CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures

Minsoo Rhu
Electrical and Computer Engineering Department
The University of Texas at Austin
minsoo.rhu@utexas.edu

Mattan Erez
Electrical and Computer Engineering Department
The University of Texas at Austin
mattan.erez@mail.utexas.edu

Abstract

Wide SIMD-based GPUs have evolved into a promising platform for running general purpose workloads. Current programmable GPUs allow even code with irregular control to execute well on their SIMD pipelines. To do this, each SIMD lane is considered to execute a logical thread where hardware ensures that control flow is accurate by automatically applying masked execution. The masked execution, however, often degrades performance because the issue slots of masked lanes are wasted. This degradation can be mitigated by dynamically compacting multiple unmasked threads into a single SIMD unit. This paper proposes a fundamentally new approach to branch compaction that avoids the unnecessary synchronization required by previous techniques and that only stalls threads that are likely to benefit from compaction. Our technique is based on the compaction-adequacy predictor (CAPRI). CAPRI dynamically identifies the compaction-effectiveness of a branch and only stalls threads that are predicted to benefit from compaction. We utilize a simple single-level branch-predictor inspired structure and show that this simple configuration attains a prediction accuracy of 99.8% and 86.6% for non-divergent and divergent workloads, respectively. Our performance evaluation demonstrates that CAPRI consistently outperforms both the baseline design that never attempts compaction and prior work that stalls upon all divergent branches.

1 Introduction

The importance of performance efficiency, in terms of both power and cost, has led to a growing number of processors that support *single-instruction multiple-data* (SIMD), or vector execution. SIMD improves efficiency, and hence potential performance, by minimizing control and partitioning register storage. Some general-purpose processors have recently extended their SIMD width from 4 to 8 [15] with a width of 16 possible in the future [26]. Graphics processors from NVIDIA currently use an instruction vector width of 32 [20] and those from AMD are 64 wide [2]. While the peak performance possible with SIMD is higher, mapping applications with irregular control flow to a SIMD architecture is challenging.

The execution model adopted by GPUs simplifies mapping irregular applications to SIMD compared to traditional vector/SIMD designs. GPUs have hardware support for conditional branches, which allows each SIMD lane to act as a logical thread. NVIDIA refers to this execution as *single-instruction multiple-thread* (SIMT) because logically each SIMD lane represents a different thread of control. When a conditional branch requires some lanes to execute the true path and other lanes to execute the false path, performance degrades. This is because only a single instruction is issued to all SIMD lanes, implying that only a subset of the lanes should actually execute operations and commit results. Recent research has shown that the impact of this *control divergence problem* can be reduced by dynamically forming SIMD-instructions from large collections of threads [8, 19]. These collections of threads are called *cooperating thread arrays* (CTAs) or *thread blocks* by NVIDIA's CUDA [21] and *workgroups* by OpenCL [3]. The SIMD instructions on GPUs are known as *warps* in CUDA and *wavefronts* in OpenCL, and the process of forming new warps after a branch is referred to as *compaction*. Compaction hardware identifies threads that follow the same control flow and groups them together to better fill the SIMD lanes.

In this paper we address a significant issue with previously proposed compaction mechanisms [8, 19] that hinders their effectiveness. In order to identify candidates for compaction, hardware stalls *all* warps within a CTA on *any* potentially divergent branch until all warps reach the branch point. The compaction only occurs once all threads are ready to continue past the branch. When effective, compaction improves performance, but the barrier synchronization, which is not required for correctness, decreases performance in some cases. We observe that this synchronization overhead is common even when there are *no benefits to be gained from compaction*; compaction is *ineffective* when execution does not actually diverge (all conditions within a warp resolve in the same direction) or when the pattern of divergence is aligned across warps, which prevents compaction. We show that the recently proposed *thread block compaction* mechanism [8], for example, unnecessarily delays an average of 88% of warps in divergent-prone appli-

This is the authors version of the work. The authoritative version will appear in the Proceedings of ISCA 2012.

cations and 99.9% in non-divergent workloads. In many cases performance is still increased, but in some cases performance degrades by up to 19.6% compared to a baseline design with no compaction.

We use the above insight about unnecessarily delayed warps to design a new approach to compaction that avoids unnecessary synchronization. The proposed mechanism only stalls those warps that are likely to benefit from compaction while allowing other warps to *bypass* compaction and continue executing. Our mechanism relies on a *compaction-adequacy predictor* (CAPRI), which dynamically classifies branches and warps based on the likelihood that compaction will be effective. Using CAPRI, the compaction hardware filters out non-divergent control flow, whether from unconditional branches or similarly-resolved conditional branches, without software support. In addition, CAPRI identifies divergent branches that are unlikely to benefit from compaction and allows warps to continue executing unhindered. To achieve this, CAPRI tracks compaction-effectiveness using low-cost hardware structures inspired by branch predictors, which we show can achieve an average prediction accuracy of 86.6% and 99.8% on a set of divergent and non-divergent benchmarks, respectively. To summarize our main contributions:

- We identify unnecessary synchronization as a significant challenge to effectively mitigating the performance degradation due to control divergence in GPUs. We analyze this phenomenon and show examples of branch instructions and warps that diverge but do not benefit from compaction.
- We introduce the concept of compaction-adequacy prediction and develop the CAPRI microarchitecture. CAPRI is a pure hardware mechanism that does not require support from the compiler or application and is able to dynamically identify which warps should wait for compaction and which should continue executing.
- We qualitatively and quantitatively evaluate CAPRI in terms of prediction quality, compaction-effectiveness, and performance using representative GPU workloads. We compare CAPRI to previously proposed state-of-the-art compaction mechanisms [8, 19]. For divergent applications, on which prior techniques do well, CAPRI provides an additional 7.6% performance boost. Moreover, CAPRI avoids the 10.1% average performance degradation observed for non-divergent workloads when applying prior compaction techniques [8, 19]. For non-divergent applications, CAPRI matches the performance of the baseline architecture to within ±1%.

2 Background and Related Work

A modern GPU, such as NVIDIA's Fermi [20] or the Cayman GPU of AMD's Radeon HD 6900 series [2], consist of multiple cores ("streaming multiprocessors" and "SIMD units" in NVIDIA and AMD terminology, respectively), where each core contains a number of parallel lanes (referred to as "CUDA cores" and "SIMD cores" by NVIDIA and AMD, respectively) that operate in SIMD/vector fashion: a single instruction is issued to each of the parallel lanes and that instruction is repeated multiple times per lane (vector-style pipelining). In Fermi, each core schedules the instruction from a *warp* of 32 *threads* while AMD's current design schedules the instruction from a *wavefront* of 64 *workitems*.

Because of the way these architectures share and partition certain resources, the SIMD grouping of warps is not explicitly exposed to the programmer [21]. Instead, the programmer groups threads into thread blocks, also known as CTAs, which consist of multiple warps. The number of threads in a CTA is an application parameter, and a CTA typically consists of enough threads for multiple warps. The core may arbitrarily interleave instructions from multiple warps and multiple CTAs. The GPU execution model and ISA do not guarantee ordering between threads unless an explicit barrier synchronization is applied.

