The Calm Before the Storm: Reducing Replays in the Cyclone Scheduler

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. Recently, Cyclone was proposed as a broadcast free scheduler with a significant reduction in logic on the critical scheduling path. However, Cyclone suffers from performance degradation due to the presence of structural hazards in the scheduler and wasted issue opportunities on selective replays.

The impact of replays and structural hazards becomes even more severe when using Simultaneous Multithreading for higher throughput. The higher utilization of the SMT processor can cause a positive feedback loop between hazards and replays that can result in extremely low performance. This loop results in instructions shuffling around and around in the Cyclone queue, creating typhoons *in the Cyclone scheduler that are characterized by extremely low scheduling throughput and a high volume of hazards and replays.*

In this paper, we propose Zephyr, an architecture that inhibits the formation of typhoons. Zephyr makes use of existing load latency prediction techniques as well as course-grain FIFO queues to buffer instructions before entering scheduling queues. Our simulation results show that Zephyr reduces the scheduling hazards and replays significantly. On average, Zephyr observes a 15% speedup over a baseline Cyclone 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 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 be 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 [7, 2].

Recently Ernst et al. proposed Cyclone [3], a broadcast-free scheduler without the need for complex wakeup and selection logic. Cyclone relies on a simple mechanism to predict the expected issue time of each instruction, and then delay the issue of the instruction based on this prediction via switchback queues. However, misschedulings can occur for the dependents of loads that miss in the first level data cache and for the dependents of instructions delayed by structural hazards in the switchback queues. Therefore, Cyclone is also able to support efficient selective instruction replay [4] in the event of an incorrect scheduling.

Prior work demonstrates that a large fraction of instructions directly or indirectly depend on load operations [5]. If a load misses, its dependents may be replayed many times before the load completes. Replayed instructions can also prevent independent instructions from moving through the switchback queues, further contributing to structural hazards. Hazards can also increase in processors with Simultaneous Multithreading (SMT) [10], where overall switchback queue utilization increases. More utilization also leads to more 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 our results. The feedback loop eventually results in many instructions requiring replay, circulating around the Cyclone queues many times before correctly scheduling. We refer to this effect as *typhoons* in the Cyclone Scheduler.

Liu et al. [5] 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 FIFO queues to sort 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 issue queue and degrade performance.

A simple solution to these challenges might be the addition of load latency prediction techniques from [5] 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 approach can suffer by creating even more structural hazards, degrading processor performance. The additional structural hazards come from the increased utilization of the Cyclone switchback queues. We will explore this impact on the Typhoon Effect in more depth in later Sections.

As an alternative, we propose Zephyr, an architecture that effectively prevents the formation of typhoons. Zephyr buffers instructions using course-grain FIFO queues. Instructions are released into the Cyclone queues only when they are close to their scheduled execution time. In 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 replay, substantially improving IPC.

Load latency prediction is not perfect, particularly when loads are not address predictable. We further consider stalling threads with less predictable loads in Zephyr to improve overall throughput. Our results demonstrate that Zephyr with this stalling option further eliminates structural hazards and replay, 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.
- The discovery of the limitations of a simple integration of load latency prediction and Cyclone.
- The observation of the Typhoon Effect, characterized by low execution core throughput due to a high volume of misschedulings and structural hazards. This phenomenon may occur in any generalized replay-based schedulers, but we limit our analysis to the Cyclone scheduler.
- The Zephyr architecture, which effectively prevents the formation of typhoons in Cyclone. Zephyr is able to improve IPC significantly over both a baseline Cyclone scheduler and a simple integration of Cyclone with load latency prediction.

The rest of this paper is organized as follows. In Section 2 we discuss the prior work, followed by an description of our experimental methodology in Section 3. Section 4 presents the architecture design of Zephyr, the challenges and our experimental results. Concluding remarks follow in Section 5.

2 Related Work

In this Section, we review the most relevant prior work to our study.

2.1 Instruction Scheduling

Ernst et al. [3] 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 that additionally supports efficient selective replay in the event of an incorrect schedule. 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. Switch back in cyclone 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 operands is not ready, the instruction is routed back to the countdown queue and replays. Cyclone assumes that all loads hit in first level cache (unless they are determined to alias a store). Cache misses will likely result in instruction misscheduling, creating replays.

2.2 Latency Prediction

Liu et al. [5] 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 FIFO queues, and a Pre-issue buffer where instructions are buffered before entering the issue queue.

