Tornado Warning: the Perils of Selective Replay in Multithreaded Processors

Yongxiang Liu[†] Anahita Shayesteh[†] Gokhan Memik[‡] Glenn Reinman[†]

[†]Computer Science Department, University of California, Los Angeles
[‡]Department of Electrical and Computer Engineering, Northwestern University

Abstract

As future technologies push towards higher clock rates, traditional scheduling techniques that are based on wake-up and select from an instruction window fail to scale due to their circuit complexities. Speculative instruction schedulers can significantly reduce logic on the critical scheduling path, but can suffer from instruction misscheduling that can result in wasted issue opportunities.

Misscheduled instructions can spawn other misscheduled instructions, only to be replayed over again and again until correctly scheduled. These "tornadoes" in the speculative scheduler are characterized by extremely low useful scheduling throughput and a high volume of wasted issue opportunities. The impact of tornadoes becomes even more severe when using Simultaneous Multithreading. Misschedulings from one thread can occupy a significant portion of the processor issue bandwidth, effectively starving other threads.

In this paper, we propose Zephyr, an architecture that inhibits the formation of tornadoes. Zephyr makes use of existing load latency prediction techniques as well as coarse-grain FIFO queues to buffer instructions before entering scheduling queues. On average, we observe a 23% improvement in IPC performance, 60% reduction in hazards, 41% reduction in occupancy, and 48% reduction in the number of replays compared with a baseline scheduler.

1. INTRODUCTION AND MOTIVATION

The performance of an out-of-order superscalar processor relies on the discovery and exploitation of instruction-level parallelism (ILP) and/or thread-level parallelism (TLP). However, the amount of ILP and TLP that a processor can extract is constrained by the design of the instruction scheduler and the size of the issue window. The instruction scheduler and issue window may prove difficult to scale to future technology goals due to the impact of wire latency. Circuit-level studies of dynamic scheduler logic have shown that broadcast logic dominates performance and power [12, 4], which complicates the scaling of the issue queue.

A number of prior studies [5, 7, 8, 6] have examined instruction schedulers that eschew the need for complex wakeup and selection logic and are able to hide the latency of the schedule-to-execute window through speculative scheduling. One example of this is Cyclone [5], which relies on a simple mechanism to predict the expected issue time of each instruction, and then delays the issue of the instruction based on this prediction via scheduling queues. However, misschedulings can occur for the dependents of loads that miss in the first level data cache. In addition, structural hazards in the switchback paths, also known as switchback conflicts/hazards, happen when an instruction wishes to cross from the replay queue to the main queue but cannot do so if that slot is already occupied. These hazards subsequently cause misschedulings of its descendants.

Prior work demonstrates that a large fraction of instructions directly or indirectly depends on load operations [10]. If a load misses, its dependents may be replayed many times before the load completes. Replayed instructions are likely to prevent independent instructions from moving through the switchback queues, further contributing to structural hazards. Hazards are likely to increase in processors with Simultaneous Multithreading (SMT) [15, 16], where overall switchback queue utilization increases. SMT also results in a greater number of loads and more contention in shared cache resources, which can lead to more replays. The positive feedback loop between replays and structural hazards can degrade performance dramatically for an SMT processor, as we will demonstrate in Sections 4 and 5. The feedback loop eventually results in many instructions requiring replay, circulating around the Cyclone queues many times before correctly scheduling. This has been called the *Tornado Effect* [2].

One solution might be to apply simple techniques such as increasing the replay interval, limiting the number of instructions in the scheduler, or flushing threads on a cache miss [14] to prevent tornadoes. However, these techniques inevitably decrease the amount of ILP, and as our experiments show, degrade performance.

Liu et al. [10] developed techniques to predict the execution time of load instructions in the early stages of the pipeline in an effort to scale the size of the instruction window for a conventional instruction scheduler. They use simple FIFOs with different buffer lengths that buffer instructions based on their predicted execution time to prevent instructions from entering the issue window before their operands are ready – effectively providing out-of-order entry into the issue window. However, their technique does not include any dynamic adaptation to mispredicted load latency, and dependents of a misscheduled load can still clog the issue queue and degrade performance.

A natural solution to these challenges might be the addition of load latency prediction techniques from Liu et al. [10] to the Cyclone scheduler. Such a solution should decrease the number of hazards and replays, thereby improving scheduler performance. However, our experimental results demonstrate that this naive combination creates even more structural hazards, and degrades processor performance. The additional structural hazards come from the increased utilization of the scheduling queues. We will explore this impact on the Tornado Effect in more depth in later sections.

As an alternative, we propose Zephyr, an architecture that effectively prevents the formation of tornadoes. Zephyr buffers instructions using coarse-grain FIFO queues. Instructions are released into the scheduling queues only when they are close to their scheduled execution time. This way, we keep the scheduling queue occupancy low to maintain its switchback efficiency. The switchback queues still provide dynamic adaptation to mispredicted instruction latency and selective replay. Our results show that Zephyr is able to eliminate a substantial amount of structural hazards and replays, improving IPC significantly.

However, Zephyr does not eliminate replays completely, and some instructions still enter the scheduler prematurely due to the underestimation of instruction waiting times. This can be due to imperfect load latency prediction or structural hazards such as conflicts for functional units. We further propose to detect the onset of a tornado early on and limit the number of instructions in the scheduling queues for a thread on the verge of forming a tornado. Our results demonstrate that Zephyr with this kind of preventive scheme further eliminates structural hazards and replays, thereby improving overall IPC.

Our contributions over prior work include:

- An investigation of the impact of structural hazards and replays on Cyclone in an SMT environment.
- A quantitative study of the Tornado Effect, characterized by low execution core throughput due to a high volume of misschedulings and structural hazards. This phenomenon may occur in any generalized speculative scheduler using selective replay, but we limit our analysis to the Cyclone scheduler.
- An analysis of the limitations of a simple integration of load latency prediction and Cyclone.
- The Zephyr architecture, which effectively prevents the formation of tornadoes. Zephyr is an integration of a load latency predictor, a sorting engine implemented with different length FIFOs, and a Cyclone-style scheduler. Zephyr is able to improve IPC significantly over both a baseline Cyclone scheduler and a simple integration of Cyclone with a load latency predictor.

The rest of this paper is organized as follows. In Section 2 we discuss prior work, followed by a description of our experimental methodology in Section 3. Section 4 describes the tornado phenomenon observed in the Cyclone scheduler. Several simple remedies are introduced in Section 5. Section 6 presents the Zephyr architecture. Concluding remarks follow in Section 7.

2. RELATED WORK

In this section, we review the most relevant prior work in the areas of instruction scheduling and latency prediction.

2.1 Instruction Scheduling

Ernst et al. [5] proposed Cyclone, a broadcast-free dynamic instruction scheduler with selective replay. The Cyclone scheduler relies on a simple one-pass scheduling algorithm to predict the time when instructions should execute. Once decided, this schedule is implemented with a timed queue structure. In the event of an incorrect scheduling, Cyclone is also able to support efficient selective instruction replay [6]. Execution time prediction is accomplished with a timing table and MAX calculation. The timing table is indexed by logical register, and returns the expected delay until the logical register is ready. Instructions use the timing table to estimate when their input dependencies will be available, and are buffered in a countdown queue for this expected waiting time.