2.1 Divergent Control Flow

While the underlying GPU hardware uses SIMD execution, the execution model allows each thread to follow its own logical control flow. The basic mechanism that enables independent branching in each thread, while maintaining SIMD execution, is hardware generated *active masks* (predicates) that indicate which threads are active and which threads are not. Inactive threads still execute the operation because of the SIMD control but the results of the computation are masked and discarded. The warp scheduler maintains a single warp-wide program counter (PC) and tracks the logical PC of each thread. When issuing an instruction, hardware masks out the operations from threads that are not at the current warp-wide PC. There are two common ways to maintaining the logical PC of each thread. The first, used by the GPU in Intel's Sandy Bridge [7, 16], maintains a separate PC for each thread and masks out threads that do not match the current per warp PC. NVIDIA and AMD GPUs use an alternate mechanism in which the active masks are stored on a *reconvergence stack*, which we explain below.

A divergent branch partially serializes execution, as the true and false paths must be executed one after another with those threads on the non-active path being masked out. Each divergence reduces the *SIMD utilization*, which we define as the average number of active threads issued when there is at least one ready warp in a core, as more and more threads are masked. To maintain performance, it is important to minimize serialization, which requires the hardware to identify those points that the true and false paths reconverge. Unless these points are identified, a divergent branch will result in serialized execution from that point until the threads complete. One option for identifying the reconvergence point of a branch is to compute it at compile time, which corresponds to its *immediate post dominator* [9, 28] in the control flow graph. Thus, this mechanism for divergent control flow is known as the post-dominator algorithm or PDOM. The reconvergence stack [9, 30] is the hardware structure used to track both the logical PCs of the threads as they diverge from one another, as well as the points of reconvergence. In our baseline architecture, a separate re-

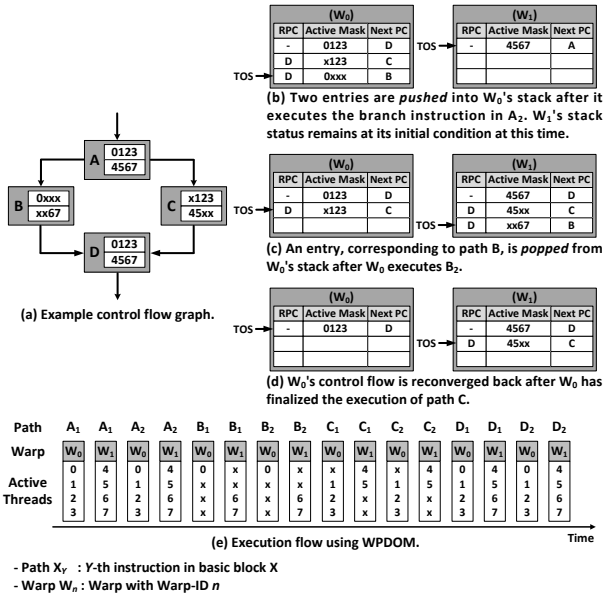


Figure 1: Example showing how WPDOM is used to handle branch divergence. Each CTA has 2 warps of 4 threads each. Each basic block consists of two instructions and a single cycle is required to consume an instruction. The numbers inside the *Active Mask* field represents the thread-IDs that are executing in that path and an *x* denotes a thread that is masked (we use numbers for clarity, while hardware uses single bits).

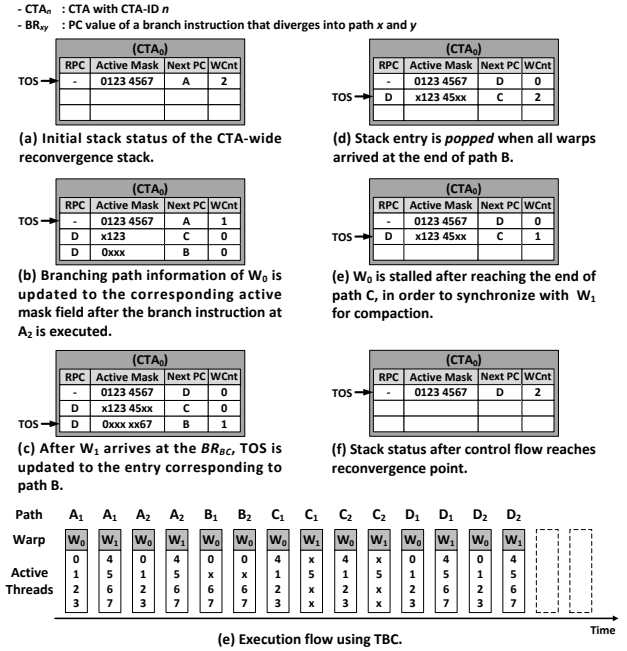


Figure 2: High-level operation of TBC. We assume the same control flow graph and assumptions from Figure 1.

convergence stack is maintained per warp and we thus refer to this technique as warp-wide PDOM (WPDOM) [9]. We explain its operation below using the example shown in Figure 1.

When a warp is to be executed, the entry at the top of its stack (TOS) is initialized to a full mask and the *Next PC* field is set to the first instruction's PC of its basic block (path A in Figure 1(a)). Note that each warp's actual PC is managed by the warp scheduler and that the next PC field in the stack is utilized to track the PC value of the reconvergence points. Each time a warp executes a divergent conditional branch, new active masks are computed for both the true and false paths and then the stack is updated as illustrated in Figure 1(b). First, the next PC field of the current TOS is updated to point to the PC of the reconvergence point, which is the next instruction to be executed with the current active mask once warps reconverge. Then, the mask, next PC, and reconvergence PC (RPC) information of the false path is pushed onto the stack, followed by the information for the true path, and TOS is updated to point to the true path. Thus, after a branch, hardware first executes instruction from the true path and masks out threads from the false path. Note that if the warp did not diverge there is no need to change the reconvergence stack because all threads in the warp can continue executing with the same PC.

When the threads at the true path consumes all the instructions in that path and accordingly the warp's PC matches the RPC at the TOS, this entry is popped off of the stack. At this point, hardware starts executing the false path of the last divergent branch (Figure 1(c)). When the warp's PC again equals the RPC on its corresponding entry, the stack is popped again, threads are reconverged, and non-divergent execution resumes (Figure 1(d)). The algorithm described above elegantly handles nested divergent branches. If another conditional branch is executed while execution is diverged, a new pair of false and true masks and PCs are pushed onto the stack. These new masks are logically AND-ed with the current mask at the top of the stack, so that the top of the reconvergence stack correctly represents the current warp to be scheduled. Each divergence, however, reduces the number of active threads and increases the number of wasted execution slots because a larger fraction of operations is masked. Several mechanisms have been proposed [8, 9, 19] to alleviate the degree of waste and we discuss the most recent state-of-the-art proposals below.

