2018 IEEE International Symposium on High Performance Computer Architecture

Warp Scheduling for Fine-Grained Synchronization

Ahmed ElTantawy[§†], Tor M. Aamodt[§]
[§]*University of British Columbia, Canada*
[†]*Huawei, Canada*
ahmede@ece.ubc.ca, aamodt@ece.ubc.ca

***Abstract*—Fine-grained synchronization is employed in many parallel algorithms and is often implemented using busy-wait synchronization (e.g., spin locks). However, busy-wait synchronization incurs significant overheads and existing CPU solutions do not readily translate to single-instruction, multiple-thread (SIMT) graphics processor unit (GPU) architectures. In this paper, we propose Back-Off Warp Spinning (BOWS), a hardware warp scheduling policy that extends existing warp scheduling policies to temporarily deprioritize warps executing busy wait code. In addition, we propose Dynamic Detection of Spinning (DDOS), a novel hardware mechanism for accurately and efficiently detecting busy-wait synchronization on GPUs. On a set of GPU kernels employing busy-wait synchronization, DDOS identifies all busy-wait loops incurring no false detections. BOWS improves performance by 1.5× and reduces energy consumption by 1.6× versus Criticality-Aware Warp Acceleration (CAWA) [14].**

***Keywords*-Warp Scheduling; Synchronization; SIMT Architectures; Busy Wait; GPGPUs**

I. Introduction

There is growing interest in running applications with fine-grained synchronization on GPUs as indicated by recently announced changes to control-flow handling on NVIDIA's next generation Volta GPU [26], [25]. Enabling faster synchronization on GPUs is important even if CPUs always have an advantage because it helps reduce the need to transfer data between CPU and GPU between processing steps in larger scientific application work flows. Demand for faster synchronization support on SIMT architectures is evident in that the performance of atomic operations has improved by orders of magnitude in recent NVIDIA GPU generations (Section II). To enable support for fine-grained synchronization, and not unlike recent academic proposals [11], [10], NVIDIA's Volta removes the stack-based SIMT reconvergence mechanism found in prior NVIDIA (and AMD) GPUs [26], [25]. This change will enable easier programming on GPUs [10] for workloads such as Graph analysis [6], [22], [20], [34], data flow algorithms [16], hash tables [13], and others. However, NVIDIA cautions "threads spinning on a lock may degrade the performance of the thread holding the lock" [26], [25]. In this paper, we focus on this challenge.

Overheads of fine-grained synchronization have been well studied in the context of multi-core CPU architectures [37], [33], [17], [9]. However, the scale of multi-threading and the fundamental differences in the architecture in SIMT machines hinders the direct applicability of the previously proposed CPU solutions (more details in Section VII). In SIMT machines, barrier synchronization overheads have been recently studied [19], [18]. These studies proposed warp scheduling policy that accelerates warps that have not yet reached a barrier to enable other warps blocked at the barrier to proceed. However, bust-wait synchronization is a fundamentally different problem. With barriers, warps that reach a barrier are blocked and do not consume issue slots. With busy-wait synchronization, threads that fail to acquire a lock spin, compete for issue slots and, in the absence of coherent L1 caches, compete for memory bandwidth.

Yilmazer and Kaeli [36] quantified the overheads of spin-locks on GPUs and proposed a hardware-based blocking synchronization mechanism called hierarchal queue locking (HQL). HQL provides locks at a cache line granularity by adding flags and pointer meta-data to each L1 and L2 block, which can be in one of six states. Negative acknowledgments are used when queues are filled and in certain race conditions. An acquire_init primitive is added to the application to set up a queue. While HQL achieves impressive performance gains when an application uses a small number of locks relative to threads, it can experience a slowdown when using a large number of locks concurrently. Moreover, HQL adds a significant area to the caches and requires a fairly complex cache protocol. While Yilmazer and Kaeli noted the potential for synchronization aware warp scheduling to help improve HQL, no details of how to implement such a scheduler were described. By judiciously modifying warp scheduling, this paper shows how to effectively approximate the benefits of queue-based locking without the complexity and overhead of directly implementing queues.

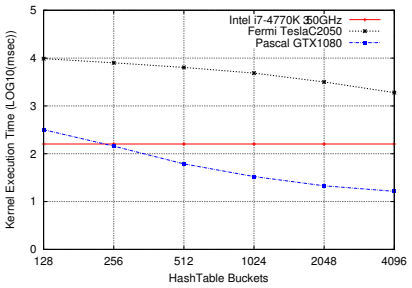
Criticality-Aware Warp Acceleration (CAWA) [14] uses run-time information to predict *critical warps*. Critical warps are those that are slowest in a kernel and as they determine execution time CAWA prioritizes them. CAWA estimates warp criticality using a criticality metric that predicts which warp will take the longest time to finish. CAWA outperforms greedy-than-oldest (GTO) warp scheduling across a range of traditional GPGPU workloads [14]. However, CAWA can reduce performance for busy-wait synchronization code as its criticality predictor tends to prioritize spinning warps.

We propose Back-Off Warp Spinning (BOWS), a schedul-

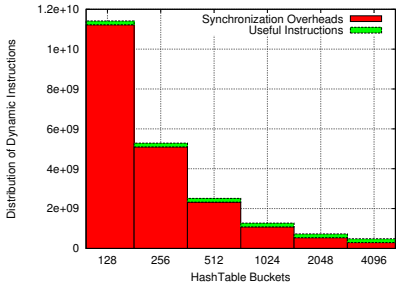
2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00040

```
1.  bool done = false;
2.  int *mutex = lock[hashValue].mutex;
3.  while(!done){
4.      if(atomicCAS(mutex, 0, 1) == 0 ){
5.          __threadfence();
6.          location->next = table.entries[hashValue];
7.          table.entries[hashValue] = location;
8.          done = true;
9.          __threadfence();
10.         atomicExch(mutex,0);
11.     }//else back-off delay code (Section4, Figure4a)
12. }
```

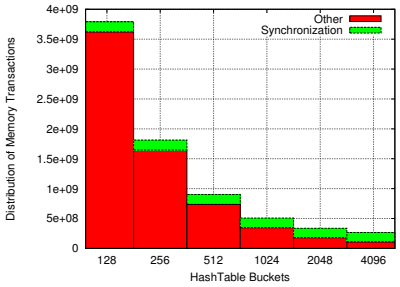
(a) Critical Section in Hashtable Insertion



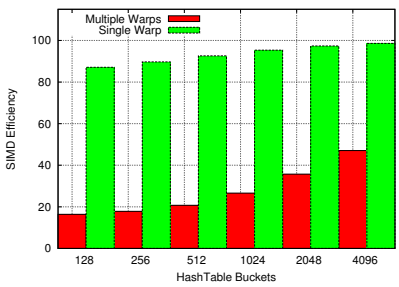
(b) Hash Table Perf. (y-axis is log scale)



(c) Dynamic Instructions Overheads



(d) Memory Traffic Overheads



(e) Divergence Overheads

Figure 1: Fine-grained Synchronization in current GPGPUs. Both CPU and GPU versions are compiled with NVCC-6.5 -O3. Overheads measured on Pascal GTX1080 using nvprof with 120 blocks each containing 256 threads.

ing policy that prevents spinning warps from competing for scheduler issue slots. BOWS approximates software back-off techniques used in multi-threaded CPU architectures [2], which are impractical to apply directly to GPUs (Figure 3). Warp prioritization in stack-based SIMT architectures is complicated by the fact that some threads within a warp may hold a lock while others do not. In BOWS warps that are about to execute a busy-wait iteration are removed from competition for scheduler issue slots until no other warps are ready to be scheduled. BOWS requires annotation of acquire and release operations. Thus, we propose a hardware mechanism, called Dynamic Detection of Spinning (DDOS), to *detect* spin locks in GPGPU code. DDOS tracks path and selective register value histories for a subset of threads within a kernel to accurately identify spin loop code. DDOS provides the first approach to achieve spin detection similar to CPU proposals [17] that is practical for GPU implementation.

II. Background and Motivation

In this section, we analyze the overheads of busy-wait synchronization on GPUs. Similar to Yilmazer and Kaeli [36] we find high overheads. In contrast to their analysis, our measurements in Figure 1 use real GPU hardware instead of simulation. Moreover, we compare against CPU-only execution, breakout instruction overheads due to spinning, explore the impact of hardware warp scheduling and software only back-off techniques.