Liu et al. [5] 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 [6] 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 sorting queues, based on its predicted latency. A locking mechanism ensures that instructions can not leave the sorting queues before their parent instructions to avoid deadlock. Instructions leave the sorting queues to a Pre-issue Buffer (PIB) where they are then delivered to the issue queue in the order of their predicted execution.

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 [10]. We randomly select 10 floating-pint and 10 integer benchmarks from the SPEC 2000 benchmarking suite. We select 11 random combinations of four-thread benchmarks. We then select the first two threads from these benchmarks as two-thread benchmarks to be evaluated. 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. [8].

Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. We model an 8-way out-of-order multithreaded processor. All threads share a 16KB 4-way set associative L1 data cache with a 2 cycle access latency and 32 byte block size; a private 16KB 2-way set associative L1 instruction cache with a 32 byte block size; and a unified 512KB 4-way set associative L2 cache with a 12-cycle access latency and 64 byte block size. The L2 cache is shared among all threads. The total round trip time to memory is 164 cycles. The processor has 8 integer ALU units, 2 load/store units, 2 FP adders, 2 integer MULT/DIV, and 1 FP MULT/DIV. Each thread has a 4K-entry BBTB and an 8K-entry gshare private predictor. The minimum branch misprediction penalty is 20 cycles.

4 Zephyr Design and Evaluations

In this Section, we examine a SMT processor enhanced with the Cyclone scheduler, and propose an architecture to reduce replays and structural hazards within this architecture.

The timing table [3] 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 the queues, even with latency prediction. We use ICOUNT [11] for thread selection, where priority to enter the Cyclone queue is given to the thread with the least number of instructions in Cyclone queues.

Cyclone [3] as described in Section 2 assumes all loads hit in first level cache, and schedules their dependent instructions based on this assumption. However, missed loads and their dependents account for a large fraction of all instructions [5]. Ignoring long latency memory accesses results 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 any replays potentially cause switching conflicts. The benchmarks `gap` and `gcc` can help to illustrate this. When run alone, `gap` sees around 2 replays per issued instruction and `gcc` sees around 6 replays per issued instruction. When run together on a 2-threaded SMT with Cyclone, around 37

replays per issued instruction are seen. Similarly, gap sees 5 structural hazards per cycle on average in the Cyclone switchback queues – gcc sees 8 on average. But when run together, they see around 19 hazards on average each cycle.

As a first pass, we extend cyclone with the load latency prediction techniques recently proposed in [5]. More accurate load latency prediction should allow Cyclone to precisely schedule instructions and reduce the number of switchback structural hazards and replays. Load latency is predicted using the hybrid approach proposed in [5]. Address and latency predictors as well as the miss detection engine and SILO are shared by the threads. level 1 caches. For architectures with shared level 1 caches (not considered in this paper), the SILO and cache miss detection engine can also be shared. We limit the number of load latency predictions from the structures in [5] to two in each cycle to reduce the number of ports required on these structures. Our experiments show there is no benefit from increasing this number.

Figure 1 shows single and multithreaded 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 ammp-art and equake-applu) and actually degrades performance for a few application mixes (like gap-gcc and bzip-gzip).

Our investigation shows that this is due to a dramatic increase in stalls for some applications. Table 1 presents the average number of structural hazards seen in the switchback queues per cycle and the average number of replays seen per cycle. Note that both of 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 (Base) and Cyclone enhanced with load latency prediction (LoadLat). LoadLat sees significantly more structural hazards – except for a few application mix (like ammp-art and apsi-crafty-gap-art where there is actually a drop in hazards.

We also observe a substantial increase in queue occupancy. On average, queue occupancy is 41% larger with LoadLat than Base - with some applications seeing 118% more occupancy with LoadLat. 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 instruction can progress farther towards the end of the Cyclone queues – the furthest point in the switchback queues from the execution engine. This increases the occupancy of the Cyclone queues. When queue occupancy is high, Cyclone develops into a *Typhoon*. As an example, a parent instruction P is not switched on time due to a structural hazard (i.e. queue conflict due to high occupancy or replay). The child instruction C may have to be replayed even if it does not encounter any hazards. Such replays occupy queue spaces, which further introduce more conflicts and replays. When a typhoon is active, the scheduler experiences extremely low useful throughput, and a high volume of replays and hazards. The reduction in replays through load latency prediction is effectively canceled for some application by the dramatic increase in structural hazards and switchback queue occupancy – all of which feeds the Typhoon effect.