2.2 Thread Block Compaction

The reason that the WPDOM technique described above suffers from low SIMD utilization is that each time a warp diverges, the true and false path must be serialized because the entire warp can only execute a single instruction at a time. The basic idea of improving utilization is to consider multiple warps, up to the entire CTA, as a single unit when a branch diverges. Instead of allowing a warp to be serialized, new warps can be dynamically formed to minimize masked execution slots [8, 9, 19]. Among these, Thread Block Compaction (TBC) [8] is a low cost and elegant mechanism to dynamically reform warps. The basic idea of improving utilization in TBC is to consider the entire CTA as a single unit when a branch diverges. TBC considers the active mask of the entire CTA and compacts it, when possible, into a smaller number of warps with fewer masked threads. This is done by changing the reconvergence stack architecture from one stack per warp to a single stack for the entire CTA (Figure 2). This single stack is used to determine the point when the active masks of the true and false path are fully known in order to apply maximal compaction. TBC, in ef-

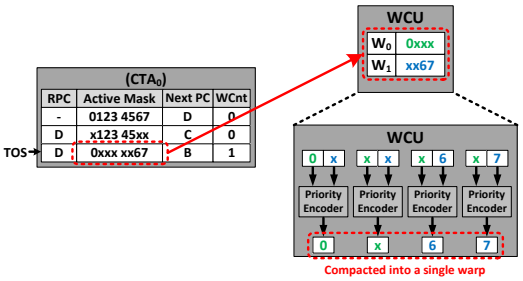


Figure 3: WCU compaction example.

fect, uses its stack structure to introduce a *barrier* after each branch so that the compaction unit has access to the entire active mask of the CTA. This hardware-induced barrier is meant to enable performance gains but is not needed for correctness. A similar approach was suggested by Narasiman et al. [19], except that the granularity for compaction is a fixed *long warp*, which is decoupled from the CTA size, and unconditional branches do not stall and do not wait for the entire CTA to reach the jump point. In Section 5, we compare CAPRI with TBC, as well as an optimized version of TBC (TBC+) that does not stall on either unconditional branches (as in [19]) or on conditional branches that are guaranteed not to diverge. Such non-divergent branches are identified by the CUDA compiler (marked with a `.uni` qualifier in PTX [22]).

Although a single active mask is maintained per CTA, warps are still scheduled independently by the warp scheduler based on available resources and operands. Unlike WPDOM, however, the single entry for the entire CTA implies that all active threads across all warps are all executing instructions from within a single basic block. All active (unmasked) threads execute the same sequence of instructions until they reach another branch, at which point they must wait for all warps to reach the branch to allow another compaction, or until they reach a reconvergence point. To achieve this, hardware needs to know when all active warps have reached a branch or are ready to reconverge. Both branching and reconvergence can only occur after all active warps have synchronized.

Accordingly, in addition to the next PC, active mask, and reconvergence PC, each entry of TBC's stack also includes a counter for determining the number of active warps in the current control flow path (WCnt). When the TOS is changed, the new TOS entry's WCnt is initialized to the number of compacted warps associated with the entry (Figure 2(c),(d),(f)), which is equivalent to the minimum number of warps that are executing along that basic block. Each time a warp executes a branch or reaches a reconvergence point, WCnt of the TOS is decremented. When WCnt reaches zero, all active warps have either reached the end of the basic block and executed the branch instruction or reached the reconvergence point. The first time a branch diverges, new entries are added to the stack for the false and true paths (Figure 2(b)). Note that the TOS is not changed because the entire CTA advances through the control flow graph in unison. The true and false active masks for the warp that just executed the branch are then added to the new entries. Each additional warp that executes the branch incrementally updates its masks as well. When WCnt becomes zero, the next basic block can execute, at which point the TOS is updated to point to the true path of the branch and the CTA-wide active mask is ready for compaction (Figure 2(c)).

Compaction is performed by the *warp compaction unit* (WCU), which is shown in Figure 3. The WCU receives the CTA-wide active mask, when WCnt becomes zero. The WCU then uses a set of priority encoders to identify the minimum number of warps required to execute the active mask while maintaining the fixed association of each thread to its SIMD lane. This is necessary to avoid high-overhead changes to the register file. The output of the WCU is a new set of active masks and the number of warps needed to execute them. This information is then used for the new TOS entry to continue execution. The context information of how the original threads are associated with the newly compacted warps, the thread ID mapping, and the individual warp PC values are stored and maintained by the warp scheduler are as detailed by Fung and Aamodt [8].

When the WCU successfully compacts the CTA-wide active mask into fewer warps than would have executed with WPDOM, performance is likely to improve as illustrated in Figure 2. Although TBC introduces more synchronization points and forces all warps within a CTA to execute in the same basic block, when there are warps available for scheduling from its own CTA or from other CTAs, the synchronization overhead can be hidden effectively. As we show in Section 5, however, in many cases compaction does not result in reducing the number of warps and parallelism is often limited. In other words, previous compaction mechanisms are often an overkill as control divergence most commonly occur in a *non-compactable* manner, especially for workloads that rarely experience divergence. Yet, because they provide no means to differentiate compaction-ineffective branches, the hardware logic units associated with compaction (such as the WCU) are always activated for all branching points – which in turn consume power unnecessarily for compaction-ineffective branches. In the following section, we discuss how our proposed CAPRI microarchitecture alleviates this problem.

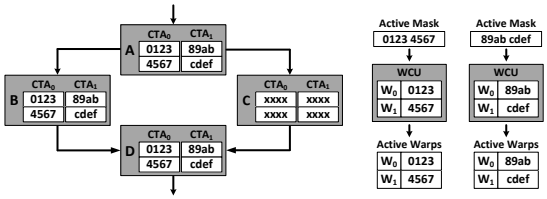
3 CAPRI

This work is motivated by the fact that not all branch instructions are likely to benefit from compaction because conditional branches often follow patterns that are repeated between warps. For example, a common software optimization is to construct kernels such that all threads in any given warp branch in the same way, which effectively eliminates branch divergence, even though the conditional branch is still dynamically determined. A simple example is a loop-end branch, which is conditional but is most typically evaluated to be the same across all threads (line 17 of Figure 4(a) and Figure 4(b)). In other cases, conditionals are not well optimized or some threads are intentionally masked to reduce memory traffic. In such cases, a divergent branch may diverge in a similar way across all warps, which renders compaction ineffective. An example of a conditional branch that diverges but is compaction-ineffective is shown in line

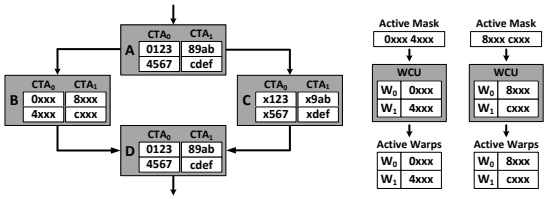
Example 1 Code that contains two potentially divergent branches.

```
0    /* - Excerpted from __global__ void bpnn_layerforward_CUDA( ) kernel of BACKP.
1       - float input_cuda[ ] designates a global memory region */
2
3    int   by   = blockIdx.y;
4    int   tx   = threadIdx.x;
5    int   ty   = threadIdx.y;
6
7    __shared__  float   input_node[HEIGHT];
8
9    if ( tx == 0 )    // Conditional branch that is divergent but compaction-ineffective
10      input_node[ty] = input_cuda[index_in];
11
12   __syncthreads();
13
14   …
15   for ( i = 1; i < __log2f(HEIGHT) ; i++ ){
16         …
17                        // Loop-end branch: Conditional branch that is non-divergent
18   }
```

(a) Examples of conditional branches that are either non-divergent or divergent but compaction-ineffective. *BACKP* is part of the Rodinia benchmark suite [6].