Instructions are injected into the tail of the Cyclone scheduler queue with a prediction of how far the instruction should progress down the countdown queue before turning around and heading back towards execution in the main queue. As mentioned, switching back from the countdown to the main queue can be a source of conflict and must be resolved. Once an instruction reaches the head of the main queue, a table of physical register ready bits is used to determine whether or not all input operands to the instruction are ready. If any operand is not ready, the instruction is routed back to the countdown queue and replays. Cyclone has an optimization option that allows a replayed instruction to consult the timing table to reevaluate its waiting time. We use this optimized version for fair comparison. Cyclone assumes that all loads hit in first level cache. Cache misses will likely result in misscheduled instructions, creating replays.

Hu et al. [7] propose WF-Replay (wakeup free replay), a 32-entry issue queue structure where instructions can be selected for issue from any queue slot. Each queue slot tracks the predicted waiting time for an instruction in a counter, decrementing this waiting time each cycle. The instructions “wake up” and can be selected for issue only when this counter reaches zero. This is designed to avoid the structural hazards in the switchback queues of Cyclone. Instructions must be replayed if their input dependencies are not ready or if they encounter a structural hazard (like insufficient functional units). As in Cyclone, a misscheduling in WF-Replay can potentially cause a chain of further misschedulings. Structural hazards can still spawn even more misschedulings – if one instruction is delayed due to functional unit contention, its dependents can wake up before their inputs are ready. The selection logic is also a potential bottleneck as every queue slot essentially participates in selection. This may limit the size of the scheduling window and the amount of load latency that may be tolerated. The Precheck enhancement [7] checks the ready-bit register to avoid replays. However, this introduces new complexity in that ready-bit register update and instructions selection must complete in one cycle to ensure dependent instructions are executed back-to-back. The WF-Segment enhancement reduces this complexity by selecting instructions only from the segment of 0 waiting time [7]. However, instructions in the other segments need to migrate to the lower segments at the appropriate time, provided that there are no hazards. As in the switchback queues of Cyclone, this instruction migration can result in an increase in structural hazards, which in turn can cause more misscheduled instructions.

Lebeck et al. propose moving instructions dependent on missed loads into a waiting instruction buffer (WIB) to prevent them from clogging the issue queue [9]. Although not explicitly mentioned, the scheduler needs to speculate the presence of a load miss, and then migrate chains of dependent instructions into the WIB. It must also speculate when a load completes, and then migrate these instructions back into the issue queue to avoid exposing the pipeline stages from scheduling to execution.

Both WIB and WF-Replay structures are examples of speculative scheduling mechanisms that can be impacted by misscheduled instructions and structural hazards beyond just contention for functional units (speculatively switching from IQ to WIB and back in the case of the WIB and switching between segments in WF-Segment). In this paper, we focus on Cyclone, which also exhibits misscheduled instructions and structural hazards (in the switchback queues), but our approach could easily be generalized to the WIB, WF-Replay,

Rank by Misschedulings	mcf,equake,ammp,gcc,gzip,lucas,sixtrack,art,parser,twolf,galgel,applu, facerec,vpr,wupwise,apsi,bzip2,mgrid,gap,perl,mesa,crafty,eon,vortex
Rank by Hazards	gcc,equake,art,mcf,applu,vpr,twolf,parser,gzip,eon,ammp,sixtrack,galgel,lucas, wupwise,crafty,perl,gap,facerec,apsi,bzip2,mesa,vortex,mgrid
Average Rank	**equake**,(mcf,**gcc**),**art**,(**gzip**,**ammp**),(**parser**,twolf,applu),sixtrack,(vpr,lucas),galgel, wupwise,facerec,eon,apsi,(perl,*gap*),(*crafty*,*bzip2*),*mgrid*,*mesa*,*vortex*

Table 1: Ranking of benchmarks by the average misscheduled instructions per cycle, and hazards caused by misscheduled instructions. The highest six (in bold) and the lowest six (in italics) from the list of average rank are chosen.

or any other approach that relies on speculative scheduling to avoid conventional wakeup and selection logic.

Ehrhart and Patel propose a speculative scheduling scheme by predicting the instruction waiting times using a PC-indexed history table [3]. However, the waiting time of a static instruction varies dynamically. As a corrective measure, an allowance is added to the predicted waiting time. The allowance is dynamically and progressively adjusted by balancing between the amount of replays and the amount of wasted issue opportunities. Decreasing the allowance will lead to underestimation of waiting times, which causes scheduling replays, while increasing the allowance will cause instructions to wait unnecessarily long, thus wasting issue opportunities. Instructions replays or wasted issue opportunities are unavoidable to guide the process of adjustment. More critically, the time that dependents of loads need to wait varies largely from one to hundreds of cycles depending on cache behavior. A history based prediction scheme performs poorly to capture long latencies caused by cache misses. Therefore, replays due to cache misses will be frequent in such speculative schedulers.

2.2 Latency Prediction

Liu et al. [10] proposed an instruction sorting engine guided by latency prediction. Their pre-scheduler attempts to deliver instructions to a conventional issue queue in the order of their execution, preventing dependents of long latency instruction from entering the issue queue and clogging it. Their scheduler consists of three major components: a latency prediction component, a sorting structure consisting of a few FIFOs with varied lengths, and a Pre-issue buffer where instructions are buffered before entering the issue queue.

Liu et al. [10] use a hybrid approach to predict memory access latency in early stages of the pipeline via several structures. The *Latency History Table* (LHT) is a last value predictor indexed by instruction PC and returns the latency experienced by the last instance of a given load. When the LHT is unable to predict load latency confidently, *Cache Latency Propagation* (CLP) is used. CLP identifies cache misses, and propagates the completion time of cache misses to any aliasing load via a structure that stores the *Status of In-flight Loads* (SILO). An address predictor is used to generate load addresses for the CLP structures. This predicted load address is used to index a miss detection engine [11] which returns *definite miss* or *maybe hit* for that load address. The SILO is accessed in parallel with the miss detection engine and returns the latency of in-flight loads.

Once an instruction's latency is predicted, it is classified and inserted into one of the FIFO queues, based on its predicted latency. Instructions leave the sorting queues to a Pre-issue Buffer (PIB) where they are then delivered to the issue queue in the sorted order.

3. METHODOLOGY

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. We have made significant modifications to SimpleScalar to model Simultaneous Multithreading (SMT) as in [16]. The applications were compiled with full optimization on a DEC C V5.9-008 and Compaq C++ V6.2-024 on Digital Unix V4.0. We simulate at least 100 million instructions for each thread after fast-forwarding an application-specific number of instructions according to Sherwood et al. [13]. The processor configuration used for most simulations is shown in Table 3.

Strong	**ammp**.**gcc** **art**.**parser** **ammp**.**art**.**gzip**.**equake**
Weak	*bzip2.gap* *crafty.mgrid* *bzip2.crafty.mesa.vortex*
Mix	**ammp**.*bzip2* **gcc**.*gap* **art**.*crafty* **parser**.*mgrid* **ammp**.**gzip**.*bzip2.mesa* **art**.**equake**.*crafty.vortex*

Table 2: Applications grouped by strong tornado effects, weak tornado effects, and the mixes.