Figure 1a shows how atomics can be used to implement locks in architectures using a SIMT-stack. This code is modified from NVIDIA's CUDA by Example [30] and implements a critical section in a hashtable[1]. Figure 1b compares the execution time of 26.2 million insertions of random keys to this hashtable on a GPU versus on a CPU while varying the number of hashtable buckets (y-axis is log scale). Few buckets mean more contention. We compare two NVIDIA GPUs and a CPU: A Tesla C2050 (Fermi), GeForce GTX 1080 (Pascal), and an Intel Core i7 running a serial CPU implementation. The GTX 1080 outperforms the single-threaded CPU implementation, that uses the same algorithm, for reasonably sized hashtables: At 4096 buckets the GTX 1080 is 9.77× faster.

Figure 1c shows synchronization overhead ranges from 61.0% up to 98.3% of dynamic instructions at high contention. Similarly, Figure 1d shows 41.5% to 95.6% of memory operations are due to synchronization. A significant portion of both is due to failed lock acquire attempts. Another source of synchronization overhead on GPUs is control-flow divergence. Figure 1e shows that if the code is executed by a single warp, the SIMD utilization (fraction of active lanes) ranges between 87.1%-98.6% but drops to 16.4%-47.1% when executing multiple warps due to inter-warp lock conflicts.

Next, we consider the impact of warp scheduling policies.

[1]We modified the code to avoid serializing threads within a warp as proposed in [30]. This improves performance on our hardware

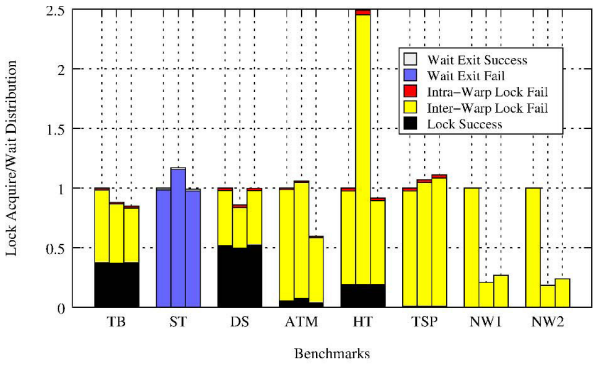


Figure 2: Synchronization Status Distribution. Bars from left to right: LRR, GTO, and CAWA. GPGPU-Sim with a GTX480 configuration. See Section V for details.

Greedy then Oldest (GTO) scheduling [29] selects the same warp for scheduling until it stalls then moves to the oldest ready warp. Older warps are those with lower thread IDs. GTO typically outperforms Loose Round Robin (LRR) [29]. In CAWA (Section I) warp criticality is estimated as: $nInst \times w.CPI_{avg} + nStall$, where $nInst$ is an estimate of remaining dynamic instruction count (based on direction of branch outcomes), $w.CPI_{avg}$ is per-warp CPI, and $nStall$ is the stall cycles experienced by a warp. Critical warps are prioritized.

Figure 2 plots the distribution of lock acquire attempts in lock-based synchronization and the wait exit attempts in wait and signal based synchronization (benchmarks and methodology described in Section V) using LRR, GTO, and CAWA scheduling policies.The figure also shows the distribution of whether the lock acquire failure is because the lock is held by a thread within the same warp (i.e., intra-warp lock fail) or in a different warp (i.e., inter-warp lock fail). Most lock failures are due to inter-warp synchronization. The figure shows that inter-warp conflicts are significantly influenced by the warp scheduling policy.

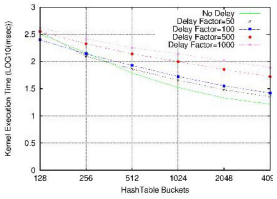
Figure 3 plots execution time of the hashtable insertion code in Figure 1a augmented with the software-only backoff delay code in Figure 3a running on GTX 1080 hardware. The results suggest that adding a backoff delay to a spin-lock degrades performance on recent GPUs. The reason is that, except at very high levels of contention, the benefits of reduced memory traffic appear insufficient to make up for wasted issue slots executing the delay code itself.

III. BOWS: Backoff Warp Spinning

To avoid wasted issue slots we propose Back-Off Warp Spinning (BOWS), a hardware scheduling mechanism that reduces the priority of spinning warps. BOWS assumes synchronization loops have been identified by programmer, compiler or DDOS (Section IV).

```
1. clock_t start = clock();
2. clock_t now;
3. for(;;) {
4.  now = clock();
5.  clock_t cycles = now>start? now-start :
      now +(0xffffffff-start);
6.  if(cycles >= DELAY_FACTOR*blockIdx.x){
7.    break;
8.  }
9. }
```

(a) Backoff Delay Code*.



(b) Execution Time.

Figure 3: Software Backoff Delay Performance in GPUs. *omp_set_lock GPU implementation for OpenMP 4.0 [4].

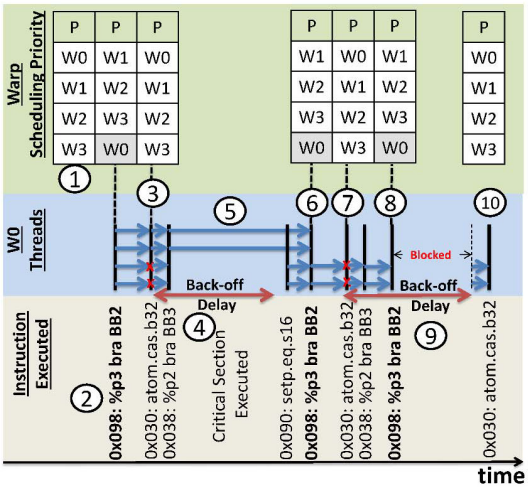


Figure 4: BOWS scheduling Policy.

A. BOWS scheduling policy

The scheduling policies examined in Section II suffer from two limitations:

- The scheduler may prioritize spinning warps in the competition for issue slots over other eligible non-spinning ones. This slows down the progress of non-spinning warps. In cases when these non-spinning warps are holding locks, this decision also slows down the forward progress of spinning warps.
- The scheduler may return back to the same spinning warp too early even if it was at the bottom of the scheduling priority because other warps are stalling on data dependencies.

BOWS avoids these issues by modifying an existing warp scheduling policy as follows:

- It discourages warps from attempting another spin iteration by inserting the warp that is about to execute another iteration into the back of the warp scheduling priority queue. Warps in this state are called Backed-off. Once a warp in the backed-off state issues its next instruction its priority reverts to normal and leaves the backed-off state.
- It sets a minimum time interval between the start of

any two consecutive iterations of a spin loop by the same warp. Warps that are about to start a new spin loop iteration prior to the end of their interval are not eligible for scheduling.

BOWS requires that Spin-Inducing Branches (SIBs) have been identified. SIBs are the backward branch of each spin loop. Once a warp executes a SIB, the scheduler control unit triggers BOWS' logic.

1) BOWS Operation: BOWS works as follows: Once a warp exits its backed-off state, a *pending back-off delay* register is initialized to the back-off delay limit. The warp then continues execution normally with the pending back-off delay register decremented every cycle. If the warp executes a SIB it *cannot* issue its next instruction until its back-off delay is zero. The back-off delay value can be determined through profiling or tuned adaptively at runtime.

Figure 4 shows an example of BOWS operation for warp W0 containing four threads for the code in Figure 7a. Backward branch `0x098: %p3 bra BB2;` has been identified as a SIB. Scheduling priority is shown in the top of Figure 4. Initially, W0 has high priority ①. Once W0 encounters a spin-inducing branch ②, it is pushed to the back of the priority queue and marked as backed-off (shaded in Figure 4). W0 is scheduled when other warps are stalling (e.g., on memory accesses for line 6 in Figure 1a) and executes the lock-acquire atomic compare and swap instruction (Figure 7a, PC=0x030). At this point, ③ three actions are taken: First, the warp loses its backed-off state; second, the warp priority reverts to normal; and third a back-off delay value is stored in the warp pending back-off delay register ④. Two threads of W0 successfully acquire the lock and proceed to the critical section while the other two threads fail ⑤. Threads reconverge at the `setp` instruction and execute the spin-inducing branch. The two threads that executed the critical section exit the spin loop while the others proceed to another iteration. Once the spin-inducing branch is executed, the warp enters the backed-off state and is pushed to the end of the priority queue ⑥. As the duration of the critical section is larger than that of the back-off delay limit W0's back-off delay is already zero and so W0 is eligible for scheduling. After W0 is scheduled it executes the lock acquire and the two remaining threads in the spin loop again fail to acquire a lock ⑦. The two threads immediately proceed to another iteration of the spin-loop ⑧. However, once W0 enters the backed-off state, it cannot be scheduled until the pending back-off delay is zero ⑨. Once the pending back-off delay is zero, W0 is eligible for scheduling ⑩.