(b) Control flow graph of the loop-end branch on line 17 (non-divergent).

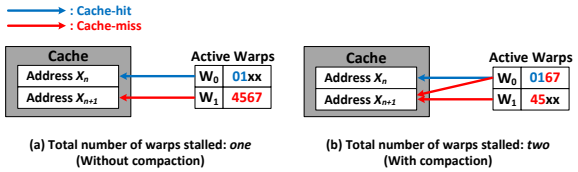


(c) Control flow graph of the divergent yet compaction-ineffective branch on line 9.

Figure 4: Example source code and control flow graphs of compaction-ineffective branches.

9 of Figure 4(a) and in Figure 4(c). Several of the benchmarks we discuss in Section 5 (LPS, BACKP, RAY, FFT, and QSRDM) exhibit a significant number of branches that cause divergence in a regular pattern across warps and as a result cannot be compacted; the number of warps before and after compaction is the same.

As we discuss in Section 5, attempting an ineffective compaction can potentially reduce performance, especially when a kernel contains a significant number branches that are mostly compaction-ineffective. In order to collect candidates for compaction, previous mechanisms stall all threads in the CTA until the last thread reaches the branch point. At that time, compaction occurs and the newly compacted warps can be scheduled. This compaction-induced barrier introduces synchronization overhead, which cannot always be hidden. While the benefits of compaction usually outweigh the overhead of synchronization when successful, that is not the case when compaction is ineffective. When unsuccessful, compaction merely result in shuffling the threads between warps which can potentially cause memory divergence (Figure 5) with no benefits in SIMD utilization and worsening power efficiency through needlessly activated compaction units. Our goal is to overcome this



(a) Total number of warps stalled: *one* (Without compaction)

(b) Total number of warps stalled: *two* (With compaction)

Figure 5: Example of a non-compactable branch that leads to memory divergence upon compaction.

inefficiency, which we demonstrate can occur quite often (Section 5), by stalling only those warps that have a high likelihood of benefiting from compaction. Warps that are not likely to gain from compaction are *bypassed* so that they can continue execution beyond the branch while maintaining its static warp structure.

We achieve this goal by utilizing a compaction-adequacy predictor (CAPRI), which uses the active mask of a warp, just after a branch point, and its adequacy history to predict whether a warp should wait for compaction or continue to execute, ignoring potential compaction opportunities. The adequacy history is the history of successes or failures of compaction with respect to a particular branch. To predict compaction-adequacy, CAPRI follows a design similar to a simple single-level branch predictor. CAPRI uses a prediction table that tracks adequacy history using prediction bits, which are updated based on a dynamically computed compaction-adequacy of conditional branches. Compaction-adequacy is computed, regardless of whether compaction was applied to the warps or not, because warps that did not stall for compaction could have benefited from compaction.

CAPRI consists of three main components: the *compaction-adequacy prediction table* (CAPT) that tracks adequacy history, the *decision logic* that determines whether to delay a warp for compaction, and the *WCU* which includes logic for determining the compaction-effectiveness of a branch. We use Figure 6 as an example to illustrate how TBC and CAPRI execute the control flow graph in Figure 6(a), which contains a non-divergent branch and a branch that is divergent but compaction-ineffective. Notice that with TBC, each branch instruction introduces a barrier, which in the example leads to three idle cycles because the barrier restricts the parallelism available to hide memory latency (Figure 6(b)). With CAPRI, on the other hand, warps that encounter branches that are compaction-inadequate do not wait and no barrier is introduced, effectively reducing the number of idle cycles to one. In general, CAPRI enables greater flexibility in warp scheduling and better latency hiding, which improves performance (Figure 6(c)). We will refer back to this example when discussing CAPRI's components and operation below.

3.1 CAPT

The CAPT is a fully-associative (potentially set-associative) tagged structure used to track adequacy history for branches (Figure 7). Each CAPT entry consists of a tag that identifies a particular branch using the PC of the branch instruction (BADDR), one or more adequacy history bits, and a valid bit. As we show later, the maximum number of CAPT entries necessary in all the benchmarks we eval-

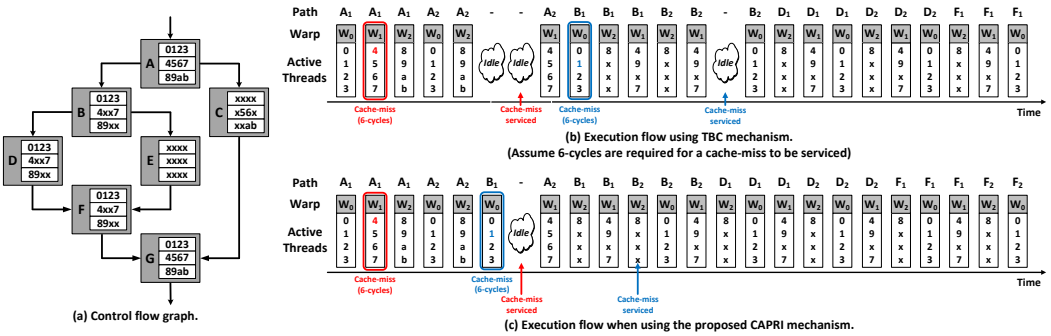


Figure 6: (a) Example control flow graph and its corresponding execution flow when (b) TBC or (c) CAPRI is used. We use the same assumptions from Figure 1 except that each CTA consists of 12 threads.