The benchmarks in this study are taken from the SPEC 2000 suite. As shown in Table 1, we rank the benchmarks by the number of misschedulings per issued instruction and the number of hazards caused by misscheduled instructions. The overall ranking is obtained by sorting the average of the two rankings – benchmarks with the same overall ranking are grouped in parentheses. Benchmarks with more misscheduled instructions and hazards suffer from stronger tornadoes. We select six benchmarks with strong tornadoes to form the "strong" group. Although `mcf` has dramatic tornadoes due to very frequent cache misses, we exclude it to make our results more representative as it receives a very large speedup (over 200%) from our approach. We select six benchmarks with weak tornadoes to form the "weak" group. These two groups, shown in Table 2, form the multithreaded workloads presented in this paper. Three multithreaded runs are formed exclusively from the strong group, three are formed from the weak group, and six are formed from a mix of the strong and weak groups.

4. THE TORNADO EFFECT

In this section, we examine the impact of the Tornado Effect on a speculatively scheduled SMT processor. We will consider the Cyclone scheduler as our representative speculative scheduler.

The Cyclone timing table [5] is indexed by logical register and thread number. It returns the expected ready time of a particular logical register. Our Cyclone scheduler uses a switchback queue length of 100 – all threads share a common switchback queue. Our experiments demonstrate that there is no benefit from further lengthening or shortening the queues, even with latency prediction. We use ICOUNT [16] for thread

Parameters	Value
Issue Width	8
ROBs	256 entries
LSQs	128 entries
Queue Length	Cyclone: 100, PIB: 64, FIFOs: 1,5,10,20, or 150
Cache Block Size	L1: 32B, L2: 64B
Shared L1 Cache	16KB, 4-way, 2-cycle lat.
Shared L2 Cache	512KB, 2-way, 12-cycle lat
Memory Latency	164 cycles
Integer FUs	8 ALU, 2 Ld/St, 2 Mult/Div
FP FUs	2 FP Add, 1 FP Mult/Div
Integer FU Latency	1/5/25 add/mult/div (pipelined)
FP FU Latency	2/10/30 add/mult/div (all but div pipelined)
Branch Pred.	Private 4k BBTB, 8k gshare
Branch Penalty	20, additional 2 for latency prediction

Table 3: Processor Parameters.

selection, where priority to enter the Cyclone queue is given to the thread with the least number of instructions in the Cyclone queues. Hu et. al. [7] simulated Cyclone with an instruction placement strategy that places instructions into paths with different forwarding lengths based on predicted latency. This avoids instructions congregating in rows 0 and 1 of the queues. In our implementation, a round-robin placement strategy is used and it effectively avoids this problem.

Cyclone [5] as described in Section 2 assumes that all loads hit in the first level cache, and schedules their dependent instructions based on this assumption. However, loads that miss in the cache and their dependents account for a large fraction of all instructions [10]. Ignoring long latency memory accesses can result in a large number of replays and structural hazards. The situation becomes worse in a simultaneously multithreading processor where switchback queue utilization is higher and misschedulings on one thread can waste issue bandwidth for other threads. Moreover, an increase in replays increases the probability of switching conflicts. The benchmarks `gap` and `gcc` can help to illustrate this. When run alone, `gap` and `gcc` see an average of 2 replays per issued instruction. Table 4 presents statistics on hazards, replays, and occupancy for Cyclone. As we can see, when they run together on a 2-threaded SMT with Cyclone, around 5 replays per issued instruction are seen. Similarly, `gap` sees 2 switchback hazards per cycle on average – `gcc` sees 3 on average. But when run together, they see around 9 hazards on average each cycle.

One misscheduled instruction can directly cause instructions dependent on that instruction to be misscheduled. Furthermore, replays can create collisions – hazards in the switchback queues of Cyclone (stalling the progress of other instructions), thereby causing even more misschedulings – even on independent instructions. This positive feedback loop can result in the formation of a *tornado* [2]. A tornado is characterized by a period of low useful throughput and a high volume of replays, when instructions that are in the process of selectively replaying circulate through the Cyclone queues over and over again until they are correctly scheduled or squashed from a branch misprediction.

Figure 1 shows a snapshot from the execution of applications `gcc` and `gap`, running together on a 2-threaded SMT processor with Cyclone (processor configuration is as described in section 3). Statistics are collected and averaged over 10-cycle intervals (shown along the x-axis). On the left side, we show the composition of the issue width for each 10-cycle interval. For the 8-issue processor we consider, this figure shows the amount of time the issue bandwidth is used for correctly scheduled instructions from `gcc`, correctly scheduled instructions from `gap`, incorrectly scheduled instructions from `gcc`, incorrectly scheduled instructions from `gap`, and when the issue slots are idle. Instructions that have their input operands ready at issue are allowed to continue to the execution engine and are classified as correctly scheduled instructions. If an instruction has been incorrectly scheduled (i.e. it issues before its input operands are ready), it is replayed to the countdown queues. The right side shows the average Cyclone queue occupancy broken down into the component contributed by `gcc` and `gap`, the presence of cache misses, and the average number of hazards in the switchback queues. The two occupancies statistics are cumulative (i.e. the height of the greater occupancy line is the occupancy of the Cyclone queue for both `gcc` and `gap`) and are shown on the primary y-axis. The number of hazards uses the secondary y-axis.

The cache miss experienced by `gcc` causes a chain of misscheduled instructions that occupy the instruction bandwidth of the processor from interval 5 to interval 15. This period is characterized by relatively high queue occupancy by the thread that spawned the tornado (`gcc`) and a large number of hazards. After interval 15, `gap` is able to get some of the issue bandwidth and issue correctly scheduled instructions. Before interval 15, `gap` had instructions that could have issued, but were not even able to get into the Cyclone queues due to the dramatic number of replays. The Cyclone occupancy of `gcc` eventually drains as the instructions that made up the tornado are scheduled correctly after the cache miss is satisfied. However, the lapse in incorrectly scheduled instructions is shortlived before another cache miss starts another tornado.

5. DEALING WITH TORNADOES

In this section, we explore a number of techniques to combat the Tornado Effect on speculative schedulers.

5.1 Reducing Replay Frequency

One approach to reduce the formation of tornadoes is to replay instructions less frequently. A natural choice is a replay interval of 12 cycles – the L2 cache latency. This effectively coarsens the granularity of replays. Since misscheduled instructions replay every 12 cycles, it is possible for an instruction to wait for a longer time than required by true data dependence latency.

However, our experiments show an average 4% degradation in performance when using this approach. Figure 2 demonstrates the performance of this longer replay interval, Replay12. While a few application combinations like `art.parser`, `ammp.art.gzip.equake`, `art.equake.crafty.vortex` see improvement, most of the benchmarks perform worse with a longer interval. The benchmarks `art`, `ammp`, and `equake` have L2 cache miss rates of over 25%. This approach helps such applications to reduce hazards, switchback queue occupancy, and replays. In addition, these applications have longer average load latencies, and are therefore able to tolerate waiting a few extra cycles to speculatively issue a misscheduled instruction. Longer average load latencies mean significantly more replays and longer tornadoes - and therefore a coarser granularity can prove highly effective here. However, it degrades the performance of most of the other applications, as these do not have such long average load latencies. The impact of waiting an unnecessarily longer amount of time outweighs any benefit from reduced replays.

We also explored a replay interval of 6 cycles, and observed a similar degradation in performance.