2) Adaptive Back-off Delay Limit: A small back-off delay may increase spinning overheads while a large back-off delay may throttle warps more than necessary. We adaptively set the delay by trying to maximize $(\frac{Useful\ Instructions}{Spinning\ Overheads})$ over a window of execution. We use $\frac{Total\ Inst.}{SIB\ Inst.} = \frac{Useful\ Inst. + SIB\ Inst. \times avg.\ Spin\ Overhead}{SIB\ Inst.}$ as a rough estimate. As the average spin overhead is almost constant across the execution of the same kernel the ratio of the $\frac{Total\ Instructions}{SIB\ Instructions}$ is proportional to $\frac{Useful\ Instructions}{Spinning\ Overheads}$.

```
for each Execution Window of T cycles:
  if(SIB Instructions > FRAC1 * Total Instructions)
    Delay Limit += Delay Step
    if((Total Instructions)/(SIB Instructions)<
       FRAC2 * (Prev. Total Instructions)/(Prev. SIB
           Instructions))
         Delay Limit -= 2*Delay Step
    if(Delay Limit > Max Limit) Delay Limit = Max Limit
    if(Delay Limit < Min Limit) Delay Limit = Min Limit
```

Figure 5: Adaptive Back-off Delay Limit Estimation.

The pseudo code in Figure 5 summarizes our adaptive back-off delay limit calculation. This algorithm is applied over successive time windows. During the current window the adaptive back-off delay estimation algorithm computes the back-off delay limit to use during the next window. Initially, the scheme attempts to increase the back-off delay limit by a fixed step as long as a non-negligible ratio of dynamic spin-inducing branches is executed. However, if the ratio of $\frac{Total\ Instructions}{SIB\ Instructions}$ in the current execution window is considerably smaller than the ratio in the previous window the back-off delay limit is decremented by a double step. Finally, lower and upper limits are applied to the back-off delay limit. The values used in evaluation are listed in Table II.

IV. Dynamic Detection of Spinning

It is possible to identify spin loops when explicit busy-wait synchronization APIs are used. The compiler can then translate a lock acquire API into a busy wait loop with the backward branch of the loop flagged as a spin inducing branch. However, such APIs are not available in current SIMT programming models. Moreover, they may are challenging to implement due to the *SIMT-induced deadlock* problem [10].

A SIMT-induced deadlock occurs when a thread is indefinitely blocked due to the cyclic dependency between SIMT scheduling constraints and a synchronization operation. For example, consider the use of a while loop to implement a lock acquire statement such as follows:

```
while(atomicCAS(mutex,0,1)!=0);
//critical section
atomicExch(mutex,0);
```

This code deadlocks on most GPUs. Instead of progressing in the critical section towards the lock release statement, thread(s) that successfully acquire a lock are blocked at the while loop exit waiting to reconverge with thread(s) in the same warp that are trying to acquire a lock held by one of the successful threads. Thus a cyclic-dependency occurs leading to a deadlock. The code in Figure 1a, addresses this problem using a standard GPU programming approach

```
1.bool transaction_done = false;
2.while(!transaction_done) {
   //try lock 1
3. if( atomicCAS(&lock1->lock,0,1)==0){
    // try lock 2
4.  if(atomicCAS( &lock2->lock,0,1)==0){
5.    //critical section
6.    atomicExch(&lock2->lock,0); // release lock 2
7.    atomicExch(&lock1->lock,0); // release lock 1
8.    transaction_done = true;
9.  }else {
10.   atomicExch(&lock1->lock,0); // release lock 1
11.  }
12. }
13.}
```

(a) Nested Locks (ATM [12] and CP [12], [5]).

```
1.for(i = 0; i < 32; i++) {
  // serialize threads within the same warp
2. if(lane_id==i) {
     // try global lock
3.   while(atomicCAS(mutex,0,1)!=0){
4.   }
5.   //critical section
6.   atomicExch(mutex,0);
7.  }
8.}
```

(b) Global Locking (TSP [28], [30]).

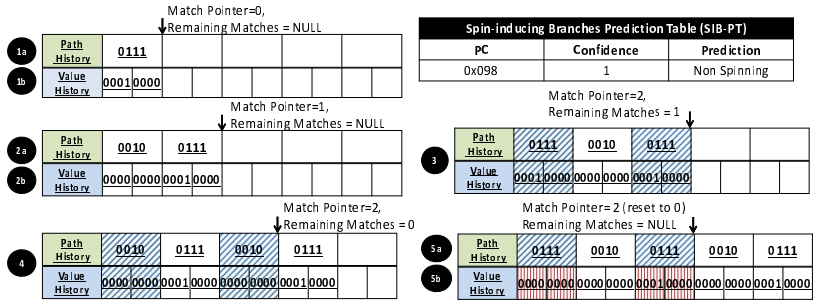
```
1.....
2.while (k >= bottom) {
3. start=startd[k];
4. if (start >= 0) { // if not wait
5.  ....
6.  if (ch >= nbodiesd) {
7.    ...
8.  }else {
9.    // child is a body
10.  sortd[start]=ch; //signal
11. }
12.  k-=dec; // move to next cell
13.}
14.}
```

(c) Wait and Signal (BH-ST [6]).

Figure 6: Examples of Inter-Thread Synchronization Patterns used in GPUs (Section V).

```
0x028: mov.s16 %r21, 0;
       BB2:
0x030: atom.cas.b32 %r15, [%rl29], 0, 1;
0x038: setp.eq.s32 %p2, %r15, 0;
0x040: @%p2 bra BB3;
0x048: bra.uni BB4;
       BB3:
       // critical section
0x088  mov.s16 %r21, 1
       BB4:
0x090: setp.eq.s16 %p3, %r21, 0;
0x098: @%p3 bra BB2;
```

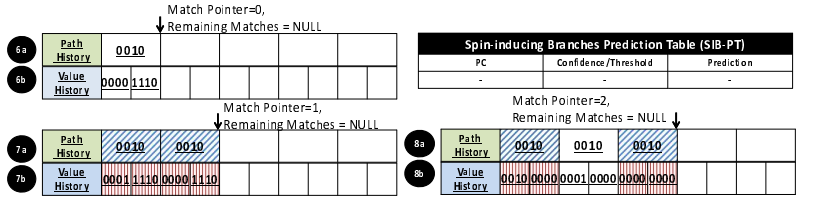
(a) Busy-Wait Loop.



(b) Updates to History Registers and SIB-PT.

```
0x020: ld.param.u32 %r15,
     [_Z14invert_mappingPfS_ii_param_3];
0x028: mov.u32 %r20, 0;
       BB2:
0x030: ld.global.f32 %f1, [%rl14];
0x038: st.global.f32 [%rl15], %f1;
0x040: add.s64 %rl15, %rl15, %rl4;
0x048: add.s64 %rl14, %rl14, 4;
0x050: add.s32 %r20, %r20, 1;
0x058: setp.lt.s32 %p4, %r20, %r15;
0x060: @%p4 bra BB2;
```

(c) Regular Loop.



(d) Updates to History Registers and SIB-PT.

Figure 7: Warp History Registers and SIB-PT Operation (Figure 8 shows the units locations in the pipeline).

involving placing the lock release inside the spin loop. In this code threads that successfully acquire the lock are guaranteed to be able to make forward progress to the lock release code. However, to generalize this approach to arbitrary synchronization patterns, it requires non-trivial and costly code transformations. Please refer to [10], [25] for details.

Consequently, current GPU programmers write synchronization code tailored to their specific application scenario to avoid deadlocks. Figure 6a shows an implementation of two nested locks that avoid SIMT-induced deadlocks from ATM and Figure 6b shows an implementation of a global lock from TSP where the execution of the critical section is serialized across threads from the same warp. Figure 6c shows busy-wait synchronization from the ST kernel in BH that implements a wait and signal synchronization rather than a lock. A thread waits in a spin loop for a condition set by another thread. The large variety of synchronization patterns makes it challenging to detect busy-wait synchronization statically [10]. Thus, below, we describe a mechanism for dynamically detecting SIBs.

To identify a SIB, DDOS first makes a prediction regarding whether each warp is currently in a spinning state or not. As noted by Ti et al. [17], a thread is spinning between two dynamic instances of an instruction if it executes the instruction and later executes the same instruction again (e.g., in another loop iteration) without causing an observable change to the net system state (i.e., to its local registers or to memory). Ti et al. [17] propose a thread spinning detection mechanism design for multi-threaded CPUs which tracks the changes in all local registers by all threads. This information is used to drive OS thread

scheduling decisions and/or frequency scaling. However, with thousands of hardware threads in GPGPUs and the energy cost of register file accesses, it is not practical to employ such mechanism in massive multi-threaded SIMT architectures. DDOS essentially approximates Ti et al.'s approach to reduce costs.