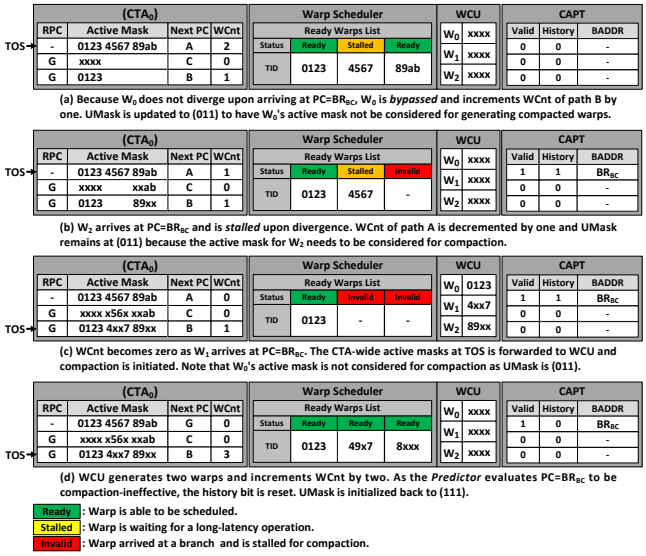


Figure 7: Example CAPRI execution (1b-Latest) of the control flow graph of Figure 6.

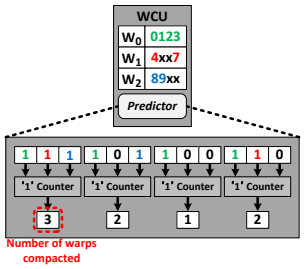


Figure 8: Evaluating the compaction-adequacy of the branch entering block B from Figure 6(a).

uated in Section 5 is 24, and in practice, a 8-entry CAPT was capable of achieving 97% of the benefits provided by an infinite CAPT among the benchmarks that were sensitive to the number of entries. Note that because of its small size, we maintain a separate CAPT for each shader core. We experiment with several configurations of the history bits of each CAPT entry: a 2-bit saturating counter, a single bit indicating the last seen compaction-adequacy behavior, and a single sticky bit that is set once a warp diverged upon a branch instruction.

3.2 Decision Logic

There are three possible outcomes when a warp executes a branch. The first is that the warp did not diverge and there is no benefit to stalling. Therefore, CAPRI's policy is to skip checking the CAPT and simply allow a non-divergent warp to continue executing. This scenario is shown in Figure 7(a). Even without prediction, the fact that non-divergent warps are not stalled unnecessarily already provides an advantage over previously proposed compaction schemes, but is not the main contribution of this study.

The second possible outcome is that the warp diverged and that the CAPT has no information about the divergent branch (Figure 7(b)). In this case, the branch is conservatively assumed to be an adequate candidate for compaction. The CAPT is updated to include this branch and the history is initialized to an *"adequate"* state. At this point, the CAPT has information about the branch, which indicates that future warps should wait for compaction as well.

The third possible outcome is that the warp diverged and the corresponding branch is already registered in the CAPT. In such a case, the history bits of the CAPT are used to decide whether to stall the warp or to bypass compaction (Figure 7(c)). Accordingly, all divergent warps within the CTA will follow the same CAPRI decision because the CAPT is only updated once the WCU evaluates compaction effectiveness and because history is maintained per branch as discussed below. Note that, as a design optimization, warps that have bypassed compaction are not permitted to execute more than a single basic block away from one another. This restriction significantly simplifies the reconvergence stack and WCU and has minimal impact on performance because it is rare for warps along the same control flow branch to have very different execution characteristics.

3.3 WCU, CAPT, and Reconvergence Stack Management

Regardless of whether the warps have been stalled or not, in order to evaluate the compaction effectiveness of a branch, the CTA-wide active mask at the TOS is always forwarded to the WCU when WCnt is zero. This is because warps that were predicted as compaction inadequate and did not wait for compaction could have, in fact, benefited from compaction. We follow the WCU design described in [8] and only make one minor change: warps that did not wait for compaction are marked as *bypassed* using a single bit per warp (collectively referred to as the update-

Table 1: GPGPU-Sim configuration.

Number of shader cores	30
Threads per core	1024
Threads per warp	32
SIMD pipeline width	32
Registers per core	16384
Shared memory per core	32KB
L1 Cache (size/assoc/line)	32KB/8-way/64B
L2 Cache (size/assoc/line)	1024KB/64-way/64B
Number of memory channels	8
Memory controller	Out-of-order (FR-FCFS)
DRAM request queue size	32

mask (UMask) in Figure 7). The WCU does not consider the active masks of the bypassed warps when compacting the warps, but the logic unit that derives the compaction effectiveness (*Predictor* in Figure 8) still processes the corresponding active masks. An example of this is shown in Figure 7(c) to (d) and Figure 8.

The predictor, which evaluates compaction-adequacy inside the WCU, is simpler than the logic used to form compacted warps – it just counts the number of warps that the WCU would have output had all warps waited for compaction. Figure 8 shows this predictor logic that counts the number of active threads associated with each SIMD lane. The maximum number of threads is equal to the minimum number of required warps. If this minimum is equal to the number of active warps, no benefit is provided even though the warps were stalled. In such a case, this branch would not be adequately compacted and we update the CAPT to reflect that. If, on the other hand, the number of post-compaction warps is smaller, the CAPT is updated to indicate adequacy so that the warps would wait for compaction in future iterations.

We experimented with multiple history bit configurations: 2-bit saturating counter, 1-bit latest adequacy result, and 1-bit sticky adequacy. Figure 7 shows how the 1-bit latest history configuration (1b-Latest, 1bL) is updated to reflect the most recent adequacy result from the WCU. The 2-bit counters work in a similar manner, incrementing and decrementing the counter when the WCU evaluates a branch to have been effectively and ineffectively compacted, respectively. The 1-bit sticky configuration (1b-Sticky, 1bS) is the most conservative scheme. It sets the history bit to adequate when the warp that executed a branch diverges and the bit remains set until the kernel completes.

4 Methodology

We model CAPRI in GPGPU-Sim (version 2.1.1b) [4], which is a publicly available, detailed cycle-based simulator of a "general purpose" GPU that supports CUDA version 2.3 and PTX version 1.4. We also implemented TBC as described in [8] and TBC+ – a version of TBC that does not stall warps from branches that cannot diverge (as indicated by the compiler). We configure the simulator to closely match NVIDIA's Quadro FX5800 as recommended in the GPGPU-Sim manual (see Table 1). We adopt the sticky round-robin warp scheduling policy, in which the warps in the currently running CTA maintain the highest scheduling priority until all of that CTA's warps are stalled. We use a 32-entry CAPT with a 1-bit latest history configuration for our default CAPRI configuration. We explicitly note when using different parameters to evaluate sensitivity.

Table 2: Benchmark properties.