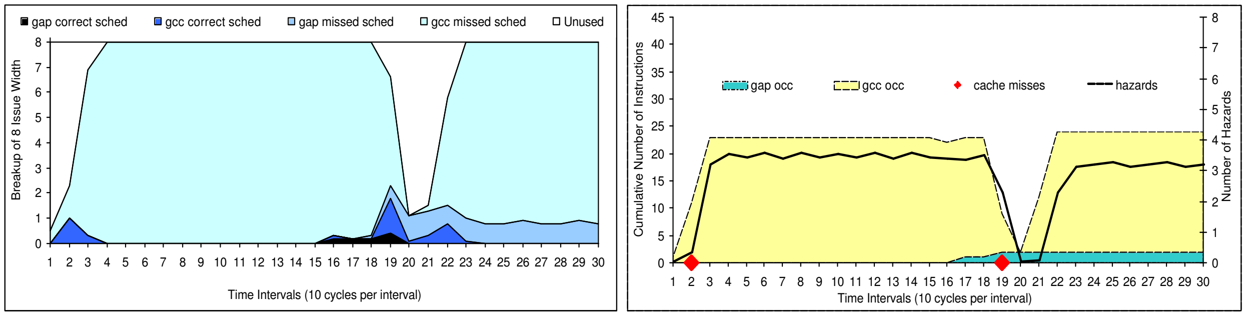

Figure 1: Snapshot of Cyclone Baseline

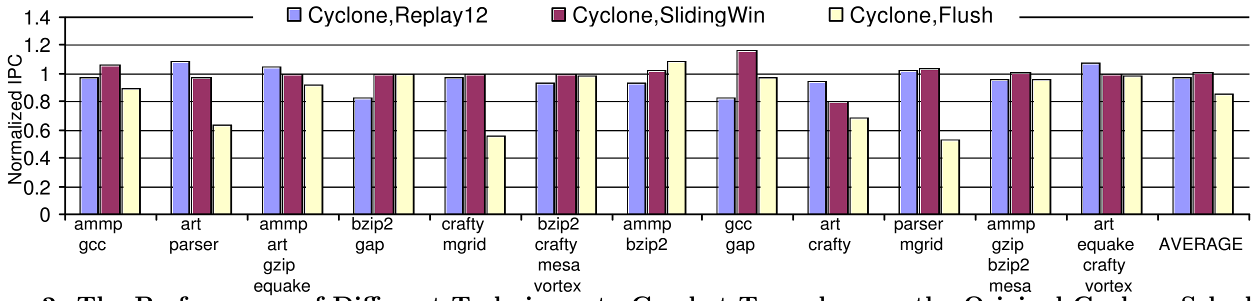

Figure 2: The Performance of Different Techniques to Combat Tornadoes on the Original Cyclone Scheduler.

5.2 Limiting Threads to Prevent Tornadoes

Once a tornado develops, it exhibits symptoms like excessive replays and a high volume of switchback hazards in the scheduling queues. One solution would be to detect tornadoes in their early stages and take preventive measures to avoid the full onset of the tornado.

We have developed effective algorithms to detect tornadoes in the context of the Cyclone scheduler. Although we have considered a number of different policies for dealing with tornadoes, the most effective approach we have found is the *Sliding Window* scheme. In this approach, we set a per-thread window size (*WIN*) which caps the total number of instructions from a given thread that are allowed in the Cyclone queues. This limit can be increased or decreased for each thread to control tornado formation.

The Sliding Window can be easily implemented as a part of the ICOUNT policy. To adequately guide this mechanism, we need to know when a thread is likely to spawn a tornado (*WIN* should be decreased) or when a thread is too severely restricted, potentially impairing ILP (*WIN* should be increased). The former condition will be referred to as *overflow* and the latter condition will be referred to as *underflow*. We will determine whether a thread is in underflow or overflow by considering how many replays are required in a given period of time. Too many replays means that the thread may be forming a tornado – too few replays means that the thread may be too constrained and is not being aggressive enough in speculative scheduling.

We define two thresholds: the overflow threshold (OF_{th}), which determines when a thread is likely to form a tornado, and the underflow threshold (UF_{th}), which determines when a thread is probably too severely restricted. To ensure that *WIN* does not oscillate wildly, we require that a thread exceeds OF_{th} for more than $Decr_{th}$ consecutive cycles before decrementing *WIN*. Similarly, a thread must not replay more than UF_{th} for $Incr_{th}$ consecutive cycles for *WIN* to be incremented.

Our implementation of this algorithm uses three counters, an *Increment_Flag*, and a *Decrement_Flag* per thread. The first counter (*R_Counter*) counts the number of replays per cycle. It resets every cycle. The second counter (*OF_Counter*) counts the number of consecutive cycles a thread remains in *overflow*. The *OF_Counter* resets whenever the thread is not in *overflow*. The third counter (*UF_Counter*) counts the number of consecutive cycles in *underflow*. The *UF_Counter* resets whenever the thread is not in *underflow*.

If *OF_Counter* $>= Decr_{th}$, the *Decrement_Flag* is turned on. Similarly, the *Increment_Flag* is turned on if *UF_Counter* $>= Incr_{th}$. When either flag is triggered, the triggering counter (OF_Counter or UF_Counter) resets. Both flags are reset every cycle, after being tested to see if *WIN* will change in a given cycle. We performed extensive experiments to tune these parameters (results not shown). Our data demonstrates that in an 8-way cyclone scheduler, the following parameters can detect tornadoes effectively: $OF_{th} = 6$, $UF_{th} = 2$, $Decr_{th} = 10$, $Incr_{th} = 5$.

WIN is reduced upon the detection of tornado symptoms (i.e. if the *Decrement_Flag* is on), is increased when instructions are smoothly scheduled (i.e. if the *Increment_Flag* is on), and not changed if neither flag is set. Initially *WIN* is set to "unlimited", which does not cap the number of instructions at all. Upon detection of a potential tornado (*Decrement_Flag*), *WIN* is set to 24. When *Decrement_Flag* is set, this value is decremented by 4, and when *Increment_Flag* is set, this value is incremented by 4. Incrementing beyond 24 sets *WIN* to "unlimited". *WIN* cannot be decreased below 4. To maintain fairness, *WIN* is reset to "unlimited" every 10,000 cycles.

Overall, we observe only a slight performance improvement of 1%. As shown in Figure 2, `gcc.gap` sees the most speedup (16%), while the remaining benchmarks see less than 6% improvement, some even performing worse than baseline cyclone. Unfortunately, this approach can significantly limit the amount of ILP that can be exploited from many applications. When Cyclone is operating smoothly, and there are no tornado effects, high occupancy in the cyclone queues can be extremely constructive, allowing the processor to see a larger window of issuable instructions. The benchmark mix of `gcc.gap` suffers from a dramatic number of tornadoes, and therefore is able to see benefit from this approach.