DDOS detects busy-wait loops in two steps. First, it detects the presence of a loop. DDOS does this by tracking the sequence of program counter values of a warp. Second, DDOS speculates whether a loop identified in the first step is a busy-wait loop or a normal loop. To distinguish these cases it leverages the observation that typically in normal loops found in GPU code an induction variable changes every iteration. Moreover, this induction variable typically contributes to the computation of the loop exit condition. In NVIDIA GPUs the loop exit condition and the divergence behavior of a thread are typically determined using a set predicate instruction (available both in PTX and SASS)[2]. For each thread in a warp, the set predicate instruction compares two source registers and writes the result to a boolean destination register. The boolean values are typically used to predicate execution of both normal and branch instructions (e.g., instructions at address 0x090 and 0x098 in Figure 7a). In normal (none busy-wait) loops, the value of at least one source register of the set predicate (`setp`) instruction(s) that determine the loop exit condition change each iteration. For example, in a 'for' loop, one of these registers would be the loop counter. DDOS tracks only the values of source registers of the set predicate instructions to determine whether a loop is a normal loop (i.e., `setp` source register values change) or a busy-wait spin loop (`setp` source register values do not change).

A. DDOS Operation

Conceptually, the spin loop detection step of DDOS works as follows: Each warp has two shift registers, a Path History Register and a Value History Register (Figure 7b). These registers track the execution history of the first active thread in the warp. We refer to this thread as the *profiled thread*. The Path History Register tracks program counter values of `setp` instructions. The Value History Register tracks the values of the source registers of `setp` instructions. To reduce storage overhead we hash program counter and source operand values before adding them to the Path History and Value History Registers. As elaborated upon in Section IV-C the Value History Register is implemented in the execution stage. DDOS' examines entries in Path and Value History Registers looking for repetition. If it finds sufficient repetition DDOS classifies the profiled thread as being in a *spinning state*.

[2]AMD Southern Islands ISA has an equivalent vector compare instruction (v_comp) [1].

Figure 7 illustrates operation of Path and Value History Registers on PTX[3] assembly examples with (Figure 7a) or without (Figure 7c) busy wait code. Figure 7a is equivalent to Figure 1a. In Figure 7a assume the first active thread is executing the `setp` instruction at $PC = 0x038$. In the busy-wait example in Figure 7b, the program counter is first hashed using: $((PC - PC_{kernel\ start})/Instruction\ Size)\%m)$[4], where $PC_{kernel\ start} = 0x000$, $m = 4$ and $Instruction\ Size = 8$. The result ($0x7$) is inserted into the Path History Register (1a). In parallel, the source operand values of the `setp` instruction are hashed and added to the Value History Register. We assume the profile thread fails to acquire the lock so that %r15 is '1'. Only the least significant k-bits (here k is 4) are used (1b). To detect repetition DDOS keeps track of two other values, Match Pointer and Remaining Matches. The Match Pointer identifies which m-bit (k-bit) portion of the Path (Value) History Register to compare new insertions against. For each insertion into the path (value) history registers, the entry before the match pointer is compared with the new entry. If they are equal, a loop is detected. To enable better selectivity DDOS requires multiple consecutive loop detections before identifying a spin inducing loop. To facilitate this the *remaining matches* register tracks the number of remaining matches required.

Continuing the example in Figure 7b, eventually the warp executes the `setp` instruction at PC=0x90 in Figure 7a. The entries in both shift registers are (logically) shifted to the right and new values inserted to their left. No match is found between the new entry ($0x2$) and the entry before the match pointer ($0x7$) (2a). As the profiled thread fails to acquire the lock %r21 remains '0'. Thus, the value history register is updated with two 4-bit zero values (2b). When the warp reaches PC=0x038 again we assume the profiled thread again fails to acquire the lock leading to a match in both path and value histories (3). Once a match is detected, the match pointer is fixed and the remaining matches value is initialized to $(matchpointer - 1)$. Once the warp reaches the `setp` instruction at PC=0x090 again an additional match is found (4). Since the remaining matches value is now zero, the warp is identified as in a spinning state. After the profiled thread successfully acquires the lock the execution of the `setp` instruction at PC=0x040 leads to a mismatch in the value history and the warp loses its spinning state (5b).

To detect SIBs DDOS employs a spin-inducing branch prediction table (SIB-PT). The SIB-PT, shown in Figure 7b, is shared between warps executing on the same SM. The SIB-PT maintains a confidence value for each branch under consideration. When a warp is in a spinning state and it executes a backward branch if that branch is not in the SIB-PT then it is added with a confidence value of 1. If the branch

[3]PTX is Nvidia GPU virtual assembly [24].

[4]We discuss other hashing techniques in Section IV-B.

is in the SIB-PT, its confidence value is incremented. Once the confidence reaches a threshold the branch is identified as a spin-inducing branch. To guard against accumulated path and value hash aliasing errors a nonzero confidence is decremented every time the branch is taken by a warp that is currently classified as non-spinning.

Returning to the example in Figure 7b, initially, the SIB-PT is empty. Once the warp executes the backward branch at PC=0x098 while in the spinning state (i.e., after ❹ and before ❺) the branch is added to the SIB-PT with its confidence set to '1'. Assuming a confidence threshold of 4, only three more instances where the backward branch at PC=0x098 is executed by a spinning warp would be required before this branch is confirmed as a SIB. Larger threshold values reduce false predictions but lead to longer detection time.

For example, consider the PTX code in Figure 7c, which is the assembly of a 'for' loop in Kmeans [7], [8]. The backward branch is at 0x060 and its associated `setp` is at 0x058. The first source operand %r20 represents the 'for' loop induction variable that is incremented by one every iteration (at 0x050), while %r15 is a copy of the kernel input indicating the number of loop iterations. The PC of the `setp` instruction is hashed to (0x2) and inserted into the Path History Register every time the instruction is executed (❻a, ❼a, and ❽a). In contrast to the busy-wait case, the contents of %r20 changes each iteration causing a mismatch with every insertion to the value history register (❼b and ❽b).

B. DDOS Design Trade-offs

Next, we evaluate the impact of DDOS parameters on the following metrics: (1) Average True Spin Detection Rate (TSDR), which is the percentage of spin-inducing branches accurately identified by DDOS; (2) Average False Spin Detection Rate (FSDR), which is the percentage of non-spin-inducing branches incorrectly classified as spin-inducing; and (3) Avg. Detection Phase Ratio (DPR), which is the average ratio of the detection phase duration of a branch to the cycles executed from the first encounter to the last encounter of the branch. The detection phase duration of a branch measures how many cycles were required to confirm a branch as a spin-inducing branch after its first encounter. For SIBs it is preferable to have a short detection phase. Table I shows the sensitivity of these metrics to the different design parameters averaged over all our benchmarks (see Section V for details).

Hashing Function: The top sub-table in Table I studies the impact of XOR and MODULO hashing. In XOR hashing, the values inserted into the path register are hashed as follows (PC[m-1:0] xor PC[2m-1:m] xor PC[3m-1:2m] ... xor PC[31:32-m]), where PC is the program counter at the execution of a `setp` instruction. The value register XOR hashes are computed similarly but using the source registers

Sensitivity to the hashing function "h" at t=4 and l=8				
h	Avg. TSDR	Avg. DPR	Avg. FSDR	Avg. DPR
XOR, m=k=4	1	0.041	0.016	0.006
XOR, m=k=8	1	0.041	0	-
MODULO, m=k=4	1	0.041	0.17	0.014
MODULO, m=k=8	1	0.041	0.104	0.001
Sensitivity to the Hashed Path/Value Width "m/k" at t=4, l=8, and h=XOR				
m/k	Avg. TSDR	Avg. DPR	Avg. FSDR	Avg. DPR
2	1	0.042	0.078	0.062
3	0.983	0.074	0.012	0.008
4	1	0.041	0.016	0.006
8	1	0.041	0	0
Sensitivity to Confidence Threshold "t" at m=k=4, l=8, and h=XOR				
t	Avg. TSDR	Avg. DPR	Avg. FSDR	Avg. DPR
2	1	0.03	0.027	0.016
4	1	0.041	0.016	0.006
8	1	0.075	0.002	0.002
12	0.992	0.105	0.002	0.003
Sensitivity to the History Registers Length "'l" at t=4 , m=k=8. and h=XOR				
l	Avg. TSDR	Avg. DPR	Avg. FSDR	Avg. DPR
1	0	0	0	0
2	0	0	0	0
4	0.625	0.032	0	0
8	1	0.041	0	0
Sensitivity to Time Sharing of History Registers "sh" at l=8, t=4, h=XOR, and epoch=1000				
sh	Avg. TSDR	Avg. DPR	Avg. FSDR	Avg. DPR
0, m=k=4	1	0.041	0.016	0.006
0, m=k=8	1	0.041	0	0
1, m=k=4	0.642	0.211	0.033	0.023
1, m=k=8	0.642	0.211	0.026	0.003

Table I: DDOS Sensitivity to Design Parameters.

in the `setp` instructions. In MODULO hashing, values are hashed by considering only the least significant m (k) bits of the value (as in Figure 7). XOR hashing considerably reduces false detections compared to MODULO hashing. With 8-bits hashing width, the XOR hashing has a zero false detection rate. False detections occur in Merge Sort and Heart Wall with MODULO hashing due to loops with power-of-2 induction variable increments larger than 2^k.