Divergent			
Name	**Description**	**Instr.**	**WPDOM SIMD$_{util}$**
BFS	Breadth First Search	17M	29%
MUM	MUMmerGPU	75M	23%
MUM++	MUMmerGPU++	139M	28%
LPS	3D Laplace Solver	81M	74%
BACKP	Back Propagation	193M	83%
TPACF	Two Pt. Ang. Cor. Fun.	293B	81%
Non-Divergent			
Name	**Description**	**Instr.**	**WPDOM SIMD$_{util}$**
RAY	Ray Tracing	259M	91%
LIB	LIBOR Monte Carlo	1B	100%
PFLT	Particlefilter	4B	99%
CFD	CFD Solver	21.4B	96%
FFT	Fast Fourier Transform	250M	91%
QSRDM	Quasirandom Generator	729M	95%

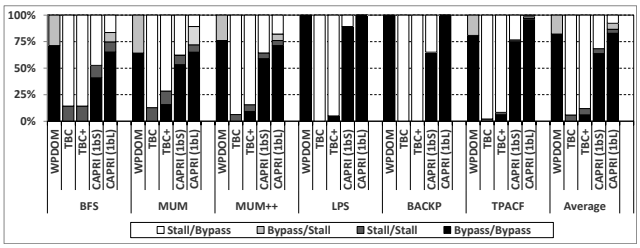
We use a collection of benchmarks from the Parboil [14] and Rodinia [6] suites, CUDA SDK [23], and other 3rd party applications provided with GPGPU-Sim [4, 10, 11, 12, 13, 25]. We ran all applications that are supported by GPGPU-Sim and that we could simulate to completion. Due to space constraints, we present the results of the 12 most representative workloads that display distinct differences across compaction schemes (see Table 2). We use *SIMD utilization rate* (SIMD$_{util}$ in Table 2) as an architecture-level metric for divergence. *SIMD utilization rate* is the average number of threads issued when any warp is ready to execute, divided by the number of threads per warp. We do not present results for the other 13 benchmarks we evaluated. Among these, CAPRI consistently outperformed TBC and TBC+ by $1-3\%$ for benchmarks that experience control divergence, and matched the performance of WPDOM to within $\pm 1\%$ for workloads with little or no divergence.

5 Experimental Results

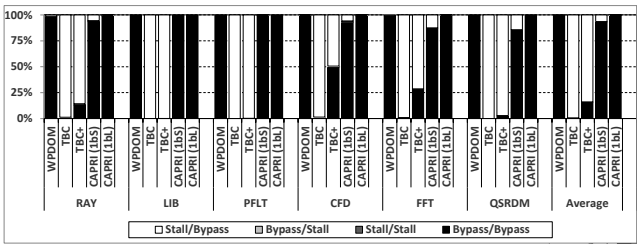
In this section we detail our evaluation of CAPRI, including the quality of the predictions, its impact on SIMD utilization and idle cycles, its impact on performance, parameter sensitivity, and implementation cost.

5.1 Prediction Quality

CAPRI predicts whether the warps at a given branch point should stall and synchronize to attempt compaction, or whether they should continue to execute without compaction. CAPRI can thus correctly or incorrectly predict to stall or to bypass. Figure 9 shows the quality of the predictions by categorizing each per-warp prediction as correctly predicting to bypass and not wait for compaction (Bypass/Bypass), correctly predicting to stall and wait for compaction (Stall/Stall), incorrectly predicting to bypass while compaction would have been effective (Bypass/Stall), and



(a) Divergent benchmarks



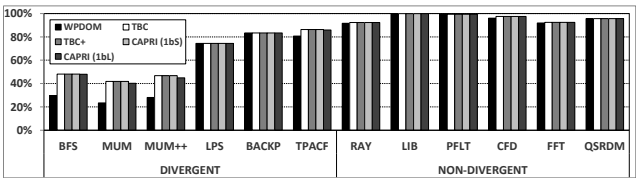
(b) Non-divergent benchmarks

Figure 9: Prediction and stall/bypass decision quality. Each bar represents the fraction of warps that were correctly bypassed or stalled (Bypass/Bypass or Stalled/Stalled), and incorrectly bypassed or stalled (Bypass/Stall or Stall/Bypass).

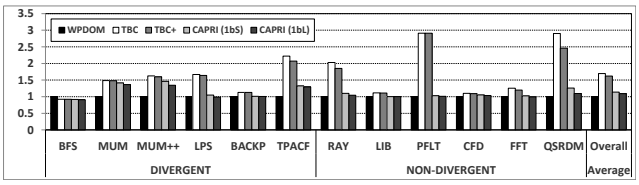
incorrectly predicting to wait when compaction is ineffective (Stall/Bypass). We determine which category each prediction falls into in the following way. For each branch, we perform an exhaustive search of all potential compactions and identify the minimum number of warps required to achieve maximal compaction for each dynamic branch. We then count how many warps were bypassed and how many waited at each specific dynamic branch when using the predictor.

WPDOM and TBC "predict" that all warps should always bypass or always stall, respectively. TBC, for example, only generates correct Stall/Stall or, incorrect, Stall/Bypass decisions. Because most branches are not adequate candidates for compaction (recall examples in Section 3), most decisions made by TBC are to incorrectly stall a warp when it should have just continued to execute without waiting for compaction. TBC makes the smallest fraction of correct decisions between all the schemes. In fact, TBC has near-zero accuracy with non-divergent workloads, which explains why it degrades performance in some cases. WPDOM makes the opposite decisions to TBC, either correct Bypass/Bypass or incorrect Bypass/Stall. WPDOM makes near-perfect decisions for non-divergent workloads. For the divergent benchmarks, WPDOM still makes a significant number of correct decisions as the majority of branches are non-compactable. It is interesting to observe that WPDOM has near perfect accuracy in LPS, BACKP, and TPACF, while TBC and TBC+ make nearly no correct predictions. These three benchmarks have significant branch divergence, but compaction is ineffective (see also Section 5.2).

Unlike TBC, TBC+ decides to bypass warps at branch points when it is statically known that no divergence can occur. There is only a small number of these branches, however, and over 80% of the decisions made by TBC+ are still incorrect. CAPRI, on the other hand, makes high-quality decisions in nearly all cases. CAPRI accurately



(a) SIMD utilization rate.



(b) Normalized idle cycles accumulated across all cores.

Figure 10: SIMD utilization rate and idle cycle count.

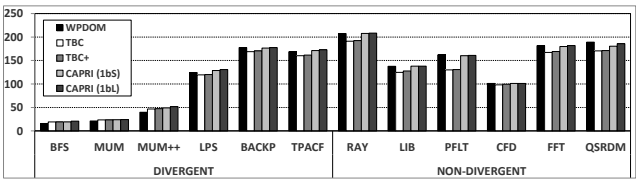
predicts 99.8% and 86.6% of the warps ideal behavior on non-divergent and divergent benchmarks, respectively. The BFS, MUM, and MUM++ benchmarks have highly irregular divergence patterns that are hard to predict. CAPRI still achieves an average of 60.2% and 74.3% accuracy for these three workloads using the 1b-Sticky and 1b-Latest history bit configurations, respectively. As expected, 1b-Sticky has overall lower accuracy, but makes more correct Stall/Stall predictions because of its bias, compared to 1b-Latest that may incorrectly react to anomalous dynamic behavior. Performance with 1b-Latest is higher, indicating that predictions should not be biased towards stalling. The correct decisions made in these three benchmarks result in highly-effective compaction (see Section 5.2). As a result, the number of dynamic warps is much smaller after compaction than in the baseline WPDOM, which is why the fraction of stall decisions is relatively low even though WPDOM has a large fraction of incorrect Bypass/Stall decisions.