4.1 Zephyr Scheduler

Figure 2 illustrates the high-level architecture of the Zephyr scheduler. Zephyr prevents the formation of Typhoons by sorting instructions in their predicted execution order and then using Cyclone to adapt to misschedulings. The Figure 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 [3] that is accessed on every instruction, and a latency prediction structure [5] 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 of [5]). Instructions with very short waiting time are placed into the fast FIFO queues, and can progress to the next stage in one cycle. Instructions with longer waiting time are placed into FIFOs queues for more than one cycle. 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 [5], 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

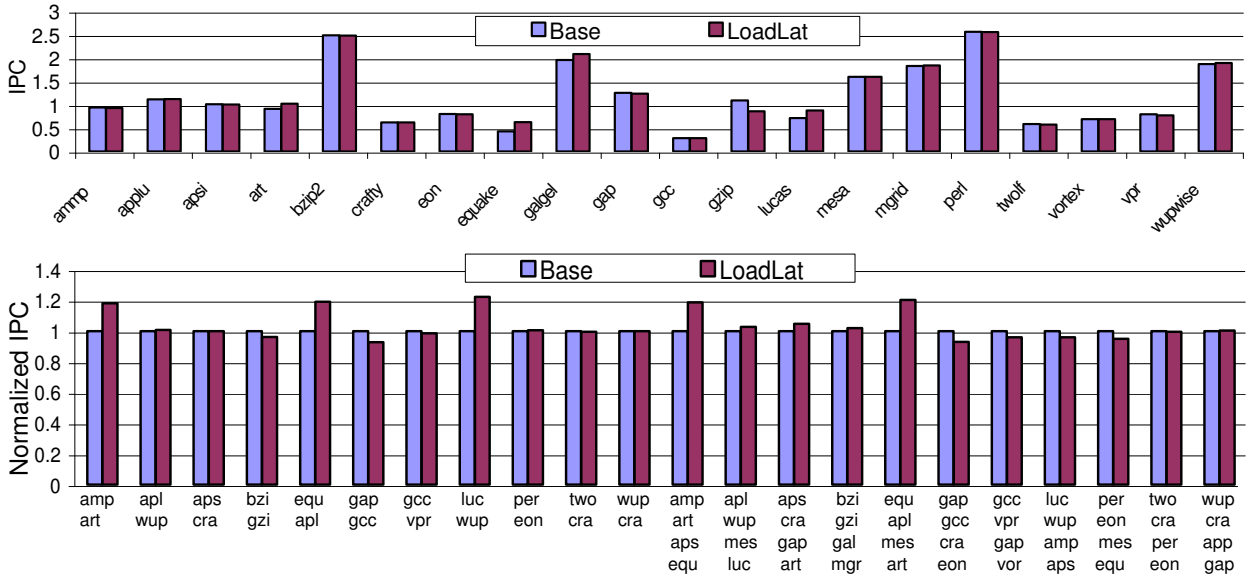

Figure 1: Single and multithread results: baseline Cyclone (Base) and Cyclone extended with load latency prediction (LoadLat). SPEC names arr abbreviated.