Hashing Width: The impact of the hashing width is quantified in the second sub-table in Table I. A 2-bit path and value width leads to aliasing that leads to 7.8% false detection rate. With 3-bits the aliasing impact is smaller and 8-bits are enough to eliminate false detections with XOR hashing.

Confidence Threshold: The third sub-table in Table I shows that as the confidence threshold (t) increases, the false detection rate decreases but the detection phase ratio increases for true detections. With $t = 12$ some SMs fail to confirm a spin-inducing branch (e.g., TB kernel of BH).

Hashing Registers Length: The fourth sub-table in Table I shows the sensitivity to the history length (l), which determines the number of `setp` instructions DDOS can track. DDOS needs at least five entries in its history registers to detect their spin behaviour.

Time Sharing of History Registers: The results of time-sharing a single set of path and value history registers among different warps in an SM is shown in the last sub-table in Table I. Here a warp uses the history registers for a certain fixed interval (1000 cycles), then another warp uses them. Time-sharing reduces detection accuracy and leads to longer detection phase.

In evaluation, we use "h=XOR, t=4, m=k=8, l=8, and time sharing disabled". The total storage per warp for both the path and value history registers is 192-bits. In our benchmarks, the maximum number of confirmed SIBs was

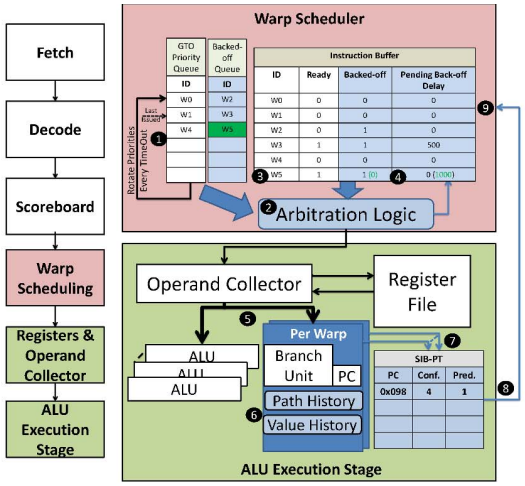


Figure 8: Operation of BOWS with DDOS.

three. However, the maximum number of concurrent entries in the SIB-PTX was 9 entries (the next maximum was only four). A conservative 16-entry SIB-PT requires 560 bits of storage per SM.

C. DDOS integration with BOWS

Figure 8 illustrates BOWS' combined with DDOS. **Warp Scheduling:** BOWS modifies the warp scheduling and execution stages. We found that strict GTO scheduling (without BOWS) can leads to livelocks on two of our benchmarks (HT and ATM). To avoid this, we modify GTO to rotate the age priority periodically (every 50,000 cycles in our evaluation). Arbitration logic first checks whether the last issued warp is ready to issue its next instruction ❶. If the last issued warp is not ready, the oldest ready warp that is not backed-off is selected ❷. If no such warp is available the backed-off queue is checked. A warp is added to the backed-off queue after executing a SIB. A warp in the backed-off queue can be scheduled only if it is both ready and its back-off delay is zero ❸. If the arbitration selects such a warp it is removed from the backed-off queue. The "Backed-off" field for the warp is set to false and the "Pending Back-off Delay" is initialized to the back-off delay limit value when the warp *exits* the backed-off state ❹.

ALU Execution Stage: Path and value history are updated during execution of `setp` instructions ❺,❻. Current GPUs already supports instructions such "shuffle" which allow threads within the same warp to access each other's registers [27]. The underlying hardware can be used to select the source registers of the first active thread. If the warp executes a backward branch, then it looks up the SIB-PT ❼. If the branch is predicted to be a spin-inducing branch the warp enters the backed-off state ❾ and is pushed to the end of the queue.

BOWS Specific Configuration			
Baseline Scheduler	GTO (+rotates age every 50,000 cycles), LRR, CAWA		
Window (T)	1000 Cycles	Delay Step	250 Cycles
Min Limit	1000 Cycles	Maximum Limit	1000 Cycles
FRAC1	0.5	FRAC2	0.8
DDOS Specific Configuration			
Hashing Function	XOR (Default)	History Width (m=k)	8
History Length (l)	8	Confidence Threshold (t)	4
Time Sharing	Disabled		
Baseline Configuration			
Parameter	GTX480 (Fermi)		GTX1080Ti (Pascal)
Number of Cores	15		28
Number of Threads/Core	1536		2048
Number of Registers/Core	32768		65536
L1 Data Cache	16 KB, 128 B line, 4-way LRU		48 KB, 128 B line, 6-way LRU
L2 Data Cache	64 KB/Channel,128 B line, 8-way LRU		128 KB/Channel, 128 B line, 16-way LRU
I$ Cache	4 KB, 128 B line, 4-way LRY		8 KB, 128 B line, 4-way LRU
Number of Warp Schedulers/Core	2		4
Frequencies (Core/Interconnect/L2,Memory)	700,700,700,924 MHz		1481,2962,1481,2750 MHz

Table II: GPGPUSim Configuration

V. Methodology

We implement BOWS in GPGPU-Sim 3.2.2 [3], [31]. We use GPGPU-Sim GTX480 for both GPGPU-Sim and GPUWattch for performance and energy evaluation. In Section VI-D, we report results for a Pascal GTX1080Ti configuration that has a correlation of about 0.85 for Rodinia to estimate the impact of BOWS on the performance of newer generations of GPUs. We evaluate the impact of BOWS on three scheduling policies; GTO, LRR, and CAWA. We use BOWS and DDOS design parameters detailed in Table II.

For evaluation, we use Rodinia 1.0 [7], [8] for synchronization free benchmarks (see Section VI-B). We use and the kernels described below for kernels displaying different synchronization patterns.

BH: BarnesHut is an N-body simulation algorithm [6]. Its Tree Building (**TB**) kernel uses lock-based synchronization [6]. The kernel is optimized to reduce contention by limiting the number of CTAs and using barriers to throttle warps before attempting a lock acquire. Its sort kernel (**ST**) uses a wait and signal synchronization scheme. We run BarnesHut on 30,000 bodies.

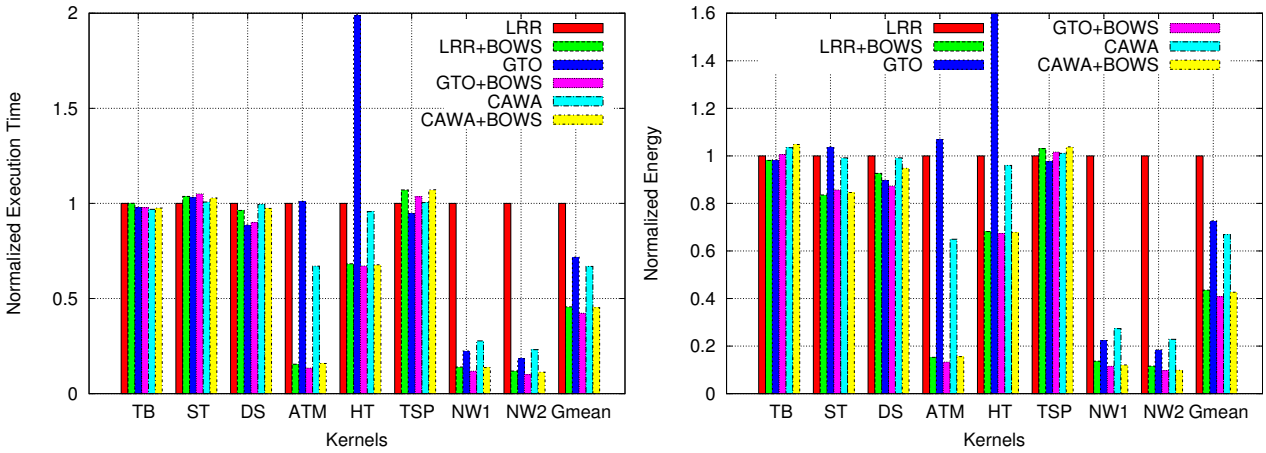
CP: Clothes Physics perform cloth physics simulation for a T–shirt [5]. Its Distance Solver (**DS**) kernel lock-based implementation uses two nested locks to control updates to cloth particles.