In addition to the configurations discussed above, we also experimented with history configurations using 2-bit counters. Accuracy and performance improved marginally, by less than 1%, and we do not include the results for brevity.

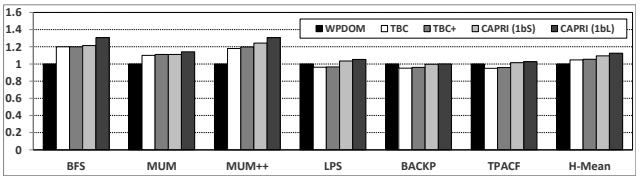
5.2 Utilization and Idle Cycles

Figure 10(a) and 10(b) show the impact of compaction on the SIMD utilization rate and the number of idle cycles. Note that overall execution time is essentially $(SIMD_{utilization} \cdot N_{cores} \cdot N_{lanes} + Idle)$ and ideally, utilization is maximized while idle cycles are minimized. Overall, TBC and TBC+ significantly improve SIMD utilization, by up to a factor of 2 for BFS, MUM, MUM++, and TPACF. CAPRI is able to correctly predict beneficial compaction and matches the SIMD utilization improvements of TBC. The 1b-Sticky history stalls a larger number of threads but achieves marginally better compaction than with the 1b-Latest configuration.

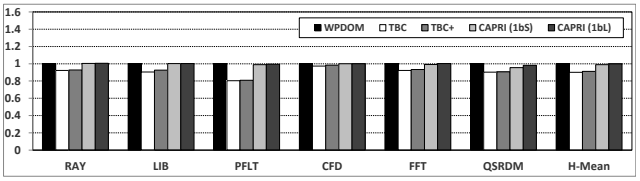
TBC does not achieve significant improvements for all other benchmarks ($< 2\%$), even for LPS and BACKP that have significant divergence. This indicates that compaction is ineffective for the majority of branches and will not improve performance – meanwhile, stalling warps and initi-



(a) Absolute IPC.



(b) Normalized IPC for a set of divergent benchmarks.



(c) Normalized IPC for a set of non-divergent benchmarks.

Figure 11: Performance of CAPRI compared to WPDOM, TBC, and TBC+.

ating compaction upon the few divergent yet compaction-ineffective branches may degrade performance as explained in Section 3. On average, TBC increases idle cycles by 86.4% with 7 applications experiencing an increase greater than 40%. By avoiding stalls on unconditional branches, TBC+ introduces fewer idle cycle, but still has 62.2% more idle cycles than the baseline WPDOM on average. While CAPRI matches the gains of TBC, it does much better with respect to idle cycles. By not stalling most warps that do not benefit from compaction, CAPRI does not increase idle cycles by more than 50% for *any* application and incurs an average of only 13.7% and 9.1% increase for the 1b-Sticky and 1b-Latest history configurations, respectively. Based on these results we expect 1b-Latest to outperform all other schemes because it is nearly optimal in terms of compaction gains and has the smallest increase of idle cycles.

5.3 Overall Performance Results

Figure 11 shows the performance of CAPRI relative to that of WPDOM, TBC, and TBC+. We show absolute IPC across the entire processor, as well as normalized IPC with separate harmonic means for the divergent and non-divergent benchmarks. CAPRI outperforms TBC and TBC+ on all benchmarks because it generally correctly distinguishes between warps that can benefit from compaction and those that only suffer unnecessary synchronization delays. Concerning divergent workloads, CAPRI with 1b-Latest outperforms WPDOM and TBC+ by 12.6% and 7.2% on average (harmonic mean), respectively. As expected, the 1b-Sticky configuration does not perform as well as 1b-Latest, but still outperforms WPDOM and TBC+. Note that TBC and TBC+ degrade the performance of the divergent LPS, BACKP, and TPACF applications, compared to WPDOM, because of the ineffective compaction and excessive synchronization overheads while gaining none or little in SIMD utilization. On non-divergent benchmarks, TBC and TBC+ suffer an average of 10.1% and 8.9% performance degradation, respectively. CAPRI correctly predicts that there is no opportunity for compaction in the non-divergent workloads and matches the performance of WPDOM to within 1%.

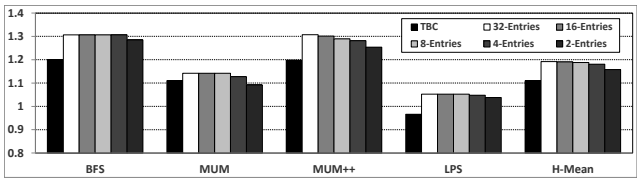
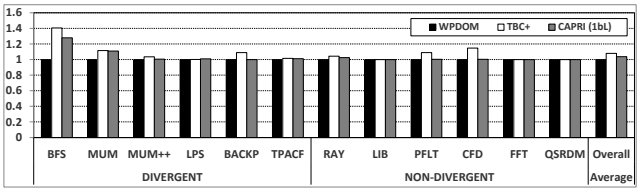
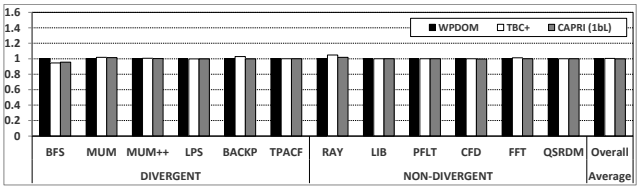


Figure 12: Normalized performance of CAPRI (1b-Latest) with variable CAPT size.



(a) Normalized L1 misses.



(b) Normalized L2 misses.

Figure 13: Changes in L1 and L2 miss count from compaction, normalized to WPDOM.

5.4 Sensitivity Analysis

Figure 12 shows the normalized IPC achieved when varying the number of CAPT entries on a subset of the divergent benchmarks. We show those benchmarks for which the number of entries has impact. All other benchmarks achieved at least 98.3% of the IPC of an unlimited CAPT with just 4 entries. MUM and MUM++ are the most most sensitive to the number of entries used. Even these benchmarks achieve over 94% of the IPC benefits of an unbounded CAPT with just 4 entries and 82.3% with a 2-entry CAPT. All results use LRU replacement and a fully-associative organization.

Compacting warps involves rearranging threads from the CTA. Because different groups of threads execute concurrently compared to program order and the baseline WPDOM, it is possible that the application memory behavior changes as well. Figure 13 shows the relative number of first- and second-level cache misses for TBC+ and CAPRI normalized to WPDOM. TBC+ significantly increases the number of L1 misses for many applications, including some applications in the non-divergent class. The reason for the latter is that the WCU rearranges threads even when compaction is ineffective (number of warps is not reduced), which can increase miss rate (Figure 5). CAPRI, on the

other hand, does much better than TBC+. CAPRI correctly predicts the lack of compaction-effectiveness of many branches, including nearly all in the non-divergent workloads. As a result, misses do not increase for these workloads and are reduced (compared to TBC+) for the highly-divergent applications. Note that for BFS and MUM, which suffer the greatest increase in L1 misses, performance is actually improved the most because of the large benefits from compaction. Even though L1 miss rate increases, there is little impact on actual L2 misses and memory traffic. This is because of the filtering effect of L1, which increases the likelihood that a cache line would still be resident and service the reordered accesses without increasing the L2 miss count.