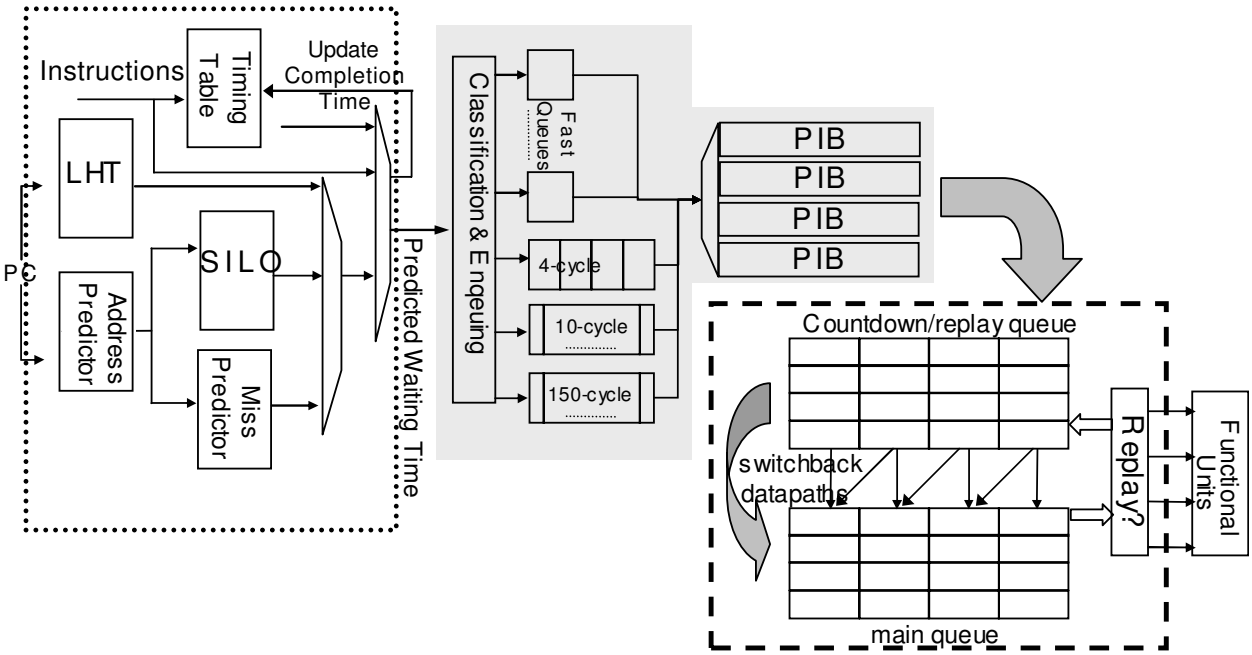

Figure 2: Zephyr Scheduler Architecture

thread has a PIB, and we use the ICOUNT [11] 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.

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, coarse-grain sorting engines of Zephyr are able to absorb some of the instruction latency to keep the countdown/replay queue occupancy low. This should reduce the structural hazards in Cyclone and inhibit the formation of Typhoons.

Figure 3 shows the IPC speedup of Zephyr, using Cyclone as the baseline. Zephyr has an average of 8% speedup over Base. This benefit comes from the more accurate waiting time prediction as load misses are taken into account, as well as the reduction in structural hazards.

As mentioned in [3], 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 35% reduction in queue occupancy comparing to baseline Cyclone. Table 1 illustrates this reduction with the data labeled Zephyr. Zephyr observes 52% reduction in structural hazards from the original Cyclone, and 60% from Cyclone using load latency prediction. The benchmarks `ammp.art`, `bzip.gzip` and `gap.gcc` all observe large reductions in structural hazards and replays. Correspondingly, these benchmarks see significant speedup over the Cyclone baseline in Figure 3. However, some applications such as `gcc-vpr-crafty-eon` and `twolf-crafty-perl-eon` still see some degradation. We notice that `crafty`, `eon`, and `gcc` have a relatively high branch misprediction rate that ranges from 5% to 12%. Zephyr has two additional pipe stages for instruction sorting and enqueuing in the preissue buffer. In addition, we add two extra cycles of branch misprediction penalty to compensate the latency of load latency predictions. The resulted degradation cancels off the benefits from Zephyr in these application mixes.

4.2 Zephyr with Thread Stalling

In [5], if the latency prediction engine is not able to confidently report a latency for load instruction, it conservatively guesses the latency of a cache hit. Our analysis shows that if a load is not address/latency predictable, it will likely miss in the cache. Treating such loads as cache hits, potentially brings instructions prematurely into the scheduling queue introducing potential structural hazards and replays. We therefore propose an extension to Zephyr – the ZephyrStall approach – where a thread is stalled when a load instruction with unpredictable latency is encountered. The thread is stalled until the unpredictable load address is available. The unpredictable load is tagged. When the tagged load is issued, it signals the stalling thread to continue. This prevents the descendants of such loads from causing potential hazards and replays, and at the same time frees the scheduling resources to other needy threads. This technique is very similar to the architecture Tullsen et al. [9] used to handle threads with long latency loads. However, there are two important differences between our approach and their prior work. We do not flush a thread on every L2 miss – we only *stall* a thread from issuing *if it is unpredictable*. There are a lot of load misses that are predictable using techniques in [5] and this helps to avoid degrading the performance of the thread suffering from the L2 miss.

With ZephyrStall, an average of 15% speedup is observed as shown in Figure 3. The additional performance mainly comes from the additional reduction of hazards and replays. The single thread degradation of 25% is seen by `apsi` in the 4-threaded run – `ammp-art-apsi-equake`, 28% by `applu` in the 4-threaded run – `eqauke-applu-mesa-art`. Other than this, we see no more than 10% degradation in per thread.