HT: Chained HashTable uses the critical section shown in Figure 1a. We run 3.2M insertions by 40K threads on 1024 hashtable buckets.

ATM: An bank transfer between two accounts [12]. It uses two nested locks. We run 122K transactions with 24K threads on 1000 accounts.

NW: Needleman-Wunsch finds the best alignment between protein or nucleotide sequences following a wavefront propagation computational pattern. We implemented the lock-based algorithm in [16] which uses two kernels NW1 and NW2 that perform similar computation while traversing a grid into opposite directions.

TSP: Travelling Salesman. We modified the CUDA implementation from [28] to use a global lock when updating the optimal solution. We run TSP on 76 cities with 3000 climbers.



(a) Normalized Execution Time

(b) Normalized Dynamic Energy

Figure 9: Performance and Energy Savings on GTX480 (Fermi)

VI. Evaluation

Figure 15 shows normalized execution time and energy consumption on busy-wait synchronization kernels. Results are normalized to LRR. BOWS uses adaptive back-off delay. It uses DDOS for detecting spin loops. The design parameters are shown in Table II).

Figure 15 shows that BOWS consistently improves performance over different baseline scheduling policies with a speedup of 2.2×, 1.4×, and 1.5× and energy savings of 2.3×, 1.7×, and 1.6× compared to LRR, GTO, and CAWA respectively.

BOWS has minimal impact on TB because TB's code uses a barrier instruction to limit the number of concurrently executing warps between lock acquisition iterations. We note this barrier approach is fairly specific to TB. For example, it requires at least one thread from each warp to reach the barrier each iteration. Also, the lack of adaptivity of this software-based barrier approach can be harmful even where it can be applied (would lead to a 28x slowdown if applied to HT, measured on hardware - Pascal GTX1080). ST shows 17.8% energy improvements with BOWS (Figure 15b) as it reduces dynamic instruction count but does not exhibit performance improvement because the performance is limited by memory latency. In TSP, the synchronization instructions consume ¡0.03% of the total number of instructions, thus synchronization code is not the dominant factor in execution time. Large back-off delay values may unnecessarily block execution leading to performance degradation (see TSP results in Figure 10).

For the NW kernels, the progress of younger warps is dependent on older warps finishing their execution. Therefore, NW prefers GTO scheduling over LRR as it gives priority to older warps. HT with the GTO scheduler runs into a pathological scheduling pattern where it prioritizes spinning warps which significantly reduce performance. BOWS eliminates such problems by deprioritizing spinning warps.

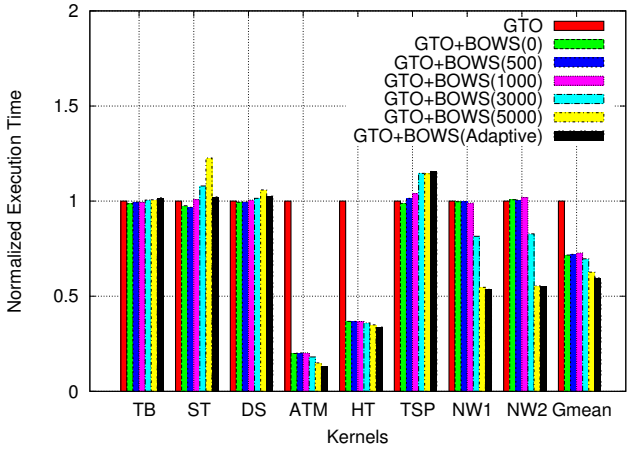


Figure 10: Normalized Execution Time at Different Back-off Delay Limit Values (using DDOS).

A. Sensitivity to Back-off Delay Limit Value

The following results use the GTX480 configuration with GTO as the baseline policy for BOWS. Figure 11 shows the average distribution of warps at the scheduler in terms of their status (backed-off or not). The first bar is GTO. The remaining bars are for BOWS as the back-off delay limit value increases. The last bar to the right is BOWS with adaptive back-off delay limit. The figure shows how BOWS impacts warp scheduling. The back-off delay is not effective until reaching a threshold unique to each benchmark. The reason is that the back-off delay sets a minimum duration between two successive iterations of a spin loop. If warps already consume a time that is larger than the back-off delay limit before they attempt another iteration, then the back-off delay has no observable effect (recall the discussion of Figure 4). The effective back-off delay value depends upon

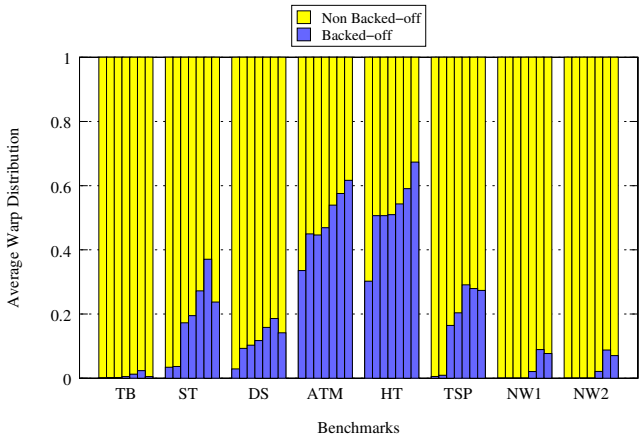


Figure 11: Distribution of Warps at the Scheduler. From left to right, GTO without BOWS, GTO with BOWS with delay limit in cycles 0, 500, 1000, 3000, 5000, Adaptive.

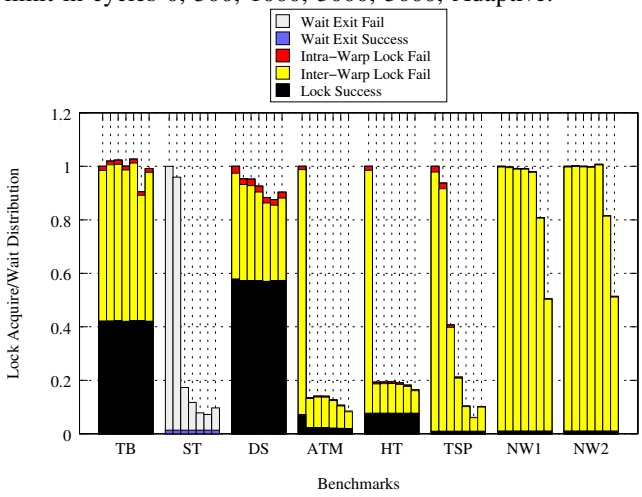


Figure 12: Distribution of Warps at the Scheduler. From left to right, GTO without BOWS, GTO with BOWS with delay limit in cycles 0, 500, 1000, 3000, 5000, Adaptive.

how many instructions are along the failure path in the busy-wait code, how many warps are running and how much memory contention there is.

Figure 12 shows the distribution of Lock acquire and wait status. The behavior aligns with the percentage of warps that are backed-off in Figure 11. This data elucidates performance gaps in some benchmarks – particularly, HT, ATM, and NW – between the different scheduling policies. For example, in HT BOWS reduces the lock failure rate by 10.8× compared to GTO.

Figure 13a shows the impact of BOWS on the dynamic instruction count. On average BOWS reduces dynamic instruction count by a factor of 2.1× compared to GTO. Figure 13b shows that BOWS also reduces the number of L1D memory transaction by 19% compared to GTO. One of the side effects of BOWS is that it increases SIMD efficiency

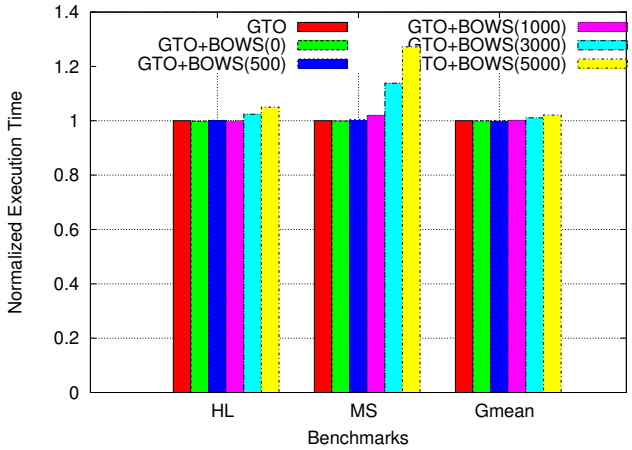


Figure 14: Overheads Due to Detection Errors.