5.3 Exploiting TLP

Prior work [14] has demonstrated that overall throughput can be improved in an SMT architecture with conventional issue queue by stalling or flushing a thread when that thread suffers an L2 cache miss. The intuition here is that the thread is consuming resources that could be used for other threads while waiting for the long latency operation. The authors propose several mechanisms to detect an L2 miss (detection mechanism) and two ways of acting on a thread once it is predicted to have an L2 miss (action mechanism). The detection mechanism that presents the best results is to predict a miss every time a load spends more cycles in the cache hierarchy than needed to see a hit in the L2 cache, including possible resource conflicts (15 cycles in the simulated architecture of [14]). Two action mechanisms provide good results. The first is STALL, which consists of fetch-stalling the offending thread. The second, FLUSH, flushes the instructions after the load that missed in the L2 cache, and then stalls the offending thread until the load is resolved. As a result, the offending thread temporarily does not compete for resources, and what is more important, the resources used by the offending thread are freed, giving the other threads full access to them.

We apply the same approach to Cyclone. In this paper, an ideal version of FLUSH is considered. We detect an L2 cache miss ideally by probing the cache structures after selection by ICOUNT. This should give FLUSH the best performance potential by avoiding any wasted issue bandwidth when an L2 miss will occur. The thread is restarted once the load instruction that missed goes to writeback. However, this approach gives an average 15% slowdown as shown in Figure 2. Only application mix `ammp.bzip2` has a noticeable speedup of 9%. This is because the L2 miss rate of `ammp` is very high and few independent instructions closely follow an L2 cache miss in program order. The program `bzip2` has few L2 misses but has very rich ILP. When instructions from `ammp` are flushed due to a L2 miss, `bzip2` is able to improve by utilizing more of the available issue width.

However, a program may have a significant amount of ILP that can be exploited even after a level 2 miss is encountered, `art` being a notable example. In addition, other threads that are NOT flushed may have little ILP to exploit the scheduling resources emptied by the flushed thread. FLUSH does poorly on such applications. The typical examples are `art.parser`, `mgrid.crafty` and `parser.mgrid`. We observe that `art` and `mgrid` have relatively higher L2 miss rates. The programs `crafty` and `parser` have few cache misses, however, unlike `bzip2`, they benefit little from increased scheduling resources due to the lack of ILP. Consequently, these applications observe over a 30% slowdown.

5.4 Cyclone+: Cyclone Extended with Load Latency Prediction

As shown in the previous sections, conservatively guessing an L1 cache hit for the latency of loads that do not alias stores is a major cause of tornadoes in the Cyclone scheduler. It seems much of the problem would be solved if the actual latency for each load was known.

To verify this, we extend Cyclone with the load latency prediction techniques recently proposed in [10]. The techniques in [10] capture 83% of the load misses, and 99% of the cache hits. More accurate load latency prediction should allow Cyclone to more precisely schedule instructions and reduce the number of switchback structural hazards and replays.

In this paper, we use the hybrid load latency prediction in [10]. Address and latency predictors, as well as the miss detection engine and SILO, are shared by the threads. We limit the number of load latency predictions to two in each cycle to reduce the number of ports required on these structures. Our experiments show there is no benefit from increasing the number of ports any further.

Figure 3 shows the performance results for this extended Cyclone architecture. The first bar shows performance for the baseline Cyclone and the second shows the performance for Cyclone enhanced with latency prediction. Contrary to our expectations, predicting load latency only improves the performance of a handful of benchmarks (like `art.parser` and `art.crafty`) and actually degrades performance for a few application mixes (like `ammp.gcc` and `gcc.gap`). On average, Cyclone+ only shows a slight IPC speedup of 4%.

Our investigation shows that this is due to a dramatic increase in stalls for some applications. Table 4 (presented on page) presents the average number of structural hazards seen in the switchback queues per cycle, the average occupancy of the queues, and the average number of replays seen per cycle. Note that these behaviors are bursty and tend to occur in clusters – however, the average behavior is still useful for purposes of comparison. The first column shows the benchmark mixes we considered, and the first two columns of the hazards and structural hazard results show data for the baseline Cyclone and Cyclone enhanced with load latency prediction (Cyclone+) respectively. Cyclone+ sees significantly more structural hazards – except for a few application mixes (like `art-parser` and `art-crafty`) where there is actually a drop in hazards.

As shown in Table 4, we observe a substantial increase in queue occupancy. On average, queue occupancy is 45% larger with Cyclone+ than baseline Cyclone - with some applications seeing double the occupancy with Cyclone+. When load latency prediction is applied, although the descendants of missed loads obtain their waiting times accurately, these waiting times are much longer than baseline Cyclone which assumes loads always hit the cache. These instructions can progress further towards the end of the Cyclone queues – the furthest point in the switchback queues from the execution engine. This increases the occupancy of the scheduling queues, thus creating more switchback hazards. When queue occupancy is high, a tornado can be formed.

As an example, an instruction is not switched on time due to a structural hazard (i.e. queue conflict due to high occupancy or replay). The dependents of that instruction may have to be replayed even if they do not encounter any hazards. Such replays occupy queue spaces, which can further introduce more conflicts and replays. When a tornado is active, the scheduler experiences extremely low useful throughput, and a high volume of replays and hazards. The benefit of reduction in replays through load latency prediction is effectively canceled for some applications by the dramatic increase in structural hazards and switchback queue occupancy – all of which feeds the Tornado Effect.

5.4.1 *Tornadoes in Cyclone+*

Figure 4 shows a snapshot from the execution of `gcc.gap` in Cyclone+. We observe relatively larger queue occupancies and relatively more structural hazards. At interval 2, a cache miss results in the formation of a tornado. By interval 17, the cache miss is satisfied and a few instructions from `gcc` are issued. Before the old instructions drain, however, a new cache miss brings more instructions into the scheduling queue. The tornado is sustained and further deteriorates.

5.4.2 *Improving Cyclone+*

The tornadoes affect Cyclone+ more than the original Cyclone scheduler, as evidenced by the increased occupancies, switchback hazards, and replays. In this section, we attempt to mitigate this by applying techniques to prevent the formation of tornadoes.

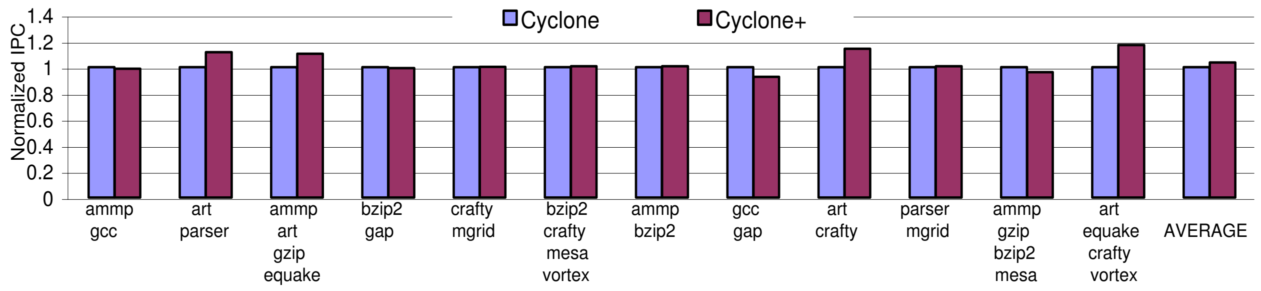

Figure 3: Baseline Cyclone (Cyclone) and Cyclone extended with load latency prediction (Cyclone+).