5.5 Implementation Overhead

Fung and Aamodt [8] evaluated the area overhead of TBC and showed it to be less than $1mm^2$ for an entire chip in $65nm$ technology. In addition to this small area, CAPRI requires area for the CAPT and for adequacy evaluation in the WCU. As shown above, an 8-entry CAPT is sufficient to attain near-maximal performance improvement. Each CAPT entry consists of a 32-bit tag, 1 valid bit, and 1 history bit, for a total of 272 bits for an entire shader core. This structure can be accessed in parallel while the branch is executing, which enables a simple and low-power implementation of its 8-way associativity.

The adequacy evaluator (predictor) added to the WCU can be implemented using simple logic for counting the number of ones in each SIMD lane, which represent an active thread in that lane. This logic is significantly simpler than the compaction logic itself and should add negligible area to the WCU.

In terms of power overhead, because the CAPT is queried once per warp for a branch and is significantly smaller than the GPU register file (128KB of registers per shader core in Fermi [20]) for example, we expect the overall power consumption of the CAPRI microarchitecture will not be burdensome – note that the predictor and compaction logic operate at the granularity of warps or even thread-blocks.

6 Related Work

We discussed the most relevant works in Section 2 and briefly summarize additional related work below. The idea of masked execution dates back to very early SIMD and vector machines, including the Illiac IV [5] and the Cray-1 [24]. The idea of masking was extended to software optimizations along with the idea of exchanging control flow for data flow with the concept of *predicated execution* [1]. In the context of vectors, Smith et al. [27] proposed *density-time execution*, which is similar to compaction but operates in time on traditional masked vector execution. Smith et al. also provide a good summary of performance comparisons among various vector ISAs that incorporate conditional operations. An idea similar to the above was developed for stream processors by Kapasi et al. [17].

Tarjan et al. [29] proposed *adaptive slip* to address memory divergence issues. Adaptive slip enables a subset of a warp to continue executing while other threads are waiting on memory. This work was extended to *dynamic warp subdivision* [18], which allows warp subsets to be scheduled independently to enhance latency tolerance. Diamos et al. [7] propose *Thread Frontier* as an alternative to the immediate post dominator reconvergence algorithm. Thread frontiers use the earliest reconvergence point possible in an unstructured control flow [31]. These other GPU mechanisms are orthogonal to CAPRI and can be applied on top of CAPRI for further optimization.

7 Conclusions

In this paper we argue that previously proposed mechanisms to mitigate the negative impact of control divergence fall short because they introduce excessive synchronization, even when compaction provides no benefit. We show that a large fraction of conditional branches cannot benefit from compaction because they happen not to diverge much or because their divergence pattern is not amenable to compaction. To overcome this fundamental deficiency of decreased performance from unnecessary synchronization, we describe and evaluate a dynamic hardware predictor that predicts whether a branch point is likely to adequately benefit from compaction. When the prediction is positive, all divergent warps that execute the branch will stall and wait for compaction. If the adequacy prediction is negative, compaction is bypassed and synchronization is avoided.

Prior compaction techniques provide benefit for highly-divergent applications but degrade performance in some cases (by up to 19.6%). The CAPRI predictor-based compaction mechanism provides superior performance improvements when improvements are possible. At the same time, by correctly identifying the lack of compaction-adequacy, CAPRI matches the performance of the baseline WPDOM to within $\pm 1\%$. Furthermore, we also show that CAPRI's impact on memory behavior is much smaller than that of prior compaction mechanisms. In conclusion, with very small area overhead, CAPRI is able to improve performance by up to average 7.6% (max 10.8%) on top of TBC on divergent workloads and avoid TBC's 10.1% average (max 19.6%) performance degradation in non-divergent cases.

Acknowledgements

We would like to express our gratitude to the anonymous reviewers who provided excellent feedback and insight for preparing the final version of this paper.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[2] AMD Corporation. AMD Radeon HD 6900M Series Specifications, 2010.

[3] AMD Corporation. ATI Stream Computing OpenCL Programming Guide, August 2010.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2009)*, April 2009.

[5] W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick. The Illiac IV System. In *Proceedings of the IEEE*, volume 60, pages 369–388, April 1972.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC-2009)*, October 2009.

[7] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD Re-Convergence At Thread Frontiers. In *44th International Symposium on Microarchitecture (MICRO-44)*, December 2011.

[8] W. W. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *17th International Symposium on High Performance Computer Architecture (HPCA-17)*, February 2011.

[9] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th International Symposium on Microarchitecture (MICRO-40)*, December 2007.

[10] A. Gharaibeh and M. Ripeanu. Size Matters: Space/-Time Tradeoffs to Improve GPGPU Applications Performance. In *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC-2010)*, November 2010.

[11] M. Giles. Jacobi iteration for a Laplace discretisation on a 3D structured grid, 2008.

[12] M. Giles and S. Xiaoke. Notes on using the NVIDIA 8800 GTX graphics card. http://people.maths.ox.ac.uk/ gilesm/hpc/, 2008.

[13] P. Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing HiPC 2007*, volume 4873, pages 197–208. 2007.

[14] IMPACT Research Group. The Parboil Benchmark Suite, 2007.

[15] Intel Corporation. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency, May 2009.

[16] Intel Corporation. Intel HD Graphics OpenSource Programmer Reference Manual, June 2011.

[17] U. Kapasi, W. Dally, S. Rixner, P. Mattson, J. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In *33th International Symposium on Microarchitecture (MICRO-33)*, December 2000.

[18] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *37th International Symposium on Computer Architecture (ISCA-37)*, 2010.

[19] V. Narasiman, C. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *44th International Symposium on Microarchitecture (MICRO-44)*, December 2011.

[20] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[21] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2011.

[22] NVIDIA Corporation. PTX: Parallel Thread Execution ISA Version 2.3, 2011.

[23] NVIDIA Corporation. CUDA C/C++ SDK CODE Samples, 2011.

[24] R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21:63–72, January 1978.

[25] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.

[26] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[27] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. In *27th International Symposium on Computer Architecture (ISCA-27)*, 2000.

[28] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[29] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for SIMD cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC-09)*, 2009.

[30] S. Woop, J. Schmittler, and P. Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24:434–444, July 2005.

[31] H. Wu, G. Diamos, S. Li, and S. Yalamanchili. Characterization and Transformation of Unstructured Control Flow in GPU Applications. In *1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, June 2011.