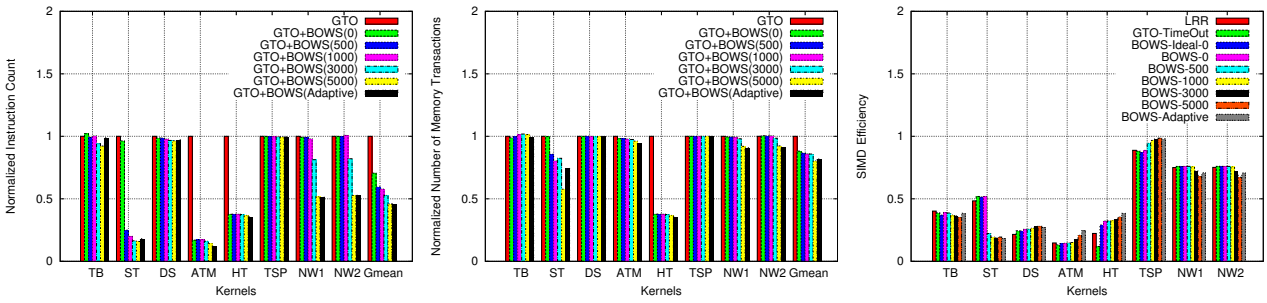
for some benchmarks. For example, BOWS improves HT and ATM SIMD efficiency by 3.4× and 1.85× respectively compared to GTO. In ST, the significant reduction of the number of spin iterations (see Figure 13a) biases the SIMD calculation results as the benchmark spends more time in executing the divergent code rather than spinning, and hence the reduction in SIMD efficiency.

B. Sensitivity to Detection Errors

Note that with the XOR hashing configuration we do not have any false detections. Thus, the results of Synchronization-Free benchmarks are identical to the baseline. Figure 14 reports the results of Synchronization-Free benchmarks under the MODULO hashing. For Synchronization-Free benchmarks, BOWS is expected to perform identically to the baseline under perfect spin detection. Only two applications from Rodinia have false detections with MODULO hashing, Merge Sort (MS) and Heart Wall (HL). In both of these applications, false detections were due to 'for' loops with a large power of two induction variable increment that is not reflected in the least significant 8-bits of `setp` source registers. In this evaluation, we use an 8-bit hash width for the path and value registers. On average, over Rodinia's 14 benchmarks, BOWS with a 5000 cycles back-off delay and MODULO hashing downgrades GTO performance by only 2.1% on these synchronization free applications. However, for MS, BOWS with MODULO has and a large backoff delay downgrades performance versus GTO significantly.

C. Sensitivity to Contention

Figure 16 uses the hashtable benchmark to study BOWS sensitivity to contention. A small number of hashtable buckets indicate higher contention. The figure shows that BOWS provides a speedup of up to 5× at high contention and down to 1.2× at low contention. Similarly, the dynamic instruction count savings ranges from 3.7× to 1.3×. Figure 16b also



(a) Dynamic Instruction Count (b) Number of Memory Transactions. (c) SIMD Efficiency.

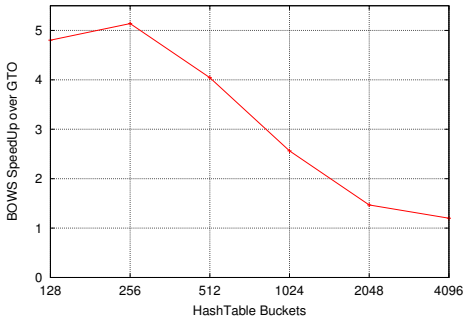
Figure 13: BOWS Impact on Dynamic Overheads.

includes data, "Ideal Block Inst. Count", that serves as a proxy for how HQL [36] might perform on this workload. This curve shows the instruction count assuming locks do not require multiple iterations to acquire. The difference between the two curves thus represents the overhead introduced by BOWS versus an ideal queuing lock system. As we can see, the benefits of an (idealized version of) HQL appear to diminish as the number of hash buckets increases.

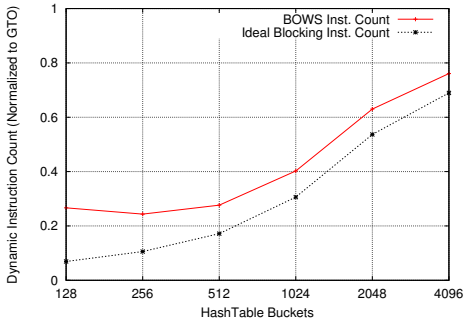
D. Pascal GTX1080Ti Evaluation

To evaluate the impact of BOWS on recent architectures, we configured GPGPU-sim and GPUWattch to model GTX1080Ti (the configurations are currently available in GPGPU-sim GitHub repository). We evaluate the same benchmarks with the same inputs used for GTX480. BOWS consistently improves performance over different baseline scheduling policies with a speedup of 1.9× on LRR, 1.7× on GTO, and 1.5× on CAWA.

One observation is that on Pascal, except for DS, the behavior is flat across the different baseline scheduling policies. The reason is that most of the input data sets for the workloads we run are set to fully utilize (without oversubscribing) the Fermi GPU but they under-utilize Pascal. Pascal has almost double the number of cores compared to Fermi (Table II). Thus, on Fermi, each core has many warps to choose from and the scheduling policy is of a great impact. However, in Pascal, each core will have about half the number of warps distributed on four warp schedulers instead of two. Thus, the number of warps available at each scheduler in Pascal is one fourth that in Fermi making the baseline scheduling policy less important (e.g., unlike the case with Fermi on NW and HT benchmarksZ). DS, on the other hand, is oversubscribed in the Fermi configuration and the number of concurrently running CTAs is limited to four due to the number of available registers per core. This helps to limit contention. However, in Pascal, each core runs up to 8 CTAs/Core and Pascal has more cores. This significantly increases the number of concurrent warps and thus lock contention. Therefore, DS performs worse with Pascal baseline than Fermi. BOWS significantly improves



(a) BOWS SpeedUp.



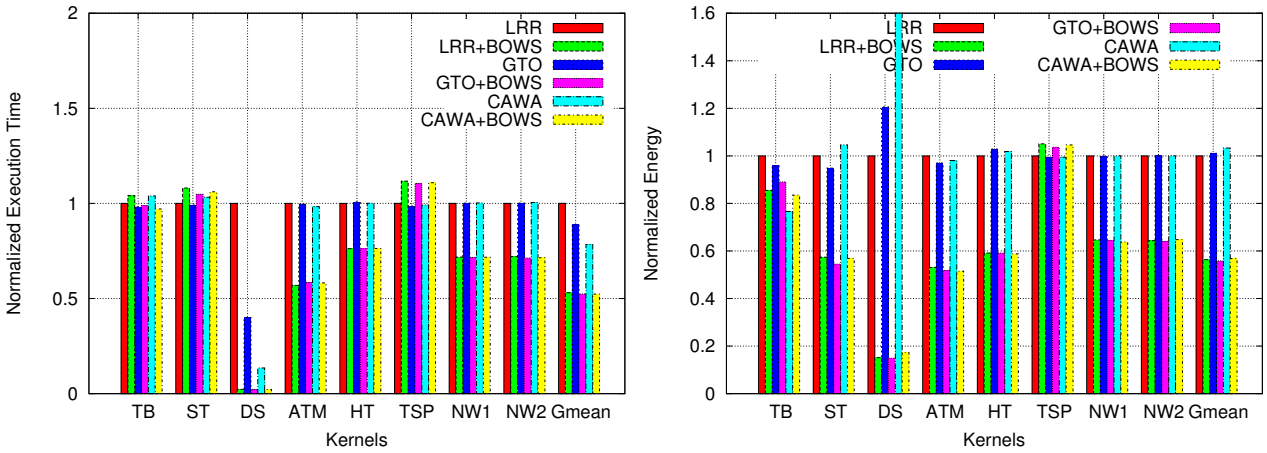
(b) BOWS Inst. Count.

Figure 16: Sensitivity to Contention.

performance as it combines deprioritizing spinning warps (which helps when there are many warps to schedule from) and throttling spinning warps by forcing them to wait by the back-off delay limit (which helps when there are few warps to schedule from).