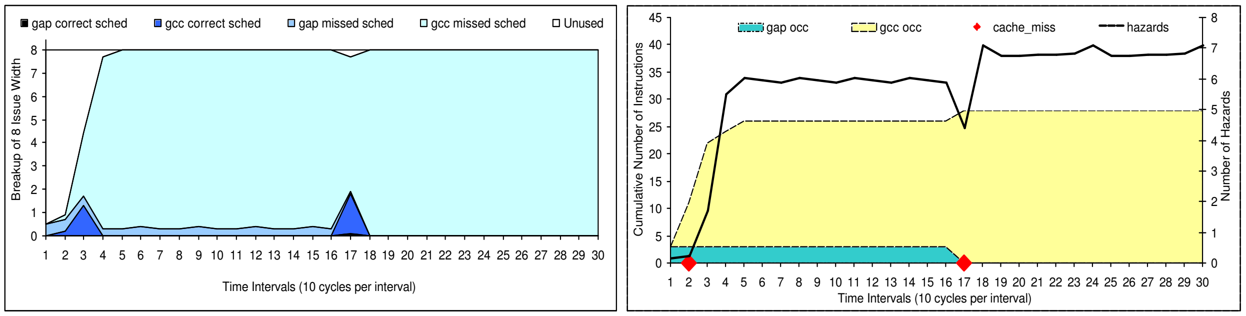

Figure 4: Snapshot of Cyclone Extended with Load Prediction (Cyclone+)

Figure 5 shows the normalized IPC (with respect to the original baseline Cyclone) of Cyclone extended with load latency Prediction (Cyclone+), the further extension of Cyclone+ with our sliding window approach (Cyclone+, SlidingWindow) and Cylone+ with FLUSH (Cyclone+,Flush). Cyclone+ with SlidingWindow performs 2% better on average than Cyclone+ alone. As was the case when applied to baseline Cyclone, the sliding window approach only improves a handful of application mixes that suffer from extremely strong tornadoes, but does not help and even hurts other application mixes. Cyclone+ with FLUSH has a comparable performance to that of baseline Cyclone with FLUSH, often degrading performance as it limits the available ILP.

6. ZEPHYR DESIGN AND EVALUATIONS

Our prior attempts to combat tornadoes either sacrificed ILP to reduce tornadoes or increased queue occupancy beyond Cyclone's ability to effectively switchback instructions. In this section, we propose Zephyr, an architecture that reduces replays and structural hazards without sacrificing ILP and without straining Cyclone's queue structure. Zephyr has the potential to improve any speculative scheduler plagued by the tornado effect, but we consider the impact on Cyclone alone for brevity.

6.1 The Zephyr Scheduler

Figure 6 illustrates the high-level architecture of the Zephyr scheduler. Zephyr prevents the formation of tornadoes by sorting instructions in their predicted execution order and then using Cyclone to adapt to misschedulings. Zephyr effectively allows instructions to enter Cyclone out of order. Figure 6 is divided into three components: the latency prediction engine (delineated with a dotted box), the coarse-grain sorting engine (colored in grey), and the fine-grain sorting engine (surrounded with a dashed box). The latency prediction engine features a timing table (similar to [5]) that is accessed on every instruction, and a latency prediction structure (similar to [10]) that is accessed on every load instruction. The result of this prediction stage is a predicted wait time for each instruction before all input operands are ready.

Instructions are then enqueued in the coarse-grain sorting engine – the FIFOs. Instructions with very short waiting times are placed into FIFOs with a buffering length of 1, and can progress to the next stage in one cycle. Instructions with longer waiting times are placed into different FIFOs with buffering lengths of more than one. During the classification, we round down waiting times to the closest granularity queue available, ensuring instructions are not delayed beyond their estimated waiting time. We adopt the same queue configurations as in [10], but double the number queues to accommodate the additional bandwidth of SMT. We have six 0-slot queues, four 5-slot queues, two 10-slot queues, two 20-slot queues, and two 150-slot queues. Instructions enter the FIFOs in program order, but can leave the FIFOs out of order.

Instructions are buffered in the PreIssue Buffer (PIB) after sorting. Each thread has a PIB, and we use the ICOUNT [16] thread selection policy to choose a PIB from which to pull instructions. ICOUNT measures the number of instructions from each thread that are currently in the fine-grain sorting engine. Note that ICOUNT cannot pull instructions from an empty PIB. Since the coarse-grain sorting has absorbed some of the expected wait time from instructions in the PIB, if there are no available instructions in the PIB, it indicates that there is currently no ILP to exploit in a given thread. Therefore, the sorting engine enables a more intelligent ICOUNT which has some notion of available parallelism in a given thread – and will selectively pull from threads that have such parallelism.

Instructions leave the PIB and enter the fine-grain sorting engine (Cyclone). Here, instructions may encounter structural hazards or may need to be replayed if they have been misscheduled. However, the coarse-grain sorting engines of Zephyr are able to absorb some of the instruction latency to keep the countdown/replay queue occupancy low. This reduces the structural hazards in Cyclone and inhibits the formation of tornadoes.

Figure 7 shows the relative performance of Zephyr, using Cyclone as the baseline. Zephyr has an average of 13% speedup over Cyclone. This benefit comes from the more accurate waiting time prediction as load misses are taken into account, as well as the reduction in occupancy, and scheduling hazards. As seen from the graph, applications with strong tornadoes have an average of 22% speedup. The mixed applications have an average of 13% speedup. The benefits on applications with weak tornadoes are small. But still, an av-

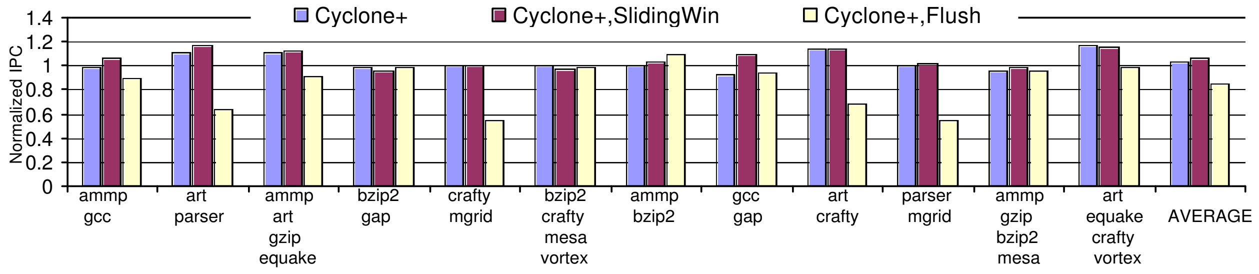

Figure 5: the IPC Performance of Cyclone Extended with Load Prediction (Normalized with Baseline Cyclone)