Table 1 shows that ZephyrStall achieves even further reduction in scheduling hazards, at a reduced rate of 48%. ZephyrStall efficiently reduce the number of replays as indicated in Table 1. We observe that ZephrStall reduces the queue occupancy by 44% from the baseline Cyclone.

5 Summary

While Cyclone is able to provide scalable instruction scheduling for deeply pipelined processors, the struc-

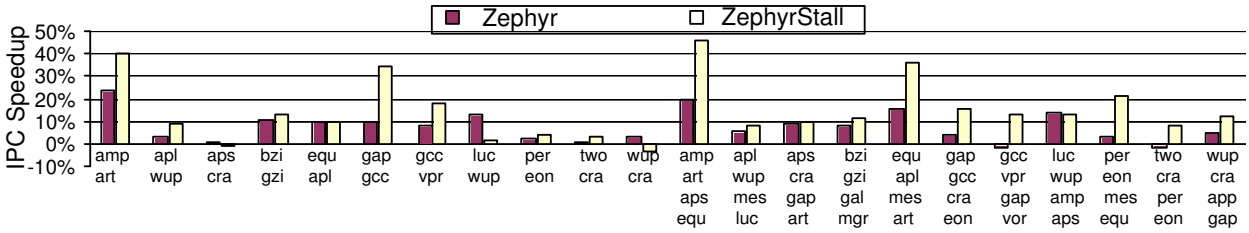

Figure 3: Performance speedup of Zephyr and ZephyrStall over the baseline Cyclone

	hazards per cycle				replays per cycle			
	Base	LoadLat	Zephyr	ZephyrStall	Base	LoadLat	Zephyr	ZephyrStall
ammp.art	52	39	27	13	6.3	5.6	5.4	2.9
applu.wupwise	20	20	14	9	5.1	5.0	5.1	3.7
apsi.crafty	4	4	2	2	1.6	1.6	1.5	1.2
bzip2.gzip	23	32	11	11	4.9	5.0	4.6	4.3
equake.applu	41	47	39	5	6.7	6.3	6.5	2.0
gap.gcc	19	34	11	4	6.2	6.0	5.1	2.3
gcc.vpr	16	28	9	2	6.6	6.4	5.4	1.5
lucas.wupwise	48	81	34	41	6.2	5.5	6.0	6.2
perl.eon	16	15	6	6	3.7	3.6	3.3	3.2
twolf.vpr	18	20	8	3	6.1	5.9	5.4	2.1
wupwise.crafty	9	9	7	4	3.9	3.8	3.8	2.9
ammp.art.apsi.equake	47	78	26	19	6.3	5.8	6.1	3.9
applu.wupwise.mesa.lucas	31	59	25	28	5.4	5.2	5.3	5.1
apsi.crafty.gap.art	23	20	10	8	5.2	3.8	3.8	2.9
bzip2.gzip.galgel.mgrid	23	27	12	15	4.1	4.0	3.9	3.7
equake.applu.mesa.art	33	55	22	14	6.1	5.4	5.8	3.8
gap.gcc.crafty.eon	15	27	9	5	5.2	5.2	4.7	2.6
gcc.vpr.gap.vortex	15	30	9	4	5.6	5.5	5.2	2.5
lucas.wupwise.ammp.apsi	46	101	24	31	5.5	5.4	5.2	5.2
perl.eon.mesa.equake	22	44	12	11	5.2	5.1	5.2	3.6
twolf.vpr.perl.vortex	17	21	6	8	4.8	4.7	4.7	3.4
wupwise.crafty.applu.gap	19	23	11	10	4.9	4.8	4.8	3.6

Table 1: Comparison of number of replays and structural hazards (i.e. switchback conflicts) in Cyclone count-down/replay queues.

tural 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 *typhoon* 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 typhoons in the Cyclone scheduler for 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 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 stall option in Zephyr, which prevents instructions dependent on loads from entering the scheduling queues prematurely when their latencies are unpredictable. The Zephyr architecture delivers consistently higher IPC throughput than the baseline Cyclone. Our experiments show Zephyr achieves 15% IPC speedup, and substantial reductions in scheduling hazards and replays.

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

[3] 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.

[4] 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.

[5] Y. Liu, A. Shayesteh, G. Memik, and G. Reinman. Scaling the issue window with look-ahead latency prediction. In *Proceedings of the 18th International Conference on Supercomputing (ICS)*, June 2004.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.