E. Implementation Cost

Table III identifies the basic components in both DDOS and BOWS and estimates their costs per SM. The main cost of DDOS is the history registers, but using time-sharing (Section IV-B) it may be possible to reduce this cost. Comparison and hashing logic can be shared across warps in the same SM. To enable back-off delay up to



(a) Normalized Execution Time (Pascal)

(b) Normalized Dynamic Energy (Pascal)

Figure 15: Performance and Energy Savings on Pascal

DDOS	SIB-PT	16-entry - 35 bits each (560 bits)
	History Registers	48 warps * 192 bits,(9216 bits)
	Comparison	8-bit comparator + 8:1 8-bitMux
	Hashing (XOR)	8 4-bit XORs
	FSM	48* 4-state FSM states
BOWS	Pending Delay Counters	48* 14 (bits) = 672 bits
	Backed-off Queue	48 * 5 (bits)
	Arbitration Logic Changes	
	Delay Limit Estimation Logic (can use functional units when available)	

Table III: DDOS and BOWS Implementation Costs

10,000 cycles requires 14-bits per Pending Delay counter. Adaptive estimation requires division. This can be done using reduced precision computation or by using existing arithmetic hardware when not in use.

VII. Related Work

Numerous research papers have proposed different warp scheduling policies with different goals (e.g., improving latency hiding [23], improving locality [29], reducing barrier synchronization overheads [19], [18], reducing load imbalance overhead across warps from the same CTA [15]). However, none of these scheduling policies have considered the challenge of warp scheduling under inter-thread synchronization. Recent work has addressed the programmability challenges associated with inter-thread synchronization on SIMT architecture [10]. In that work, the authors propose a compiler analysis and transformation that helps eliminate SIMT-induced deadlocks (see Section II). However, it does not address with busy-waiting overheads.

Overheads of fine-grained synchronization have been well studied in the context of multi-core CPU architectures [37], [33], [17], [9], [32]. Ti et al. [17] proposed a thread spinning detection mechanism for multi-threaded CPUs that tracks changes in all registers. Directly applying such a technique to a GPU would be prohibitive given the large register files required to support thousands of hardware threads. Instead, DDOS employs a speculative approach. In [37], the authors propose a synchronization state buffer that is attached to the memory controller of each memory bank to cache the state of in-flight locks. This reduces the traffic propagated to the main memory and the latency of synchronization operations. However, when the buffer is full the mechanism falls back to software synchronization mechanisms. This work builds on the following observation "at any instance during the parallel execution only a small fraction of memory locations are actively participating in synchronization" to maintain a reasonably sized buffer [37]. Although this observation holds true for modestly multi-threaded CPUs, it does not apply to massively multi-threaded SIMT architectures with tens of thousands of threads running in parallel. A similar technique that requires an entry per hardware thread to track locks acquired by each thread is used in [32].

In [36], the authors propose hierarchal queuing at each block in L1 and L2 data caches with the use of explicit acquire/release primitives. Their goal is to implement a blocking synchronization mechanism on GPGPU. In that work, locks can be acquired only on a cache line granularity. Locked cache lines are not replaceable until released. If a cache set is full with locked lines, the mechanism reverts back to spinning for newer locks mapped to the same line. Thus, the efficiency of this mechanism drops as the number of locks increase and starts to perform worse than the baseline [36]. For example, in the hashtable benchmark, the proposal in [36] performs worse than the baseline starting from 512 buckets (in contrast to our proposal, see Section VI-C which consistently outperform the baseline). Further, unlike [36], our work does not assume explicit synchronization primitives which require non-trivial compiler support and/or significant hardware modifications [10] to run correctly on SIMT architectures.

Transactional memory and lock-free synchronization are

other approaches to implement inter-thread synchronization [12], [35], [21]. However, both techniques rely on retries upon failure which lead to overheads and contention that is similar to busy-wait synchronization. GPU transactional memory proposals to date achieve lower performance versus fine-grained synchronization [35], [12]. Similar results have been also reported for lock-free synchronization [22].

VIII. Conclusion

This paper proposes DDOS, a low-cost dynamic detection mechanism for busy-wait synchronization loops on SIMT architecture. DDOS is used to drive BOWS a warp scheduling policy that throttles spinning warps to reduce competition for issue slots allowing more performance critical warps to make forward progress. On a set of kernels that involve busy-wait synchronization, BOWS reduces dynamic instruction count by a factor of 2.1× and reduces memory system accesses by 19% compared to GTO. This leads to an average speedup of 1.4× and dynamic energy reduction by a factor of 1.7×.

Acknowledgments

We would like to thank Mieszko Lis and the reviewers for their insightful feedback. We thank Timothy G. Rogers and his students for GTX1080Ti configuration for GPGPU-Sim 3.2.2 and Scott Peverelle and Tayler Hetherington for their help with GTX1080Ti configuration for GPUWattch. This research was funded in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] AMD Corporation, "Southern Islands Series Instruction Set Architecture," 2012.

[2] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, 1990.

[3] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2009, pp. 163–174.

[4] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU Threads for OpenMP 4.0 in LLVM," in *Proc. LLVM Compiler Infrastructure in HPC*, 2014.

[5] A. Brownsword, "Cloth in OpenCL," Khronos Group, Tech. Rep., 2009.

[6] M. Burtscher and K. Pingali, "An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm," *GPU computing Gems Emerald edition*, 2011.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. IEEE Symp. on Workload Characterization (IISWC)*, 2009.

[8] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proc. IEEE Symp. on Workload Characterization (IISWC)*, 2010.

[9] K. Du Bois, S. Eyerman, J. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, 2013.

[10] A. ElTantawy and T. M. Aamodt, "MIMD Synchronization on SIMT Architectures," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2016.

[11] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2014.

[12] W. W. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2011, pp. 296–307.

[13] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "MemcachedGPU: Scaling-up Scale-out Key-value Store," in *to appear in proceedings of the ACM Symposium on Cloud Computing (SoCC'15)*, 2015, pp. 88–98.

[14] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU workloads," in *ISCA*, 2015.

[15] S.-Y. Lee and C.-J. Wu, "CAWS: criticality-aware warp scheduling for GPGPU workloads," in *Proc. IEEE/ACM Conf. on Par. Arch. and Comp. Tech. (PACT)*, 2014.

[16] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-grained Synchronizations and Dataflow Programming on GPUs," in *Proc. ACM Conf. on Supercomputing (ICS)*, 2015.

[17] T. Li, A. R. Lebeck, and D. J. Sorin, "Spin Detection Hardware for Improved Management of Multithreaded Systems," *IEEE Transactions on Parallel and Distributed Systems*, 2006.

[18] J. Liu, J. Yang, and R. Melhem, "Saws: Synchronization aware gpgpu warp scheduling for multiple independent warp schedulers," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 383–394.

[19] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang, and C. Xu, "Barrier-aware warp scheduling for throughput processors," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 42.

[20] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU Implementation of Inclusion-based Points-to Analysis," in *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, 2012.

[21] P. Misra and M. Chaudhuri, "Performance Evaluation of Concurrent Lock-free Data Structures on GPUs," in *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2012.

[22] N. Moscovici, N. Cohen, and E. Petrank, "POSTER: A GPU-Friendly Skiplist Algorithm," in *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, 2017.

[23] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2011, pp. 308–317.

[24] NVIDIA, "PTX: Parallel Thread Execution ISA Version 3.1," *http://developer. download. nvidia. com/compute/cuda/3*, vol. 1, 2013.

[25] Nvidia, "NVIDIA TESLA V100 GPU ARCHITECTURE," 2017.

[26] NVIDIA Corp., "https://devblogs.nvidia.com/parallelforall/inside-volta/," accessed: July 31, 2017.

[27] NVIDIA, CUDA, "NVIDIA CUDA Programming Guide," 2011.

[28] M. A. O'neil, D. Tamir, and M. Burtscher, "A Parallel GPU Version of the Traveling Salesman Problem," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2011.

[29] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2012, pp. 72–83.

[30] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[31] T. M. Aamodt et al., *GPGPU-Sim 3.x Manual*. University of British Columbia, 2013.

[32] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy, "Supporting fine-grained synchronization on a simultaneous multithreading processor," in *HPCA*, 1999.

[33] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero, "Architectural support for fair reader-writer locking," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2010.

[34] Y. Xu, L. Gao, R. Wang, Z. Luan, W. Wu, and D. Qian, "Lock-based Synchronization for GPU Architectures," in *Proc. Int'l Conf. on Computing Frontiers*, 2016.

[35] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian, "Software Transactional Memory for GPU Architectures," in *Proc. IEEE/ACM Symp. on Code Generation and Optimization (CGO)*, 2014, p. 1.

[36] A. Yilmazer and D. Kaeli, "HQL: A Scalable Synchronization Mechanism for GPUs," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, 2013.

[37] W. Zhu, "Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures," in *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, 2007.