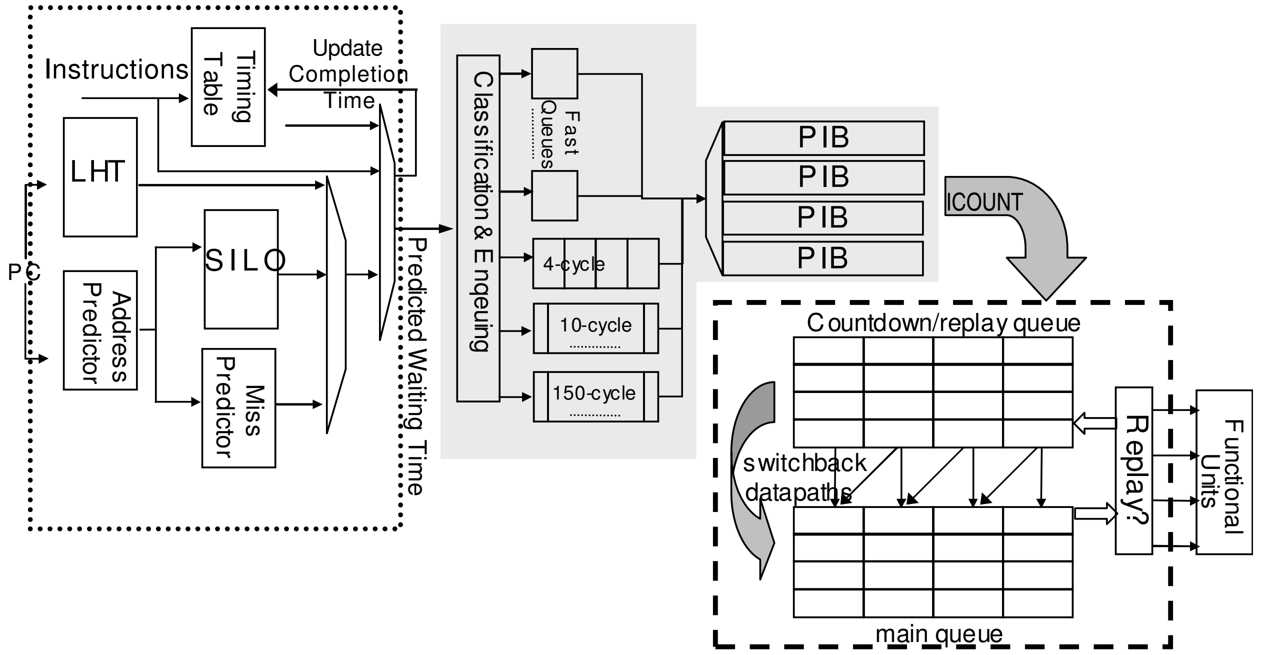

Figure 6: Zephyr Scheduler Architecture

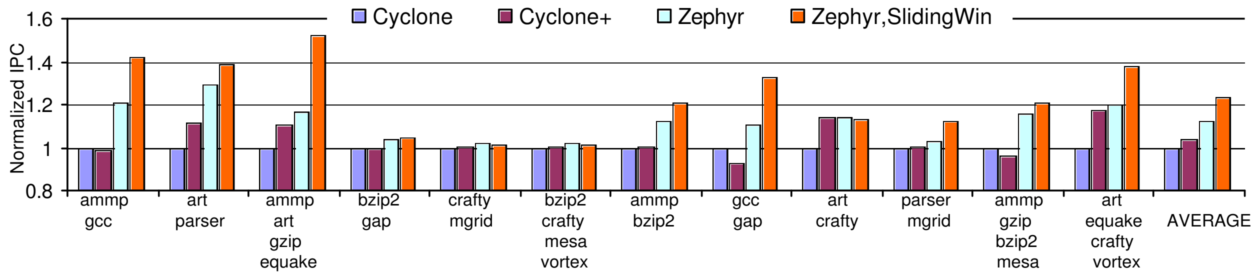

Figure 7: Speedup of Zephyr and Zephyr with Sliding Window

erage of 3% speedup is observed.

As mentioned in [5], the greatest contributors to IPC loss with Cyclone are structural hazards (switchback conflicts) and replays in the scheduling queues. Zephyr reduces the number of hazards by buffering the instructions in the coarse-grain sorting structure. Instructions entering the scheduling queues are expected to have their operands ready in a short time. We observe an average of 31% reduction in queue occupancy compared with baseline Cyclone. Table 4 illustrates this reduction with the data labeled "Z" – Zephyr. Zephyr observes a 42% reduction in structural hazards from the original Cyclone, and a 60% reduction from our Cyclone+. Zephyr also observes a 24% reduction in the number of replays relative to the original Cyclone design. The applications with strong tornadoes and the mixed applications observe large reductions in structural hazards and replays. Correspondingly, these applications see significant speedup over the Cyclone baseline in Figure 7. Applications from the group of weak tornadoes see less of a drop in queue occupancy, hazards, and misschedulings, and therefore see a smaller performance improvement.

6.2 Zephyr with the Sliding Window

Zephyr cannot eliminate all replays. There are only a finite number of sorting queues, and therefore the fine grained sorting in the Cyclone queues can still cause switchback hazards even for instructions with highly predictable latencies. Moreover, if the latency prediction engine is not able to confidently report a latency for a load instruction, it conservatively guesses the latency of a cache hit. On average, 17% of load misses are not captured. In addition to underestimating loads that are not address or latency predictable, an instruction's waiting time may be underestimated due to other types of structural hazards such as contention for functional units, memory ports, and the memory bus. This underestimation potentially issues instructions prematurely, spawning replays and potentially more structural hazards.

We consider using the preventive measures presented in Section 5 to reduce replays spawned by unpredictable loads and structural hazards. The best performing approach with Zephyr is the Sliding Window from Section 5.2. We use this approach to throttle instructions from entering the Cyclone queues via the PIBs on a per thread basis. We use the same set of parameters as described in section 5.2.

Since Zephyr is able to safeguard ILP by effectively buffering those instructions that are waiting on input dependencies, we expect our Sliding Window to be more effective in reducing tornado formation without impacting performance.

Figure 7 shows the performance of the Sliding Window on Zephyr. This approach shows significant speedup from the baseline Cyclone. On average, we observe a 23% improvement in performance, 60% less hazards, a 41% drop in occupancy, and 48% fewer replays. This represents a further performance improvement from Zephyr alone. Applications in the strong group observe further speedups ranging from 40% to 50%. The mixed applications observe an average of 23% speedup. A single thread degradation of 13% is seen by `gcc` in the 2-thread run `gcc.gap`, and a degradation of 15% by `parser` in the 2-thread run `parser.mgrid`. Other than this, we see no more than a 4% per thread degradation.

Load latency prediction is unable to capture 10% to 30% of load misses in `art`, `ammp`, and `equake`. In `gcc`, `parser` and `gzip`, frequent conflicts for FUs cause a great deal of underestimation. In these application mixes, the Sliding Window scheme successfully limits the underestimated instructions from entering the scheduling queues before the onset of a tornado. As a result, these applications experience substantial performance improvement.

The applications from the weak group see fewer tornadoes. Hence, these applications see no further improvement using our sliding window. The mix of `bzip2.crafty.mesa.vortex` even experiences a slight degradation of 1% relative to Zephyr alone. This is because the application has very few tornadoes, and the sliding window can still limit ILP.

Figure 8 shows a snapshot of 300 cycles from the execution of `gap.gcc` using Zephyr with our Sliding Window. In general, we observe more accurate scheduling and throughput and fewer hazards and Cyclone queue occupancy. When cache misses are captured by the predictor, their dependents cannot enter the scheduling queues immediately, but have to go through the sorting queues and PIBs. They can only enter the scheduling queue when they are close to their predicted issuing time. This reduces unnecessary queue occupancy and switchback hazards. As can be seen in the figure, instructions are soon issued after they enter the scheduling queues.

