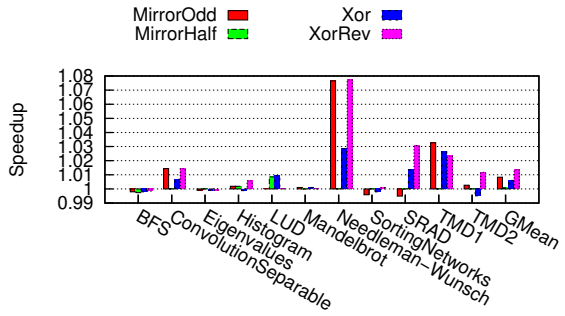
7. Conclusion

We presented and evaluated two complementary techniques that leverage the computational power of GPU architectures that is underexploited due to branch and memory divergence. Each technique performs fine-grained execution resource allocation by incorporating one additional form of parallelism. SBI takes advantage of branch-level parallelism across SIMD instructions that are guaranteed not to overlap. Two problems that arise consist in ensuring the reconvergence of warp-splits and tracking instruction dependencies. We overcome them by respectively introducing reconvergence constraints and an improved scoreboard design. SWI exploits warp-level parallelism. Instructions from different warps are guaranteed to be free of dependencies. The difficulty lies in finding instructions with non-overlapping activity masks. We solve it using a set-associative mask lookup and improve warp affinity using lane shuffling.

Both SBI and SWI execute threads of close identifiers together, in order to preserve memory access patterns and their locality. These techniques rely on full dynamic scheduling and require minimal compiler involvement.



(a) SBI constraints



(b) SWI lane shuffling policy

Figure 8. Effect of constraints and lane shuffling on respective performance of SBI and SWI (irregular applications).

SBI and SWI complement each other. SBI works best on irregular workloads, achieving an average speedup of 41%. Regular workloads benefit most from SWI, which accelerates their execution by 25%. The combination of both techniques retain most of their individual advantages, leading to respective speedups of 40% and 23% on irregular and regular workloads.

Remaining work includes developing robust adaptive policies to schedule warps, perform lane shuffling and decide when running ahead is beneficial. Flexibility may be improved further by allowing more decoupling between lanes, without compromising efficiency. We expect to see more blurring of the lines between SIMT and clustered SMT architectures in the future, as each architecture incorporates successful aspects of the other.

Acknowledgements

We thank Michael Shebanow and Andy Glew for early discussions on the original idea, and Mourad Gouicem and Pierre Fortin for their help with the TMD application. We also thank the anonymous reviewers for their constructive suggestions.

This project was supported by Kalray and ÉNS de Lyon through a research collaboration on advanced computer architecture and parallel processing and its application to the Kalray MPPA® line of products.

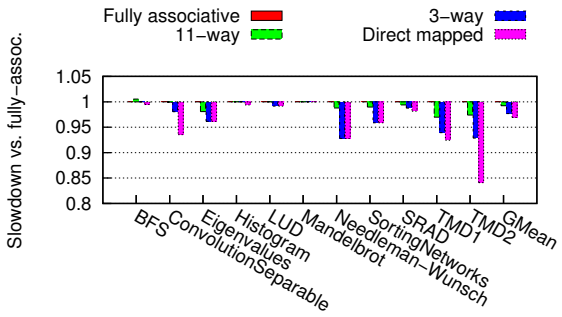


Figure 9. Slowdown of SWI lookup set-associativity compared to fully-associative lookup.

References

[1] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary. Transparent control independence (TCI). *SIGARCH Comput. Archit. News*, 35:448–459, June 2007.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE Workload Characterization Symposium*, 0:44–54, 2009.

[3] C. Collange, M. Daumas, D. Defour, and D. Parello. Barra: a parallel functional simulator for GPGPU. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 351–360, 2010.

[4] C. Collange, D. Defour, and Y. Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Europar 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, volume LNCS 6043, pages 46–55, 2009.

[5] C. Collange and A. Kouyoumdjian. Affine Vector Cache for memory bandwidth savings. Technical Report ensl-00649200, ENS Lyon, Dec. 2011.

[6] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *IEEE/ACM International Symposium on Microarchitecture*, pages 129–140. IEEE Computer Society, 2004.

[7] B. W. Coon, P. C. Mills, S. F. Oberman, and M. Y. Siu. Tracking register usage during multithreaded processing using a scoreboard having separate memory regions

and storing sequential register size indicators. US Patent 7434032, October 2008.

[8] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke. PEPSC: A power-efficient processor for scientific computing. In *PACT*, 2011.

[9] M. Dechene, E. Forbes, and E. Rotenberg. Multithreaded instruction sharing. Technical report, North Carolina State University, 2010.

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

[11] R. Dolbeau and A. Seznec. CASH: Revisiting hardware sharing in single-chip parallel processor. *Journal of Instruction-Level Parallelism*, 6:1–16, 2004.

[12] R. Espasa and M. Valero. Multithreaded vector architectures. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, HPCA'97, pages 237–244, 1997.

[13] P. Fortin, M. Gouicem, and S. Graillat. Towards solving the table maker dilemma on GPU. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 12)*, 2012.

[14] W. Fung and T. Aamodt. Thread block compaction for efficient SIMT control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 25–36, February 2011.

[15] W. W. L. Fung, I. Sham, G. Yuan, and T. 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.

[16] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceeding of the 38th annual international symposium on Computer architecture*, pages 235–246, 2011.

[17] A. Glew. Coherent vector lane threading. *Berkeley ParLab Seminar*, 2009.

[18] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread architecture. *IEEE MICRO*, 24(6):84–90, 2004.

[19] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *IEEE/ACM International Symposium on Microarchitecture*, pages 195–206, 2004.

[20] A. Lashgar and A. Baniasadi. Performance in GPU architectures: Potentials and distances. In *9th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD11), in conjunction with ISCA-38*, 2011.

[21] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460, 2010.

[22] J. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[23] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong. Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 337–348, 2010.

[24] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3):235–246, 2010.

[25] P. C. Mills, J. E. Lindholm, B. W. Coon, G. M. Tarolli, and J. M. Burgess. Scheduler in multi-threaded processor prioritizing instructions passing qualification rule. US Patent 7949855, May 2011.

[26] V. Narasiman, C. J. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU performance via large warps and two-level warp scheduling. In *MICRO 44: Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, December 2011.

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

[28] NVIDIA CUDA SDK, 2010. `http://www.nvidia.com/cuda/`.

[29] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, march 2006.

[30] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector lane threading. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 55–64, 2006.

[31] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23:392–403, May 1995.