As shown in the figure, there are still a substantial amount of premature instructions being replayed though at much reduced scale. This is because instructions can enter the scheduling queue several cycles in advance due to the finite granularity in the sorting stage. This can also be due to underestimated waiting time caused by load latency mispredictions, hardware hazards, and scheduling hazards. However, Zephyr with our Sliding Window can take preventive measures before premature instructions can form tornadoes. When early tornado symptoms are observed via the (*Decrement_Flag*), the underestimated instructions are forced to wait longer in the PIBs. This helps these instructions to enter the scheduling queues at the right time. Overall, Zephyr with our Sliding Window prevents tornadoes, and is able to schedule instructions correctly with sustained useful throughput.

6.3 Zephyr vs Cyclone vs Conventional Issue Queue

Our baseline cyclone scheduler sees a 10% average IPC degradation compared to a 32-entry conventional wakeup-and-select issue queue with perfect speculative scheduling. This degradation is slightly less than seen in [5] due to the round-robin placement strategy (Section 4). The degradation mainly comes from inaccurate estimation of waiting time, scheduling replays, and structural hazards. Zephyr effectively combats these problems, and is able to see an average IPC speedup of 11% over a conventional 32-entry 8-way issue queue.

7. SUMMARY

While Cyclone is able to provide scalable instruction scheduling for deeply pipelined processors, the structural hazards and replays that are possible with Cyclone can severely degrade performance in a multithreaded environment. Useful issue bandwidth can be wasted on misscheduled instructions, limiting the amount of thread level parallelism that can be exploited. There exists a positive feedback loop between structural hazards and replays that can result in the misscheduling of a large portion of Cyclone-issued instructions. This is characterized by instructions from different threads continually shuffling around the scheduler, with low useful scheduling throughput. This *Tornado Effect* can also happen with other replay-based schedulers that employ speculative scheduling.

In this paper, we present Zephyr, an architecture that intelligently schedules instructions to avoid tornadoes in multithreaded processors. Scheduling instructions dependent on loads is extremely challenging as load access time is highly nondeterministic, particularly when considering loads that alias with in-flight memory requests. Assisted by prior work

	hazards per cycle				occupancy				replays per instruction			
	C	C+	Z	ZWin	C	C+	Z	ZWin	C	C+	Z	ZWin
ammp.gcc	21	23	12	5	101	126	63	36	6.2	6.1	4.7	1.6
art.parser	12	10	6	1	56	101	40	28	6.0	4.4	3.1	1.4
ammp.art.gzip.equake	25	41	10	11	82	166	49	63	4.7	4.0	3.9	2.2
bzip2.gap	6	7	3	3	34	37	21	21	1.4	1.4	1.3	1.2
crafty.mgrid	5	5	4	3	29	30	21	20	1.5	1.5	1.4	1.2
bzip2.crafty.mesa.vortex	6	6	5	4	33	36	27	24	1.0	1.0	1.0	0.9
ammp.bzip2	22	22	12	9	87	92	55	42	2.2	2.2	1.9	1.4
gcc.gap	9	16	4	1	45	88	30	16	5.3	5.6	3.7	1.5
art.crafty	10	2	1	1	51	57	19	18	4.3	1.3	1.2	1.0
parser.mgrid	5	7	5	4	31	51	33	28	2.6	2.4	2.1	1.4
ammp.gzip.bzip2.mesa	17	20	6	8	61	71	33	40	1.7	1.7	1.3	1.2
art.equake.crafty.vortex	13	21	10	5	56	128	52	40	3.3	2.4	2.6	1.2

Table 4: Comparison of the number of replays and structural hazards (i.e. switchback conflicts) in the scheduling queues. Symbols: "C" — Cyclone, "C+" — Cyclone+, "Z" — Zephyr, "ZWin" — Zephyr Sliding Window.

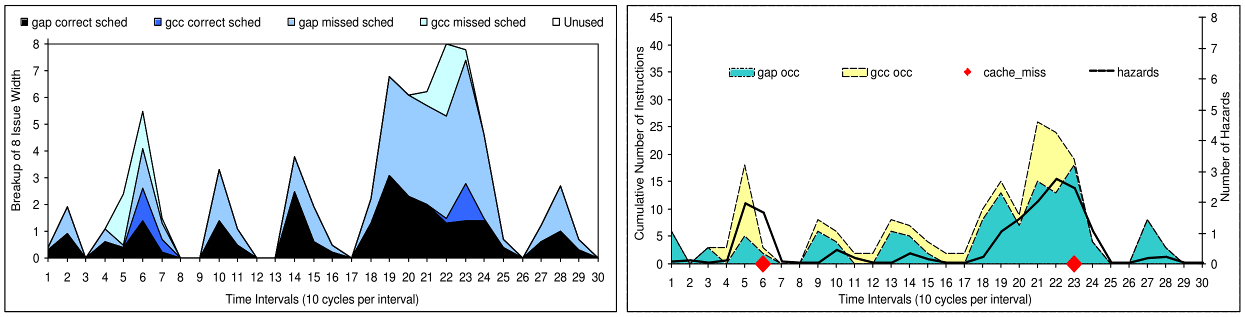

Figure 8: Snapshot of Zephyr with Sliding Window

on load latency prediction, Zephyr is able to predict instruction waiting times accurately, including instructions dependent on loads. Zephyr buffers instructions in a coarse-grain sorting engine, resulting in an approximate execution ordering for each thread. This ordering is buffered in a per-thread buffer that can then steer thread selection for execution. Instruction scheduling is still done via Cyclone, with fine-grain sorting handled by the Cyclone switchback queues. Cyclone is also able to dynamically adapt to unpredictable load latencies and misschedulings using selective replay.

We further propose a Sliding Window algorithm to enhance Zephyr, which dynamically caps the number of instructions allowed in the scheduling queues by observing the symptoms that lead to the formation of a tornado. With this option, the Zephyr architecture delivers consistently higher IPC than the baseline Cyclone. Our experiments show Zephyr has a 23% improvement in IPC performance, 60% less hazards, a 41% drop in occupancy, and 48% fewer replays compared with a baseline scheduler.

8. REFERENCES

[1] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[2] D. Carmean. Distinguished lecturer series presentation. 2004.

[3] E. Ehrhart and Sanjay J. Patel. Reducing the scheduling critical cycle using wakeup prediction. In *Proceedings of The Tenth International Symposium on High-Performance Computer Architecture (HPCA10 2004)*, pages 222–231, 2004.

[4] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[5] D. Ernst, A. Hamel, and T. Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, June 2003.

[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[7] J. S. Hu, N. Vijaykrishnan, and M. J. Irwin. Exploring wakeup-free instruction scheduling. February 2004.

[8] I. Kim and M. Lipasti. Understanding scheduling replay schemes. In *Proceedings of The Tenth International Symposium on High-Performance Computer Architecture (HPCA10 2004)*, pages 138–147, 2004.

[9] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, May 2002.

[10] Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. Scaling the issue window with look-ahead latency prediction. In *Proceedings of the 18th annual international conference on Supercomputing (ICS 2004)*, pages 217–226. ACM Press, 2004.

[11] G. Memik, G. Reinman, and W. H. Mangione-Smith. Just say no: Benefits of early cache miss determination. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[12] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[13] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[14] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, 2001.

[15] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture (ISCA)*, June 1995.

[16] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. *SIGARCH Comput. Archit. News*, 24(2):191–202, 1996.